A Scalable Embedded DSP Core for SoC Applications

C. Panis
Carinthian Tech Institute
c.panis@cti.ac.at

U.Hirnschrott, S.Farfeleder, A.Krall
Vienna University of Technology
uli@complang.tuwien.ac.at
stefanf@complang.tuwien.ac.at
andi@complang.tuwien.ac.at

G.Laure, W.Lazian
Infineon Technologies Austria
glaure@sbox.tu-graz.ac.at
wlazian@sbox.tu-graz.ac.at

J. Nurmi
Tampere University of Technology
jari.nurmi@tut.fi

Abstract

Increasing system complexity of SoC (System-on-Chip) and SiP (System-in-Package) applications leads to the strong demand of platform based solutions. Software programmable embedded cores are required to provide flexibility to these platforms. Compared with dedicated hardware implementations the provided flexibility leads to increased silicon area and power dissipation, which is problematic for high volume products.

This paper introduces xDSPcore, a scalable embedded DSP processor which allows to scale major architectural features to application specific requirements. Compatibility issues caused by different core versions are covered by the support of efficient programming in high-level languages like C, which is achieved by an optimizing C-compiler and by a compiler friendly core architecture. A particular core definition is specified by a XML based configuration file.

1. Introduction

Increasing complexity of SoC and SiP applications increases the demand on powerful embedded cores. The flexibility provided by the usage of software programmable cores quite often leads to an increase in consumed silicon area and an increased power dissipation. Therefore dedicated hardware has been favored over software-based platform solutions. The picture is changing. Increasing mask costs (due to the use of advanced process technologies) and difficulties to enter such high-volume products to the heterogeneous market (to justify the high non-recurring cost) together increase the pressure for developing product platforms. These platforms are used for a group of applications such that software executed on programmable core architectures is used for differentiating the products.

Embedded general purpose core architectures cause an inefficient use of the core resources. Each class of applications has requirements, which cannot be efficiently covered by a general-purpose architecture in terms of area and power consumption. To close the gap between dedicated hardware and software-based solutions, a platform-specific adaptation of the core architecture is required.

For embedded DSP cores (Digital Signal Processors) an additional problem is virulent. Due to non-orthogonal core architectures (which have been preferred because of obtaining better performance and less area consumption when mapping DSP algorithms onto a processor) DSPs are still often programmed manually in assembly language. The price for the better usage of the available processor resources is an architecture-dependent description of the algorithms which makes changes in the core architecture difficult and costly (due to compatibility issues) and prohibits application-specific adaptations. Therefore products based on a programmable core architecture are sticking to the same architecture for a long time frame, even if not state-of-the-art any more. Additional risks and costs by changing the core architecture thus lead to solutions that are not competitive.

This paper introduces xDSPcore, a scalable embedded DSP core for SoC applications. The first part of the paper gives an overview of existing solutions and positions the xDSPcore project. The second part gives an overview of the main architectural features of the core architecture. The third part introduces some architectural decisions influenced by the compiler development.

2. State of the art

This section briefly introduces available DSP concepts and differentiates the core project presented in this paper from them.

2.1. Traditional DSP core architectures

The latest announcements and products of traditional DSP core architectures are the Starcore SC1200/SC1400 [1] announced by Starcore LCC, which is a cooperation between Motorola, Agere and Infineon Technologies, and Blackfin [2], the outcome of a cooperation between Analog Devices and Intel Inc. Both core concepts are RISC-based load-store architectures, claiming to be efficiently programmable in high-level languages like C/C++. The ISA (Instruction Set Architecture) and the micro architecture are fixed. This prevents application

0-7803-8558-6/04/$20.00 ©2004 IEEE.

specific modifications, which would be needed to close the gap between hardwired implementations and software.

2.2. Scalable core architectures

The best known examples of scaleable architectures are provided by ARC and Tensilica. Both are based on traditional microcontroller architectures. Therefore efficient implementation of traditional DSP algorithms is difficult and issues such as minimizing the worst-case execution time are mostly ignored. Software support for using DSP-specific features is inadequate. The philosophy of integrating "just an additional MAC unit" directs the focus towards increasing theoretical performance, instead of an analysis of the overall system performance.

2.3. Architecture description languages

LISA (Language for Instruction Set Architecture) from Coware, mainly developed at RWTH Aachen [3], is the best known architecture description language. Later projects are for example, the ArchC project in Brazil [4]. The concept of defining your own specific core architecture to fulfill the requirements of your application code sounds unbeatable at first glance. In the very large solution space provided by an architectural description language, very many core architectures can be defined, but only a few of them are compatible with the development of an optimizing high-level language compiler. Most design teams are interested in integrating a core into their SoC or SiP solution. To use an architecture description language like LISA and to generate efficient solutions, deep knowledge of processor architecture is required.

Unfortunately, automatic generation of high-level language compilers for DSP cores is still not feasible. Even with approaches like COSY from ACE, the quality of the code produced by an automatically generated compiler is poor. This may mislead designers into making bad decisions in regard to architectural modifications. The problem can be summed up as follows - understanding how the application requirements affect the core architecture needs an efficient high-level language compiler, but this cannot be generated automatically with acceptable quality and certainly not for each core.

2.4. xDSPcore

The xDSPcore approach introduced in this paper is our attempt to solve the problems described above. xDSPcore is a general-purpose DSP core architecture; it is based on a RISC load-store model and it enables efficient execution of traditional DSP algorithms. During definition of the microarchitecture system aspects like the possibility of minimizing the worst-case execution time have been considered. To close the gap between hardwired ASIC implementations and software based system solutions, the core concept enables scaling of the main architectural features, while the micro-architectural model is not changed. The micro-architecture has been defined so that the requirements for developing an optimizing C compiler are satisfied. The C compiler permits analysis of the design space of specific application code. To keep the validation and verification effort low, a unique configuration file based on XML is used. The configuration file allows to scale core features while taking care that the effects of all changes are propagated to hardware, tools and documentation.

3. Core architecture

xDSPcore is based on a modified dual Harvard load-store architecture [5][6]. VLIW (Very Long Instruction Word) is the chosen programming model. Static scheduling allows shifting dependency resolution into the C compiler and therefore reduces the complexity of the core architecture. However VLIW produces poor code density. To overcome this problem, xLIW (scalable Long Instruction Word) is introduced [7]. xLIW is based on VLES (Variable Length Execution Set), which enables decoupling of fetch and execution bundles. Compared to VLES, xLIW allows a reduction in the size of the program memory port (and therefore reduces the wiring effort) without limiting the peak performance of the core architecture. To speed up the execution of inner loops of DSP algorithms, we introduced an instruction buffer that overcomes the possible bandwidth mismatch resulting from the reduced size of the program memory port. A typical program memory port size would be 4 instruction words whereas a xLIW instruction can use up to 10 instruction words. The buffer also reduces transition activity at the program memory port during execution of hardware and software loops. Once the loop body is fetched, no further program memory access is needed, and this reduces dynamic power dissipation.

3.1. Register file

For load-store architectures, the register file is a central part of the core architecture. Separate instructions are used for moving data between a register file and data memory; all arithmetic instructions use operands stored in the register file. The register file of xDSPcore is split into three parts: data register file, address register file including modifier registers and a separate branch file (which is not fully visible to the instruction set architecture). The data register file supports three types of register sizes. Data registers are 16-bit wide, whereas long registers are 32-bit wide and accumulator registers are either 40 or 64 bits wide depending on the core variant (however the size of the registers is scalable according to the application). Two consecutive data register can be accessed as one long register. The long register with

additional guard bits for an increased range of fixed point values forms the accumulator register. 64-bit wide accumulator registers are used for xDSPcore variants which execute four MAC operations in SIMD instructions concurrently.

The structure of the register file is orthogonal. There are no restrictions on the usage of registers for special instructions. The same is true for the address register. Each 32- bit wide address register has a modifier register, which is used for modulo addressing and bit-reversal address mode (introduced for a more efficient implementation of FFT algorithms).

The third part of the register file is the branch file. The branch file contains status information of the core architecture. It holds static information about the register content (e.g. sign and zero flag for each register of the data register file) and dynamic information updated by the data flow (e.g. overflow flags or flags indicating loop status). Static information is updated each time register contents are changed. Therefore it is not necessary to update static information during interrupt handling or task switching. However dynamic information has to be handled by the user. A separate branch file has been chosen to relax the number of read/write ports associated with the register files, which is already stressed by the orthogonality requirements.

3.2. Predicated Execution

Status information stored in the branch file is used for conditional branch instructions and to control predicated execution.

xDSPcore supports a rich set of predicated or conditional execution instructions, thus reducing the frequency of branch instructions and therefore avoiding branch delays [8]. When executing control code, predicated execution can significantly reduce the number of unused branch delay slots [9].

3.3. Pipeline

xDSPcore features a RISC-like three phase pipeline: instruction fetch, decode and execute. Each of the phases can be split into several clock cycles, which results in higher clock frequencies. However, splitting the instruction fetch into several clock cycles increases the number of branch delays. Spending several clock cycles on the execution phase increases load-use and define-use dependencies. Compensation methods for the arising drawbacks (bypasses, branch prediction) are available [9] but increase core complexity and silicon area.

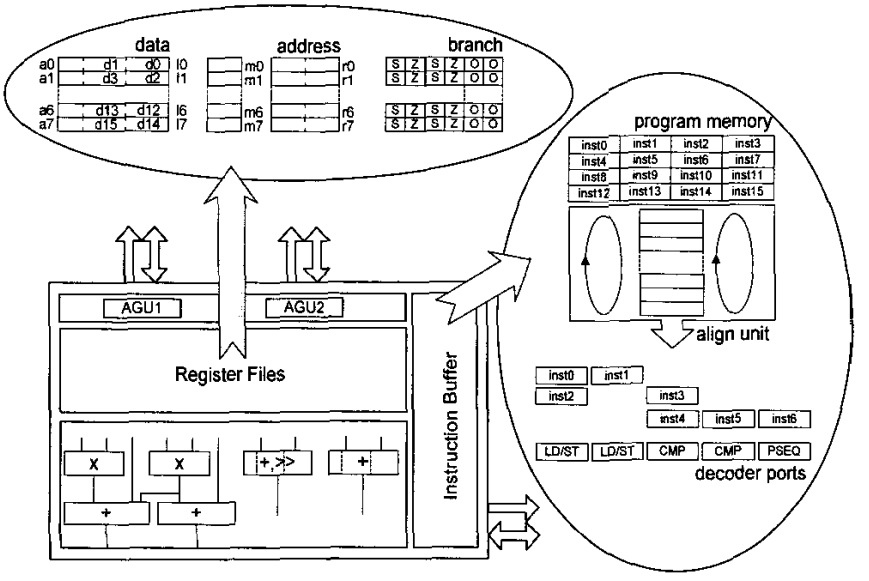

Figure 1: Core overview

3.4. SIMD

Code density is a measurement of how efficiently an algorithm can make use of the chosen core architecture. To increase code density and exploit available parallelism, xDSPcore supports SIMD instructions. Filter operations can be speeded up by executing two MAC instructions in parallel (adding the results of the two multiplications into one accumulator, which reduces register pressure). The ALU data paths can be used for executing two instructions in parallel on the same data path (e.g. two 16-bit additions taking place on the same adder structure). SIMD on the ALU paths permits both increased code density and a

reduction in the clock cycles needed to execute an algorithm. To reduce the number of move operations between registers in the register file, SIMD cross operations are supported. The high and low words of long registers can be used for SIMD operations, combining one high operand with one low operand in a crosswise manner, and vice versa.

4. Compiler aspects

This section introduces some examples for design considerations to enable the development of an optimizing C-compiler providing efficient code generation.

4.1. Orthogonal instruction set

Orthogonal instruction set in the term as defined in [6] requires instructions with free choose of resources and to each instruction the complement instruction (e.g. an add requires a subtract). None of the instructions of xDSPcore contains implicit operand addressing (e.g. as in the SC140 [1]) or micro architectural limitations in the sense of mode dependent resource allocation, which can be e.g. found at ARM Thumb [10].

4.2. No mode registers

Mode registers are often introduced to increase code density, which enables a kind of *paging mechanism* for the instruction set. But mode registers limit instruction scheduling whereas the instructions introduced to set and reset modes can lead to a decreased code density. xDSPcore does not support mode registers, whereas available features are encoded in regular instructions.

4.3. Simple issue rules

The lean pipeline structure allows omitting bypass circuits for operand forwarding. Further it reduces the number of issue rules which simplifies resource checking during instruction scheduling. No implicit dependencies are introduced between instructions; predicated execution is destination register based and therefore the implicit dependency between condition generation and evaluation is reduced.

4.4. Orthogonal register set

None of the registers of the register file is assigned to dedicated instructions. Both aspects would limit the register allocator and instruction scheduler and can lead to a reduced usage of the available hardware resources.

5. Results

Application code benchmarks like EFR (Enhanced Full Rate Encoder) compiled for xDSPcore and competitive core architectures illustrates an increased code density of about 50%, resulting in only half of the program memory [11]. The first prototype implementation approaching end of this year is done in a 130nm Chartered, 6-metal layer CMOS technology, featuring 150 MHz military worst case conditions (5-way VLIW, two MAC Units).

6. Conclusion

This paper introduces xDSPcore, an embedded DSP core architecture, whose major architectural features can be scaled to application-specific requirements. To overcome compatibility issues programming of the core architecture in architecture independent high-level languages like C is supported. Beside a brief overview of the key aspects of xDSPcore, design considerations are introduced, reflecting requirements of the compiler.

7. Acknowledgment

The work has been supported by the EC under the project SOC-Mobinet (IST-2000-30094), the Christian Doppler Gesellschaft and Infineon Technologies Austria.

8. References

[1] "SC140 DSP Core Reference Manual", Motorola Inc., MNSC140CORE/D, Revision 3, November 2001.

[2] "Blackfin DSP Instruction Set Reference", Digital Signal Processor Division, Analog Devices Inc., First Revision, March 2002.

[3] "Lisa 2.0 Language Reference Manual, Manual RM_2002.02", LisaTek, February 2002.

[4] "The ArchC Architecture Description Language v0.8.1, Reference Manual", www.archc.org, 2004.

[5] J. L. Hennessy, D. A. Patterson, *Computer Architecture. A Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo CA, 1996.

[6] P. Lapsley, J. Bier, A. Shoham and E.A. Lee, *DSP Processor Fundamentals, Architectures and Features*, IEEE Press, New York, 1997.

[7] C. Panis, R. Leitner, H. Grünbacher, J. Nurmi, "xLIW – a Scaleable Long Instruction Word", *in Proc. The 2003 IEEE International Symposium on Circuits and Systems (ISCAS 2003)*, Bangkok, Thailand, May 25-28, 2003, pp. V69-V72

[8] C. Panis, U. Hirnschrott, A. Krall, G. Laure, W. Lazian, J. Nurmi, "FSEL – Selective Predicated Execution for a Configurable DSP Core", *in Proceedings of IEEE Annual Symposium on VLSI (ISVLSI-04)*, Lafayette, Louisiana, USA, February 19-20, 2004, pp. 317-320.

[9] D. Sima, T. Fountain, P. Kacsuk, *Advanced Computer Architectures: A Design Space Approach*, Addison Wesley Publishing Company, Harlow, 1997.

[10] "An Introduction to Thumb", Advanced RISC Machines Ltd., Version 2.0, March 1995.

[11] C. Panis, "Scalable DSP Core Architecture Addressing Compiler Requirements", PhD thesis at Tampere University of Technology, Tampere, Finland, August 2004.