SAMIRA: A SIMD-DSP architecture targeted to the Matlab™ source language

Gordon Cichon
Vodafone Chair, TU-Dresden
D-01062 Dresden
+49-351-463-35674
cichon@ifn.et.tu-dresden.de

Pablo Robelly
Vodafone Chair, TU-Dresden
D-01062 Dresden
+49-351-463-35674
robelly@ifn.et.tu-dresden.de

Hendrik Seidel
Vodafone Chair, TU-Dresden
D-01062 Dresden
+49-351-463-33442
seidel@ifn.et.tu-dresden.de

Additional Authors: Torsten Limberg and Gerhard Fettweis, Vodafone Chair, TU-Dresden

This paper presents a novel processor architecture tailored to the efficient execution of digital signal processing algorithms efficiently that are developed using the Matlab™ programming language. Based on the synchronous transfer architecture (STA) and SIMD processing the goal of the architecture is the minimization of the overhead associated with programmability when compared to hardwired solutions. With more than 40% of the core area being dedicated to performing actual signal processing computations, the SAMIRA processor provides a high computing performance with little overhead required for programmability.

Introduction

Digital signal processing system implementers of mobile communication systems face enormous challenges: The first one is the pace of product development, in which different products must be developed in ever shorter intervals. At the same time, system implementation complexity rises. An increasing number of different standards need to be implemented. A third factor is flexibility. Changes on standards must be considered also during the development phase. Finally, efficiency is one of the most crucial parameters. Devices must exhibit both a high performance and small battery consumption.

Algorithmic research and rapid system prototyping of complex signal processing systems require a system development environment supporting a high level of domain-specific abstraction. A specific requirement of those environments is the ability to specify systems at the abstraction level of signal vectors and operators working on them. For these requirements, the Matlab™ programming language is a widely used suitable tool.

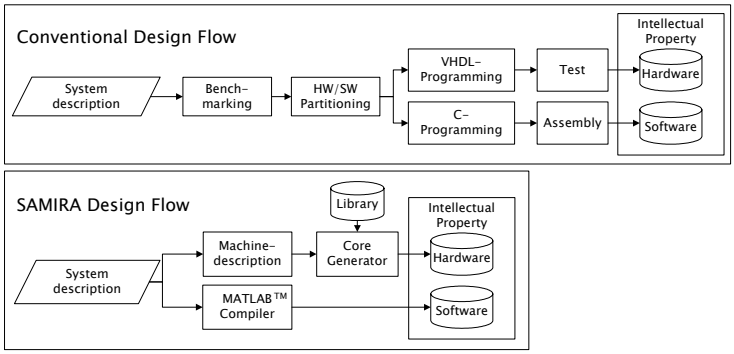

Figure 1: Integrated Design Flow of SAMIRA

The Matlab™ programming language started out as a user-friendly frontend of a set of numerical computation libraries written in the Fortran programming language, i.e. BLAS, LINPACK, LAPACK, etc. Matlab™ is an interpreted language similar to Basic or Perl, lacking explicit type declarations of variables. Hence, the classical design flow system requires two consecutive implementations of digital signal processing, as shown on the upper side of Figure 1: one in Matlab, and second one in a lower level language, like C or VHDL (see Figure 1).

The automation of the engineering task of implementing a high-level system model in a hardware system has two technological prerequisites: An advanced compiler technology suitable for the high-level language, and an efficient hardware architecture.

This paper presents an application of the Synchronous Transfer Architecture (STA) [2] for designing a processor that is both efficient and compiler-friendly. STA was developed by the authors at the Vodafone Chair and shown to be a high-performance, low-power DSP architecture [9, 10]. The objective of the following paper is to explain the design ideas behind the SAMIRA architecture that aims to prove that STA is also a compiler-friendly architecture and thus facilitates rapid system development and flexibility

To this end, the most important developments in Matlab™ compilers and processor architecture is briefly reviewed. Then, the principles of STA and their modifications to support efficient Matlab™ program execution are described. These principles are executed to obtain the SAMIRA architecture. Finally, the current status of its implementation is illustrated.

Matlab™ Compilers

The issue of efficient compilation of Matlab™ has not been raised until recently. Despite the absence of type declarations, two characteristics make the language attractive for compiler parallelization: It does not con-

tain pointers and it features vector-valued arithmetic. The latter allows for explicit expression of parallel operations. Both properties significantly leverage compiler parallelization.

The MATCH Matlab™ compiler [15] is marketed by the company AccelFPGA. It models the programs as functional pipelines whose functional units communicate via remote procedure calls. This representation can be synthesized to an FPGA configuration. The Leiden Architecture Research and Exploration Tool (LAURA) [16] is a compiler that transforms a program into a network of stream-based function objects. Like MATCH, this approach requires FIFO queues for the communication between functional units, thereby incurring a high hardware implementation overhead.

The MaJIC system [17] is a just-in-time (JIT) compiler. With its dynamic compilation approach the compiler has potential for more aggressive optimizations. If the assumptions made during the compile time turn out not to apply in a specific case, the compiler may interrupt the program at runtime and adapt it to the new situation. The overhead associated with the approach is not feasible in low-power signal processing systems.

These approaches have in common that they either rely on general-purpose processors, or they result in FPGA implementations. Both of these solutions suffer from a low performance and high power consumption.

Processor Architecture

The proposed STA architecture evolved from the consistent application of the design principles of RISC to the area of digital signal processing. This section explains these principles:

The development of RISC architecture was driven by the observation that compilers could hardly utilize the full flexibility offered by increasingly complex instruction sets, and that code size and performance do not directly correspond [13]. RISC architectures simplify instruction processing, and thus allow much higher clock rates – however, at the cost of a large instruction footprint. The resulting RISC architecture was designed to enable pipelined execution of one instruction per clock cycle.

For even higher degrees of parallelism, substantial additional overhead for instruction processing had to be added to obtain the super-scalar execution model: e.g. larger caches, register renaming, branch prediction, dynamic instruction scheduling, register files with many ports, etc.

In contrast, classical digital signal processors (DSP) do not require the additional flexibility for their number-crunching tasks. Saving die area and power is more important than compiler-friendliness. Therefore, classical DSPs supply CISC instructions for specific high-throughput tasks. These instructions do not consume high amounts of memory while exploiting the full degree of available parallelism. A typical example is a MAC (multiply-accumulate) instruction that performs a multiplication, an addition, two address calculations with circular buffering and bit-reverse addressing, and two loads at once, coded with 16 bits of instruction memory.

With an increasing degree of instruction level parallelism, wiring dominates computational capacity. In [12], the Transport Triggered Architecture (TTA) is presented that alleviates this problem: It reduces the amount of interconnection resources and the number of register file ports while enabling efficient code generation by compilers. Yet, these extensions are not sufficient to satisfy the demand of ever more performance-hungry DSP applications.

Designing for ASIC-like Programmable Devices

Traditionally, signal processor designers chose to implement their systems either on DSP, on ASIC or on FPGA. Digital signal processors have a high flexibility, and software development for processors is well understood. On the other side, traditional DSPs have a limited degree of parallel processing capabilities. ASICs offer high performance and low power consumption, however, they are tedious to implement and the resulting systems are not flexible. FPGA systems combine flexibility with a high degree of parallelism, but they suffer from poor performance and high power consumption.

In order to compare different architectures, the authors establish a metric for the degree of efficiency. This metric compares the hardware efforts of actual computation (payload) to the total effort (i.e. computation and overhead for administrative tasks). It can be established for area consumption η_{area}, power consumption η_{power}, or other magnitudes which can be separated into payload and overhead. During the next section, this metric will provide a motivation for specific design decisions in the SAMIRA architecture.

$$\eta_{area} = \frac{A_{payload}}{A_{payload} + A_{overhead}}$$

Architectural Templates used for SAMIRA

In order to enable a Matlab™-programmable device with a high degree of parallelism and a high degree of efficiency, the SAMIRA processor is designed using the following architectural templates:

- SIMD (single instruction, multiple data) processing yields a high performance. Sharing a single instruction controller among several processing elements likewise results in a low hardware overhead.
- STA (synchronous transfer architecture) is a novel micro-architecture that allows the compiler to exploit maximum instruction level parallelism without the need for expensive super-

scalar hardware overhead (see [2]). It is explained in the next section.

- TVLIW instruction compression reduces the memory consumption of instruction words, thereby minimizing the overhead for storing and retrieving instructions (see [14]).
- A specialized memory architecture is tailored to the efficient access to vectors and matrices, as they occur in Matlab™ programs. It will be illustrated later in this paper how it enables direct implementation of high-level vector and matrix operations.

Synchronous Transfer Architecture (STA)

The Synchronous Transfer Architecture is designed to minimize the hardware overhead necessary for controlling the operation of different functional units and the communication between them. While many other approaches require the implementation of a register file, a queue, or a FIFO in the path from an output port of a functional unit to the input port of another functional unit, STA uses a single output register and a multiplexer.

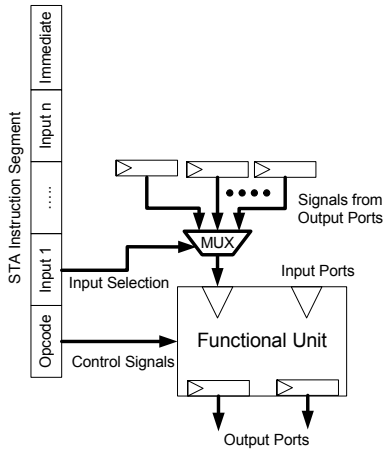

Figure 2: Architectural Template of STA

Figure 2 shows the corresponding architectural template: the design is split up into an arbitrary number of functional units, each with input and output ports. Every input port is connected to a selection of output ports. Alternatively, they may originate from immediate fields in the instruction word. For each computational resource, a segment of the STA instruction word contains the control signals (Opcode) for the functional unit. For each input port, a plain multiplexer selects the appropriate source.

As shown, the communication between these computational resources takes place synchronously. Consequently, neither temporary storage nor synchronization logic is required. As shown in the left part of Figure 2, a very long instruction word facilitates a high degree of instruction level parallelism. Since storing and fetching these instruction words directly from memory would cause a high overhead, the instructions are represented in compressed form. For example, TVLIW [14] represents an efficient 2-dimensional run-length encoding scheme.

Memory Architecture

SAMIRA's memory subsystem is tailored to the efficient support of vectors and matrices, allowing for conflict free strided access, matrix-vector, and matrix transposition operations. The mathematical foundation of the approach is described in [18].

The efficient access to a matrix in transposed form requires a stride permutation. Figure 3a shows the permutation required for transposing a 4x4 matrix as an example. For the sake of simplicity, this example used only 4 parallel processing elements (PEs).

In order to facilitate parallel memory access, the matrix elements are stored in memory in a permuted fashion so that conflict-free access to all elements is possible in either sequential or transposed fashion. Figure 3b shows the calculation performed to obtain a row address *r* and an element index *m* from a sequential address *a*. Finally, Figure 3c illustrates the fact that the processor can access all elements of both the rows and columns of the matrix simultaneously with different PEs.

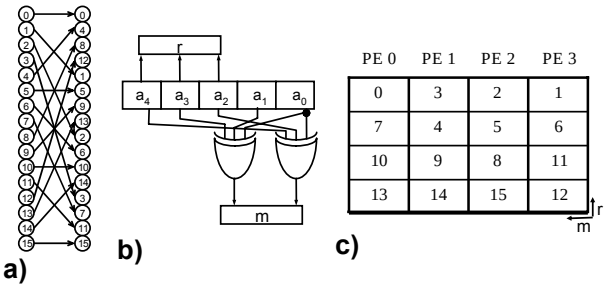

Figure 3: Conflict-Free Memory Access

Compiler Technology

Due to its regular structure, well-known compiler techniques for RISC, VLIW, and TTA architectures can be applied to STA processors, as shown in Figure 3.

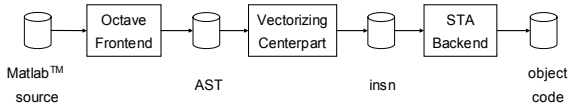

Figure 4: Compiler Overview

A frontend for the Matlab programming language based on the Octave interpreter is presented in [1]. This frontend reads Matlab source code and generates an abstract syntax tree.

This abstract syntax tree is processes by a compiler phase called "centerpart". It performs type analysis and recognizes data parallelism. See [11] for an explanation of the classical Allen/Kennedy approach for vectorizing programs. [4, 5] develop a novel approach for extracting parallelism based on tensor-algebraic program transformations. The centerpart outputs an in-

termediate representation called sequential instruction list (*insn*) of the program that contains data-level parallelism, but not yet instruction level parallelism.

The compiler backend takes the insn representation and performs instruction-selection, scheduling, register-allocation, and instruction compression to produce object code. [3] presents the technology of the STA compiler backend.

The compiler is retargetable to arbitrary STA processors, as long as the following requirement is met: All compiler-visible hardware resources of the processor may serve exactly one of the following purposes:

- **behavior**, i.e. a computational resource that performs some function based on its control signals and its input values. Functional units are not allowed to contain any state excepting pipeline registers.
- **state**, i.e. a resource that stores data. It is not allowed to perform any computation and uses an address input to designate a specific location. If the address originates from an immediate field, it is used as a register file. Otherwise, it is used as a memory by the compiler.

The STA interconnection multiplexers act like the bypass connections found in classical processors.

SAMIRA Architecture

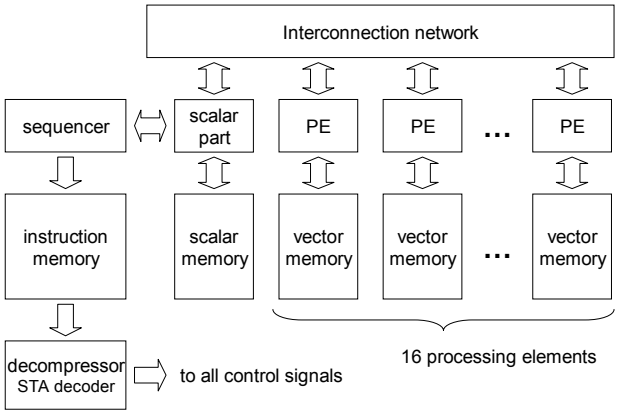

Figure 5: SAMIRA Architecture

Figure 5 shows the architecture of the SAMIRA processor. An instruction sequencer interacts with the scalar part of the processor. It provides an address to the instruction memory, which reads an instruction and passes it to the instruction decoder. The instruction decoder then decompresses the instruction and generates control signals for the synchronous operation of the processor. The instruction memory has 64KB in this design.

The scalar part processes 16 bit and 48 bit integers. It performs loop control and address calculation tasks and is connected to a memory of 4KB.

The SIMD part consists of 16 independent processing elements, each operating on 16 bit integers and containing a local memory of 4KB. The PEs communicate with each other via an additional interconnection network performing permutation operations on SIMD-vectors to arrange the data correctly after matrix memory accesses. It appears as an independent functional unit to the compiler, while in the hardware implementation, it is distributed across the core, filling the space between the PEs.

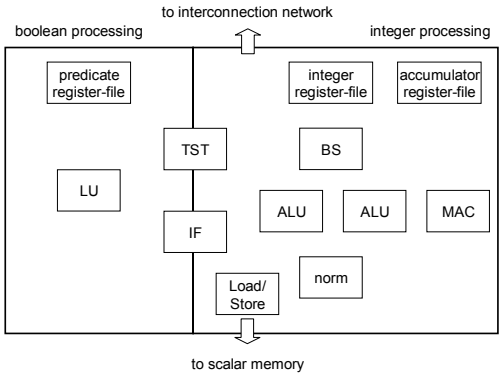

Figure 6: Scalar Part of SAMIRA

Figure 6 depicts an overview of the scalar part of the SAMIRA processor. The left side shows the integer processing part, which may also be used to perform floating point computations.

It contains two integer register files (one or 16-bit, another one for very few 48-bit values), two integer ALUs (one 16 bit, one 48 bits wide), a multiply-and-accumulate (MAC) unit, and a load/store unit. For floating point operations, the barrel shifter (BS) is able to perform an exponent adjustment operation required to add two floating point numbers. A normalization unit performs normalization after floating-point addition. All functional units support the force-one rounding mode for floating-point operations.

The right side of Figure 6 shows the Boolean processing part of the processor that processes single bit quantities. It contains a register file and a logical unit (LU) for handling the carry flags of the ALU, e.g. A testing unit (TST) can determine a sign and zero for an integer value. The multiplexer-unit (IF) can select one of two integer input values according to a Boolean selection input.

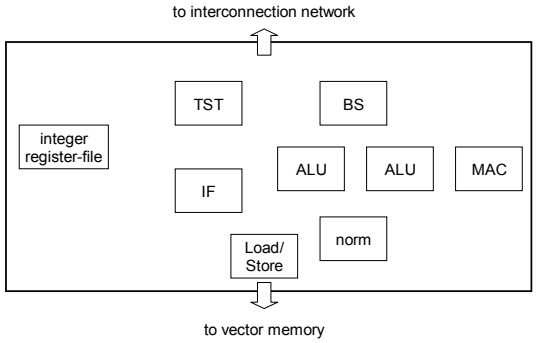

Figure 7: Processing Element of SAMIRA

Next, Figure 7 shows the structure of a processing element of the processor. It is equal to that of the scalar part, except the absence of a Boolean processing part. This task is performed by the scalar part which uses its integer units to process vectors of Boolean values as they occur in the vector part.

Floating Point Mapping

Differing from other approaches, the SAMIRA processor is designed with a data width of 16 bit, a common fixed-point precision for mobile communication standards. Thus, the processor achieves the lowest power consumption and highest speed when performing fixed-point computations. However, this mode of operation is not supported by unattended compilation of Matlab code. Though the compiler can cast floating-point data types to integer or Q15 fixed-point types, this strategy is not general and works with suitable programs only.

For the translation of generic Matlab™ code, the compiler expands floating point operations into separate mantissa and exponent calculation operations. For example, a floating point multiplication operation is expanded into an mantissa multiplication and an exponent addition, while an addition corresponds to an exponent adjustment operation, a mantissa addition, and a normalization operation.

These additional operations increase the computation's latency and consume additional power. However, for floating point numbers with 16 bit mantissa precision, this additional overhead is negligible when compared to the overhead of building dedicated floating-point units. Additionally, all input and output data has to be converted into the internal number representation with separated mantissa and exponent.

For programs requiring full IEEE double precision arithmetic, a significant performance degradation incurred by the construction of floating point operations from 16-bit integer operations are expected. While for double-precision floating-point addition, the 48 bit wide accumulation datapath can be used, a double-precision multiplication has to be constructed from 9 multiplications of 16-bit width. The implementation of IEEE rounding modes instead of force-one rounding would require additional hardware resources.

Preliminary Implementation Results

The SAMIRA processor is implemented using the UMC 0.13 µm, 8-layer-metal CMOS process available from Europractice. Synthesis is performed with Synopsys Design Compiler, place&route is performed with Cadence Encounter.

Figure 8 illustrates the current status of silicon layout. The processor is in development to date, with tape-out being expected within the next months. Most of the processor's features have been implemented and currently subject to verification.

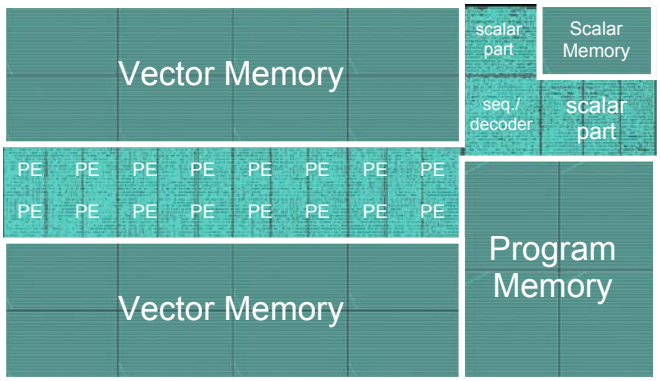

Figure 8: Layout picture of SAMIRA processor

The design is synthesized and routed to run at 300 MHz with unpipelined multipliers, to enable a power-efficient operation. With pipelined multipliers, the processor is expected to reach a clock speed of 750-900 MHz. Table 1 summarizes the theoretical peak performance. The actual performance is likely to be limited by memory bandwidth and operation latency.

Low Power Variant (300 MHz)	
integer performance	200,000 MIPS
16-bit floating-point	10 GFLOPS
48-bit floating-point	3 GFLOPS
High-Performance Variant (900 MHz)	
integer performance	600,000 MIPS
16-bit floating-point	30 GFLOPS
48-bit floating-point	9 GFLOPS

Table 1: Theoretical Peak Performance

The total area consumption of the SAMIRA processor is 7 mm^2; of which 6 mm^2 are consumed by memory. Power simulations that were performed with previous STA processors [10] indicate that a total power consumption of about 360 mW can be expected for SAMIRA running at 300 MHz. The major part of it is consumed by the memories.

Design	**Efficiency η_{area}**
DVB-T receiver	35.78%
OFDM-DSP	38.29%
802.11/UMTS combo	42.48%
SAMIRA	**41.36%**

Table 2: Degree of Efficiency η_{area}

In order to estimate the efficiency of SAMIRA, its die area efficiency according to the previously established efficiency metric η_{area} is calculated: See Table 2 for the results. The presented figure is based on the core logic area and does not take into account the area occupied by memory. When compared with application-tailored DSP designs [8-10] which were developed using the STA approach, the SAMIRA general purpose design shows a competitive degree of efficiency while still being generic and programmable with a high-level programming language.

Conclusions & Future Work

This paper presented an efficient architecture for use as a coprocessor dedicated to execute signal processing programs written in the Matlab programming language.

With more than 40% of the core area being dedicated to performing actual signal processing computations, the SAMIRA processor provides a high computing performance with little overhead required for programmability.

This design results in an architecture that provides a high performance signal processing system with low power consumption and a small die size. The processor is supported by a high-level language compiler for rapid development of digital signal processing applications.

Detailed performance and power results of the SAMIRA processor will follow once the silicon run is finished.

Acknowledgements

This work has been sponsored in part by the German Science Foundation (Deutsche Forschungsgemeinschaft, DFG) within SFB358-A6, and the DFG-Priority Program VIVA.

Bibliography

[1] G. Cichon and G. Fettweis. *MOUSE: A shortcut from MatlabTM source to SIMD DSP assembly code.* In Proc. of Third International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'03), pages 126–130, Samos, Greece, July 2003.

[2] G. Cichon, P. Robelly, H. Seidel, E. Matúš, M. Bronzel, and G. Fettweis. *Synchronous transfer architecture (STA).* In Proc. of Fourth International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS' 04), pages 126–130, Samos, Greece, July 2004.

[3] G. Cichon, P. Robelly, H. Seidel, M. Bronzel, and G. Fettweis. *Compiler scheduling for STA-processors.* In Proc. of Fourth International Conference on Parallel Computing in Electrical Engineering (PARELEC'04), Dresden, Germany, September 2004.

[4] P. Robelly, G. Cichon, H. Seidel, and G. Fettweis. *Implementation of recursive digital filters into vector SIMD DSP architectures.* In Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP'04), Montreal, Canada, May 2004.

[5] P. Robelly, G. Cichon, H. Seidel, and G. Fettweis. *A HW/SW design methodology for embedded SIMD vector signal processors.* Int. Journal of Embedded Systems, 1(11), January 2005.

[6] P. Robelly, G. Cichon, H. Seidel, and G. Fettweis. *Automatic code generation for SIMD DSP architectures: An algebraic approach.* In Proc. of Fourth International Conference on Parallel Computing in Electrical Engineering (PARELEC'04), Dresden, Germany, September 2004.

[7] H. Seidel, G. Cichon, P. Robelly, E. Matúš, and G. Fettweis. *Generated DSP cores for implementation of an OFDM communication system.* In Proc. of Fourth International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'04), Samos, Greece, July 2004.

[8] H. Seidel, G. Cichon, P. Robelly, E. Matúš, and G. Fettweis. *SFB358-A6 demonstrator: HW/SW co-design of a DVB-T receiver.* In Proc. of Fourth International Conference on Parallel Computing in Electrical Engineering (PARELEC'04), Dresden, Germany, September 2004.

[9] E. Matúš, O. Prätor, A. Zoch, G. Cichon, and G. Fettweis. *Software reconfigurable baseband ASSP for dual mode UMTS/WLAN 802.11b receiver.* In IST Mobile and Wireless Communications Summit, Lyon, France, June 2004.

[10] M. Hosemann, G. Cichon, P. Robelly, H. Seidel, T. Dräger, T. Richter, M. Bronzel, and G. Fettweis. *Implementing a receiver for terrestrial digital video broadcasting in software on an application-specific DSP.* In Proc. IEEE Workshop on Signal Processing Systems 2004 (SIPS'04), 2004.

[11] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, Burlington, MA, 2001.

[12] H. Corporaal. *Microprocessor Architecture from VLIW to TTA*. John Wiley & Sons, 1997.

[13] J. Hennessy and D. Patterson. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2nd edition, 1996.

[14] M. Weiß and G. Fettweis. *Dynamic codewidth reduction for VLIW instruction set architectures in digital signal processors.* In 3rd. Int. Workshop in Signal and Image Processing (IWSIP '96), pages 517–520, Jan. 1996.

[15] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee. *A system for synthesizing optimized FPGA hardware from Matlab™.* In Proc. Int. Conf. on Computer Aided Design, San Jose, CA, November 2001.

[16] T. Harriss, R. Walke, B. Kienhuis, and E. Deprettere. *Compilation from Matlab™ to process networks realized in FPGA. Design Automation of Embedded Systems*, 7(4), 2002.

[17] G. Almasi and D. Padua. MaJIC: *Compiling MATLAB for speed and responsiveness.* In Proc. ACM SIGPLAN PLDI, Berlin, 2002.

[18] J. Takala and T. Järvinen. Register-Based Permutation Networks for Stride Permutation. In Proc. of Third International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'03), pages 126-130, Samos, Greece, July 2003.