Microelectronics Journal 46 (2015) 237–247

Contents lists available at ScienceDirect

Microelectronics Journal

journal homepage: www.elsevier.com/locate/mejo

A proposed synthesis method for Application-Specific Instruction Set Processors

Péter Horváth*, Gábor Hosszú, Ferenc Kovács

Department of Electron Devices, Budapest University of Technology and Economics, Magyar tudósok körútja 2, H-1117 Budapest, Hungary

A R T I C L E I N F O

Article history:
Received 13 May 2014
Received in revised form
13 December 2014
Accepted 2 January 2015

Keywords:
Application-Specific Instruction Set Processor
ASIP
Architecture description language
ADL
System on Chip
SoC
RTL processor design

A B S T R A C T

Due to the rapid technology advancement in integrated circuit era, the need for the high computation performance together with increasing complexity and manufacturing costs has raised the demand for high-performance configurable designs; therefore, the Application-Specific Instruction Set Processors (ASIPs) are widely used in SoC design. The automated generation of software tools for ASIPs is a commonly used technique, but the automated hardware model generation is less frequently applied in terms of final RTL implementations. Contrary to this, the final register-transfer level models are usually created, at least partly, manually. This paper presents a novel approach for automated hardware model generation for ASIPs. The new solution is based on a novel abstract ASIP model and a modeling language (Algorithmic Microarchitecture Description Language, AMDL) optimized for this architecture model. The proposed AMDL-based pre-synthesis method is based on a set of pre-defined VHDL implementation schemes, which ensure the qualities of the automatically generated register-transfer level models in terms of resource requirement and operation frequency. The design framework implementing the algorithms required by the synthesis method is also presented.

© 2015 Elsevier Ltd. All rights reserved.

1. Introduction

Today's increasingly developing manufacturing technology makes it possible to build complete data processing systems on a single chip including digital and analog building blocks with on-chip memories and communication. These complex systems called System on Chips (SoCs) comprise various computational submodules called Application-Specific Integrated Circuits (ASICs) with a well-defined functionality, which cannot be modified after manufacturing. ASICs provide favorable energy-efficiency and high computation performance since they are optimized for a specific function. However, besides the high computation capacity, the reusability and flexibility as design constraints have gained significant importance because of the extremely high manufacturing costs.

There are two major ways to ensure flexibility in SoCs. The first approach is based on embedded reconfigurable logic devices, such as Field Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs), which make it possible to implement arbitrary digital functionality and can be reconfigured "on the field" after manufacturing. The other solution is the usage of general purpose devices operating according to a stored program. These devices are the well-known microprocessors whose functionality can be changed with a simple software update.

Both solutions provide flexibility on the cost of limited computation capacity. In case of general purpose microprocessors, the additional memory accesses and other administrative operations necessitated by the programmable nature cause a significant penalty in computation performance. In case of programmable logic devices the reconfigurability is achieved by generic logic cells and a high amount of programmable interconnection and wiring resources, which cause high path delays that result in a relatively low operation frequency and high power consumption.

The concept of Application-Specific Instruction Set Processors (ASIPs) is a promising result of the trade-off exploration between flexibility and computation performance [1]. ASIPs are microprocessors with a unique instruction set optimized for a specific application domain. Since they execute a program, they can quickly adapt to the varying functional requirements. At the same time they have instructions and hardware resources optimized for the target application; therefore, they provide a higher computational performance than the general purpose microprocessors [2,3].

Due to the rigorous time-to-market requirements, reducing the time-consumption of ASIP design is essential. The primary design tools of ASIPs are the Architecture Description Languages (ADLs), which are specific modeling tools for instruction sets and microarchitecture. The design frameworks based on these formal language

* Corresponding author. Tel.: +36 1 463 3072; fax: +36 1 463 2973.
E-mail address: horvathp@eet.bme.hu (P. Horváth).

http://dx.doi.org/10.1016/j.mejo.2015.01.001
0026-2692/© 2015 Elsevier Ltd. All rights reserved.

models mainly concentrate on the software parts of the microprocessor systems, namely the instruction set simulator, assembler, compiler, and debugger generation. The automated hardware synthesis plays a secondary role since the complex datapaths including internal data storage subsystems and interdependent pipeline stages characteristic to the instruction set processors require a high level of optimization, which cannot be achieved by contemporary Computer-Aided Design (CAD) tools. Therefore, the ADL-based design frameworks, even if they are able to generate a hardware model, often compromise or neglect the quality of the final register-transfer level (RTL) implementation [4].

This paper presents a novel approach for ASIP modeling with great emphasis on automated hardware generation. The main idea of our solution is similar to High-Level Synthesis (HLS) [5–7], where the design is described as an algorithm, then a High-Level Synthesis procedure generates an RTL model. With lowering the abstraction of the formal specification while keeping the readable algorithmic design style, our approach makes it possible to achieve a high level of optimization and a reduced development time.

The basis of the presented approach is a new abstract model of ASIP architectures and a mixed algorithm and RTL description language called Algorithmic Microarchitecture Description Language (AMDL) optimized for the proposed abstract architecture model. The algorithmic language environment of AMDL combined with the detailed description style characteristic to RTL ensures the rapid architecture implementation and the comprehensive control over the microarchitectural details as well. The initial formal language model comprises a lot of structural information, which makes it possible to generate an optimized, technology-independent RTL output, which can be transformed into a gate level model using the existing logic synthesis tools.

This paper is organized as follows: Section 2 provides a brief overview of SoC implementation solutions, ASIP modeling methodologies, and their drawbacks in terms of hardware generation. Section 3 presents a proposed novel architecture model of ASIPs and Section 4 describes a proposed synthesis approach and modeling language for this architecture model. Section 5 gives an overview of the design framework implementing the proposed synthesis method. Section 6 presents experimental results and Section 7 draws conclusions.

2. Background

2.1. System on Chip implementations

Numerous heterogeneous architectures can be created with the combination of application-specific functional units, general purpose microprocessors, and FPGA resources [8–11]. The different solutions enable a trade-off between flexibility and computation performance.

A general purpose microprocessor combined with an application-specific functional unit as an accelerator is used when certain computational tasks need a significant speed up. The accelerator may be loosely or tightly coupled to the microprocessor depending on the application. Both solutions define an interface between the two major components of the system, which may result in a bottleneck in terms of computation performance. Furthermore, if the accelerator is implemented in ASIC, both its functionality and its interface to the microprocessor are fixed.

A more flexible solution can be achieved with the application of FPGA fabric for implementing the accelerator functionality. In this case both main components provide post-fabrication flexibility but the fixed interface between the microprocessor and the programmable logic may prohibit comprehensive optimizations.

FPGA vendors usually provide another solution for combining instruction set processors with reconfigurable hardware. The soft-core processors implemented by FPGA resources provide a limited configurability in terms of instruction set, internal memories and pipeline implementation. In this case, additional accelerators can also be placed beside the microprocessor using the reconfigurable FPGA fabric. The special purpose high performance resources of the FPGAs, such as block memories, DSP slices, and high speed communication interfaces can be used either by the microprocessor or the accelerator. This solution is favorable in terms of flexibility but the interface issue mentioned above still exists and the reconfigurable nature of the hardware results in a limited operation frequency and poor energy efficiency.

2.2. Application-Specific Instruction Set Processors

ASIPs represent special types of stored-program microprocessors, whose instruction set is optimized for a certain application or application domain. This approach is similar to the processor–accelerator system but the two main parts are not separated. There is no well-defined interface between the application-specific functionality and the instruction set processor; therefore the drawback caused by their interface is completely eliminated.

There are two main approaches in ASIP design methodologies. Both of them are based on a low-level profiling of the target application. In the first case called instruction set customization, the profiling data are used to determine a subset of a general instruction-set, which the application does not use. By neglecting the unused instructions in the synthesis step the resource requirement and hence the cost and the area can be decreased. In the other approach called microarchitecture customization also known as Instruction Set Extension (ISE) the profiling data is used to determine complex functionalities the application frequently uses. Then this functionality is synthesized as a special instruction (or as a set of special instructions) implemented in highly optimized functional units called Application-Specific Custom Unit (ASCU) integrated into the processor's datapath. This solution significantly improves the computation performance.

The ASIPs incorporate the flexibility of programmable solutions and the high computational performance of ASICs. Due to the significant demand for flexibility, the rapidly developing wireless communication is one of the most important application fields of ASIPs. [12] and [13] present specific digital signal processing (DSP) architectures for decoding and demapper implementations, which can easily adapt to varying network and communication standards. [14–18] utilize the favorable computation performance of ASIPs, which can be used for efficient implementation of signal processing algorithms in multimedia applications, such as Fast Fourier Transform (FFT), Discrete Cosine Transform (DCT), Retinex-filtering, QR Decomposition (QRD), Singular Value Decomposition (SVD) and Motion Estimation (ME). [19] presents another application field of ASIPs, namely encryption standards, which also demand a high computation performance. [20] presents a high-throughput ASIP with specialized Single Instruction Multiple Data (SIMD) instructions frequently used in biological sequence alignment algorithms. All the above mentioned works describe typical ASIP architectures in a sense that they include application-specific pipelines, which operate according to a stored program.

2.3. Algorithmic modeling of ASIPs

In SoC design industry, a widely used method for speeding up a design process is the high abstraction level design entry combined with automated design steps. In electronic system design the so-called High-Level Synthesis (HLS) [21–23] is a typical implementation of this concept. An HLS algorithm is used to transform an

algorithm level specification into a synthesizable RTL model. Although the HLS method reduces the time required to develop high quality hardware, it cannot be used in case of every type of data processing systems. There are two major problems regarding HLS [5–7]: (i) the syntactic variance, which means that the synthesis result significantly depends on the coding style, and (ii) the lack of interactivity; the high-level programming language model used as design entry of the HLS, does not include detailed information about the register-transfer level functional elements and their interconnections, because the exact microarchitecture takes shape only during the automated synthesis steps and it is difficult for designers to control this process. Presently, the HLS is widely used for designing streaming DSP applications. However, in the design flow of ASIPs, the HLS is not frequently applied because in that case the foresaid problems occur more significantly. Its reason is that the computation model of ASIPs does not fit well with those in streaming DSP algorithms, which HLS is optimized for. Namely, a streaming DSP system usually consists of interconnected submodules (pipeline stages) communicating via first-in first-out (FIFO) channels. The behavior of the ASIPs is different. They provide a wide scale of functionality, they often contain complicated internal data-storage subsystems, and their pipeline stages are more interdependent than those in the digital signal processing systems. This functional diversity increases the effect of syntactic variance; therefore the optimization algorithms of the highly automated CAD tools may not be efficient enough; the time-consuming and error-prone manual optimization is unavoidable.

Due to the foresaid problems the ASIPs have their own design methodology based on a specific subtype of modeling languages called Architecture Description Languages (ADLs), which provide language constructs to describe the behavior and also the micro-architectural details of instruction set processors. The design frameworks based on ADLs place great emphasis on instruction set simulation and other software components (compiler, assembler, and debugger generation) [24–26] rather than the automated hardware generation [27,28]. In case of ADL-based design frameworks whose aim is to generate a hardware model from the ADL specification, the designer often has to deal with significant restrictions in terms of microarchitecture [29–31] (e.g. single-issue pipeline, VLIW, implicit instruction pointer, and interrupt handling mechanism), and they often compromise or neglect the quality of the final ASIP implementation. The final RTL models are usually created completely manually or with a significant amount of manual modifications applied on the automatically generated RTL models. This latter solution necessitates additional knowledge in RTL microprocessor development. Furthermore, ADLs are not able to model application-specific data processors with dedicated functionality [4,30].

2.4. Earlier work related to AMDL-based modeling

An earlier stage of our work has been reported in [32], where an overview of the concepts of behavioral RTL and structural RTL coding styles have been discussed much more shortly than in this paper in Section 4.2. The language AMDL has already been introduced; however, this language has been improved significantly ever since; the earlier version of the language only supported a single target architecture model which was a less detailed, initial version of the target architecture model called "dedicated multicycle machine" described in Section 3. The up-to-date version of the AMDL is discussed in Section 4.1 including the statement block syntax and semantic elements supporting the other three target architecture models. The proposed manual design method presented in [32] has also been improved. A complete design framework including the software tools for synthesis and the assembler-generation has been prepared, which is presented in Section 5 in this paper.

3. A proposed architecture model of ASIPs

In this section we present a novel abstract model of ASIP architectures, whose main objective is to provide a higher structural flexibility than the target architecture models applied in contemporary ASIP design environments and to underlie a proposed synthesis method presented in the following sections. The machine model describes the structure of the data processing systems as a hierarchy of elementary architecture-elements distinguished by their programmability and control mechanism. The Unified Modeling Language (UML) class diagram in Fig. 1 shows the architecture-elements and their relations and Table 1 shows their main properties. The capability of instantiation is denoted in the UML diagram as an *aggregation* association.

As described in Section 2.2, ASIPs usually comprise of application-specific pipelined datapaths whose control unit operates according to a stored program. In order to keep our data processor architecture model as broad as possible, besides the programmable pipelines, the concept of dedicated pipelines and multicycle machines has also been introduced. Dedicated means that the machine does not require a program for controlling the operation (it does not include e.g. an instruction pointer and an instruction register typical of instruction set processors and its main functionality cannot be changed after the synthesis). The multicycle machine can be applied when the application does not need high computation performance. In case of multicycle machines the latency (the number of clock cycles the input data or an instruction spends in the datapath) is equal to their throughput (the number of clock cycles the machine needs to produce new output data or the result of an instruction), the operations are not overlapped. The benefit of multicycle machines is that they usually require less resource than pipelines to perform the same operation, because there is no need to deal with pipeline hazards and no pipeline registers are required.

4. ASIP synthesis based on a unified RTL representation

The ADL-based synthesis methods usually use Hardware Description Languages (HDLs) to describe an intermediate representation of the designed system. Hardware description languages, such as VHDL and Verilog have been developed to specify the

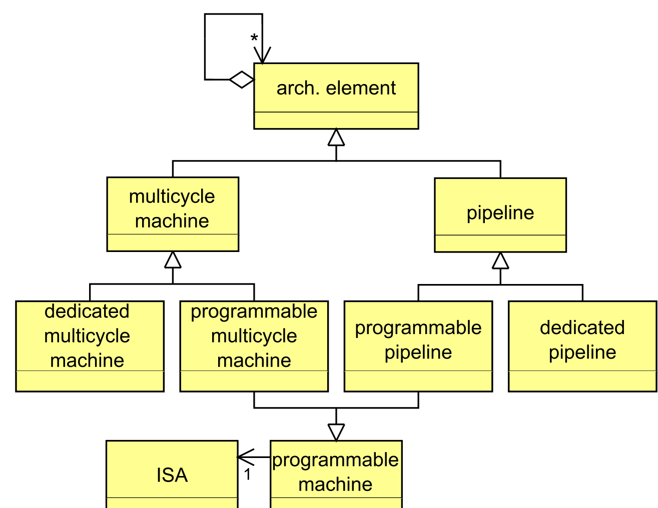

Fig. 1. The architecture-element types constituting the proposed ASIP architecture model. ISA stands for Instruction Set Architecture.

Table 1
Architecture-element properties.

Architecture-element	Properties
Programmable multicycle machine	It implements a unique instruction set. Multiple clock cycles are required to execute a single instruction.
Dedicated multicycle machine	It does not have an instruction set; it is able to perform a dedicated data-processing algorithm. Multiple clock cycles are required to perform the dedicated function.
Programmable pipeline	It implements a unique instruction set. The control mechanism makes possible to overlap the execution of the consecutive instructions.
Dedicated pipeline	It does not have an instruction set; it is able to perform a dedicated data-processing algorithm. The control mechanism makes possible to overlap the execution of the consecutive data processing steps.

structure and also the behavior of complex digital circuits. They provide algorithmic language constructs, such as instruction sequences, conditional statements and loops, which make it possible to describe an algorithm. Moreover, HDLs are able to describe the structure via component instantiations and port mapping. Due to this diversity of language constructs, the syntactic variance is in case of HDLs also a significant issue. If we examine the different synthesis-related RTL hardware models in detail, it can be observed that they are very different in terms of language constructs and coding style. Based on the language constructs applied in the HDL descriptions, we can observe two sub-categories of RTL; there are behavior-like and structural-like models. The first one represents a slightly higher level of abstraction and uses behavior-like language constructs of HDLs; the other one represents a lower level of abstraction and it contains more information about the structure of the described system. The language constructs characteristic to these sub-categories can be mixed in a complex design. We have introduced two implementation schemes, called behavioral RTL and structural RTL, representing the endpoints in terms of model granularity. The behavioral RTL implementation scheme is a coarse-grained representation; it may describe very complex functionalities in a single design unit, while the structural RTL implementation scheme is fine-grained with a lot of design units and interconnections. After the logic synthesis these two RTL models eventuate in different gate-level representations in terms of resource requirement and timing. The differences are discussed in detail in Section 6.

We propose a synthesis method, which is based on an intermediate abstraction (algorithmic RTL, ARTL) between the high level algorithmic models and the aforementioned HDL-based RTL models (see Fig. 2). This abstraction level is implemented by a novel modeling language called Algorithmic Microarchitecture Description Language (AMDL) presented in the following subsection.

AMDL can be considered a common design entry for the behavioral RTL and the structural RTL hardware models. It is optimized for the abstract data processor architecture model presented in the previous section; therefore, it is able to model a wide scale of functionality, including data processors with dedicated functionalities and instruction set processors as well. The problems of algorithmic processor design detailed in Section 2 have been eliminated by reducing the algorithmic language constructs to instruction sequences, infinite loops, and conditional statements. In fact, the language only provides an algorithmic environment, which the RTL design can be performed in. The high level synthesis tasks, such as allocation, scheduling, and binding have to be performed manually in this algorithmic language environment; therefore a high optimization level can be achieved. That means that the concept of ARTL represents an intermediate stage between the traditional RTL and the algorithmic level in terms of design effort and elaboration. The output of the ARTL synthesis method is a pre-synthesized behavioral RTL and structural RTL model of the desired data processing system, which can be efficiently transformed into a high-quality gate level model with the existing logic synthesis tools.

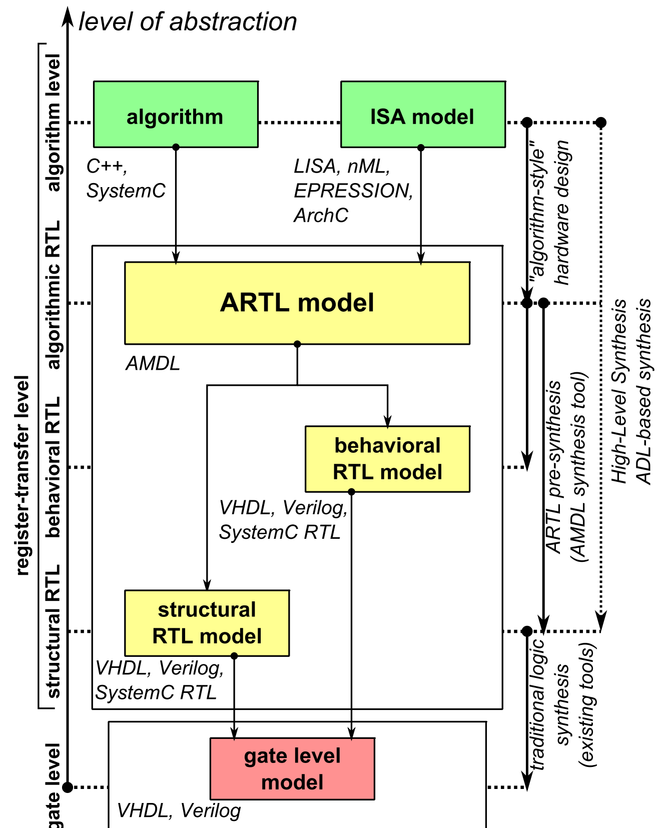

Fig. 2. Scope and objective of AMDL-based hardware design.

4.1. Algorithmic Microarchitecture Description Language (AMDL)

In order to achieve an efficient ASIP design methodology the algorithmic coding style of high level synthesis and the microarchitecture-related details of ADLs can be combined with each other. In this section we present a novel proposed modeling language called Algorithmic Microarchitecture Description Language (AMDL) which is able to implement this combined description style [32].

There are two different ways to embed RT level structure information in the machine description. The traditional approach is to describe the behavior and the structure in two different sub-models and the other way is to embed the structural information directly into the algorithmic behavioral part. Our description language follows this approach. Fig. 3 shows an AMDL model example of an iterative, run-time configurable FIR filter. Configurability means that the coefficient memory is external and the order of the filter can be altered up to 255 without resynthesizing the design.

Based on the design example shown in Fig. 3 the differences between a traditional HLS design entry and the AMDL can also be recognized. The most important characteristics of AMDL are that

```
machine fir is
  -- resource declarations --
  signal   ena:                  ctrlin  [1];
  dataport px:                   input   [32];
  dataport py:                   output  [32];
  dataport addr_2_coeff_mem:     output  [8];
  dataport order:                input   [8];
  dataport data_from_coeff_mem:  input   [32];
  storage  cmar:                 reg     [8];
  storage  acc:                  reg     [32];
  storage  y:                    reg     [32];
  storage  tmp_mem:              regfile [1][1][8][32];
  operator mul:                  async   (multiplicand[32],
                                          multiplier[32])
                                         (product[32]);
  operator add32:                async   (in1[32],in2[32])(result[32]);
  operator add8:                 async   (in1[8],in2[8])(result[8]);
begin -- functional statements --
  structure output_port_assignments:
    addr_2_coeff_mem <= cmar;
    py <= y;
  end structure;
  controlpoint idle: cmar <= 8U"0";
                     loop
                       if ( ena = "1" ) then break; end if;
                     end loop;
  loop
    if ( ena = "0" ) then redirect to idle;
    else
      acc <= mul.product(data_from_coeff_mem,px);
      if ( cmar = 8U"0" ) then
        y <= add32.result(tmp_mem.A[cmar],acc);
      elsif ( cmar = order ) then
        tmp_mem.W[add8.result(cmar,8S"-1")] <= acc;
        cmar <= 8S"-1";
      else
        tmp_mem.W[add8.result(cmar,8S"-1")] <=
        add32.result(tmp_mem.A[cmar],acc);
      end if;
    end if;
    cmar <= add8.result(cmar,8U"1");
  end loop;
end machine; -- fir --
```

Fig. 3. The AMDL model of an iterative FIR filter.

the assignments constituting the algorithm consist of left-value and right-value expressions which refer to a specific RT level data storage element or functional unit declared in the resource declaration part of the description. The resource declaration does not mean component declaration but instance declaration. Therefore, the designer performs the resource allocation and the binding manually in an algorithmic language environment while HLS algorithms handle this automatically without providing a detailed interface for the designer to control the process. Since there are no exact FSM states in AMDL, the scheduling is performed by the statement blocks and their semantics. For example if no statement blocks are used, every assignment represents an independent state in the FSM realizing the behavior. If the concurrent statement block is used, the assignments included in the statement block relate to the same FSM state, which means that the resources asserted in the assignments inside the concurrent block are scheduled into the same clock cycle.

At the same time AMDL provides language constructs of the structured programming: instruction sequences, loops, and conditional statements. These language constructs and the other AMDL control statements described later in this paper give a comprehensive means of scheduling, which makes it possible to influence the control states of the prospective circuit in a more detailed way than in case of HLS.

4.1.1. Resource and signal types

Regarding its syntax, AMDL is a domain-specific formal language optimized for data processors. It operates with resources and signals which appear in every type of data processing systems. There are three resource types and seven subtypes available in AMDL (see Table 2). The input and output ports make it possible to connect the system to the external world, the registers and register files are able to implement internal variables and the operators realize the data manipulations.

The signal types of AMDL represent the control inputs, control outputs, internal control signals, and status signals of data processing systems (see Table 3). There is no need to declare the control signals and the status signals; their use is an implicit declaration, but control inputs and outputs have to be declared explicitly.

Table 2
Resource types and subtypes in AMDL.

Resource type	Resource subtype	Description
Dataport	Data input (*input*)	It is a connection point to the external world. It does not store any data.
	Data output (*output*)	
Storage	Register (*reg*)	It is a bitvector with an arbitrary length. It can be used as internal variable.
	Register file (*regfile*)	It is an array of registers. The number of read and write interfaces, the capacity and the word length are arbitrary.
Operator	Asynchronous operator (*async*)	Asynchronous functional unit.
	Synchronous operator (*sync*)	Synchronous functional unit. It provides the result in several clock periods.
	Multicycle operator (*multicycle*)	Synchronous functional unit. It provides the result in several clock periods. This operator type makes possible to implement hierarchical designs.

Table 3
Signal types in AMDL.

Signal type	Description
Control input (*ctrlin*)	It is an external control input that influences the behavior of the control unit. It can occur in the AMDL description in the condition field of the conditional statements (e.g. interrupt lines).
Control output (*ctrlout*)	It is a control signal sent by the control unit to the external world (e.g. memory strobe signals).
Control signal	It is a control signal sent by the control unit to the datapath (e.g. clock enable signals of the registers).
Status signal	It is a signal provided by a functional unit of the datapath. It influences the behavior of the control unit. It can occur in the AMDL description in the condition field of the conditional statements (e.g. flags).

4.1.2. Statement blocks and their semantics

In most formal languages it is possible to form blocks of instructions. Sometimes the only benefit is to make the code more readable but usually the instruction block indicates some semantic coherence as well; namely, the parsing algorithms must handle the statements included in the same block together (e.g.: statements in the same loop, statements in the same VHDL process or Verilog always block). In AMDL, there are five different statement blocks with specific functionality. Table 4 summarizes the semantics of these statement blocks.

4.1.3. Design units, design structure

A design unit is an elementary part of an AMDL model, which can be constructed independently. There are three design unit types in AMDL, machine definition, ISA definition and pipeline definition. Table 5 summarizes the role of the different design unit types. The UML class diagram in Fig. 4 shows the general structure of an AMDL model.

A machine can be instantiated in another machine or pipeline as a multicycle operator. The association between the machine definition and the pipeline definition means that a particular machine can contain multiple pipelines. The coupling between a machine and a pipeline embedded into it is tighter than that between a machine and a multicycle operator. The machine and the associated pipeline are able to read the internal storage resources of each other, while a multicycle operator has a well-defined interface. Therefore the language implements the binding between machine definition and pipeline definition in different ways. There is no need to explicitly declare the associated pipeline in the declaration part of the machine design unit but the heading of the pipeline definition includes the name of the machine it is associated with.

4.2. Implementation schemes for behavioral RTL and structural RTL abstractions

In order to perform an automated algorithmic RTL-based pre-synthesis, exact VHDL implementation schemes have been developed for the behavioral RTL and the structural RTL abstraction levels, including HDL model structure, applied language constructs, clocking schemes, Finite State Machine (FSM) implementations and a set of pre-defined HDL models of storage resources. The exact definition of the combination of language constructs used in an RTL model is important in AMDL-based system design; these two model types result in very different synthesis results, mainly in terms of resource requirement and clock frequency, ensuring the comprehensive design space exploration (see Section 6).

Structural RTL. There are two different implementation schemes for structural RTL. The distributed structural RTL representation consists of two main HDL design entities; a datapath, which contains the data-storing and manipulating resources (registers, register files and operators) declared in the AMDL description as independent submodules, and a control unit implemented by specific FSMs. The fused structural RTL scheme describes the system in a single design unit, but the different AMDL resources are represented by separated VHDL blocks. In case of ASIP modeling, the controller FSM often includes an instruction pointer (also known as program counter) and an instruction register. In contrast, the structural RT level VHDL implementations of the controller FSMs do not include any internal registers except the state register and a 1-bit register on the outputs respectively (it is required in order to reduce clock-to-output delay and prevent glitches on the control lines). The VHDL code contains only assignments with a control signal or control output on the left side and a constant on the right side. Fig. 5 shows the VHDL model structure of the distributed structural RTL implementation of the FIR filter described in Fig. 3.

Behavioral RTL. The structure-oriented RTL implementations described above are very detailed in terms of the interconnection of the storage and data manipulating resources. The behavior of the system is hidden in the FSMs and in the implementation of the operators. The behavioral RTL approach represents a higher abstraction level. In this case, the VHDL models of the different design units contain the functionality of the control unit and the data manipulation resources in a single VHDL process, only the synchronous operators are described in separate processes. Although the behavioral RTL

Table 4
Statement blocks in AMDL.

Statement block	Semantics
Structure	The structure block indicates that the resources and interconnections realizing the assignments inside the block have to be implemented but the control unit does not assign any control states to the assignments.
Concurrent	The statements embedded into a concurrent block are performed parallel.
Stage	The pipeline design unit consists of stage blocks in order to keep the code readable. Similar to the concurrent block, the stage block is a parallel structure.
Bypass	Sequential block inside the pipeline. A unique condition can be assigned to each bypass block. If the condition comes true during the pipelined execution, the pipeline stops and the bypass block is executed.
Observer	The observer block is responsible for assigning a condition to the bypass blocks.

Table 5
Design units in AMDL.

Design unit	Semantics
Machine	The machine definition is the base unit of an AMDL model. It represents the multicycle machine class shown in Fig. 1. Since a particular machine can be able to implement an instruction set, an ISA definition can be assigned to the machine design unit.
Pipeline	The pipeline definition represents the pipeline machine class as shown in Fig. 1. It describes a pipeline, which can be embedded in an instruction set processor (even several times, if needed) or a data processor with dedicated function. Since a particular pipeline can be able to implement an instruction set, an ISA definition can be assigned to the pipeline design unit.
ISA	The Instruction Set Architecture (ISA) definition contains the instruction definitions of a particular machine or pipeline, which it is assigned to. It describes the program memory capacity, word width, operation codes, mnemonics and the bit positions of the certain instruction parameters.

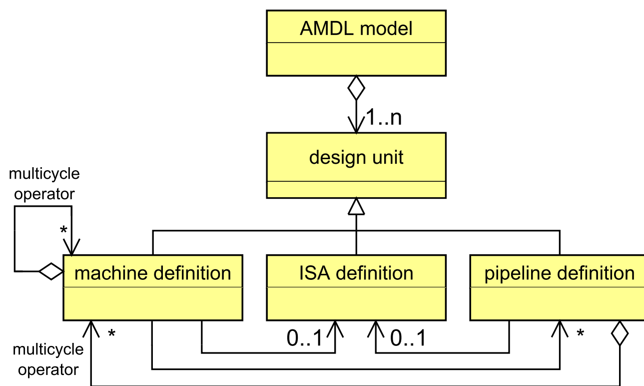

Fig. 4. General structure of an AMDL model.

implementation of an AMDL model describes the whole functionality via complex arithmetic and logic expressions in the signal assignments and subroutine calls, it does not describe the interconnections and sharing of data manipulating resources exactly. Table 6 shows how AMDL resources and language constructs appear in the different VHDL implementation schemes.

5. AMDL design framework

In this section, we present a design flow based on a software framework being under development. Fig. 6 shows the AMDL design flow with the provided software tools, the required textual models and the intermediate system representations.

Since AMDL is synthesis-oriented, the software part of the design flow is less sophisticated, only a practical configurable assembler is provided to speed up the creation of the initial test programs of ASIPs. Based on the ISA definitions described with AMDL the generic assembler tool can be used to create simple assembly test programs.

The entries of the design flow's hardware part are an ISA specification (or specifications in case of asymmetric multi-core systems) and the AMDL model of the top-level system, the different pipelines, and the multicycle operator resources (dedicated functional units or internal ASIPs).

The Parser is responsible for checking the syntax and generating an internal representation of the described system. The synthesizer generates an RT level HDL model (behavioral, distributed structural or fused structural). This model does not include the exact behavior of the asynchronous/synchronous operator resources instantiated in the AMDL description. In practice the async/sync resources realize simple arithmetic, logic, and concatenation etc. operations which can be implemented in a generic form. In this case, the designer has to select the appropriate predefined component from a HDL component library to his own async/sync resources. If the required functionality is more complex and application-specific, the operator has to be implemented and verified manually as an independent entity/architecture pair. It should be taken into consideration that AMDL has been intended to use in specific applications. For example if an FFT is needed by the application, it could be described with AMDL but, since there are numerous optimized soft-IPs implementing FFT, it would be sub-optimal to re-implement the FFT in AMDL. Instead, the designer could use the pre-defined HDL model of the generic FFT soft-core as a multicycle operator in the embedding system described with AMDL. The multicycle operators and dedicated pipelines should only be described with AMDL itself, if there are no applicable, pre-defined and optimized versions of the functionality required. The HDL linker transforms these entity/architecture pairs to the form required by the desired implementation scheme (into procedures and processes in case of behavioral RTL, VHDL blocks in case of fused structural RTL, or component instantiations in case of separated structural RTL).

A behavioral simulator is also under development. In the behavioral simulation a model of the datapath components which are not included in any pre-defined libraries should be created in a form that can be integrated into the behavioral simulation environment. This specific form only means a C++ function call with a pre-defined style of input and output parameter-passing methods and variable types. The simulator itself is based on SystemC, which is completely hidden from the designer.

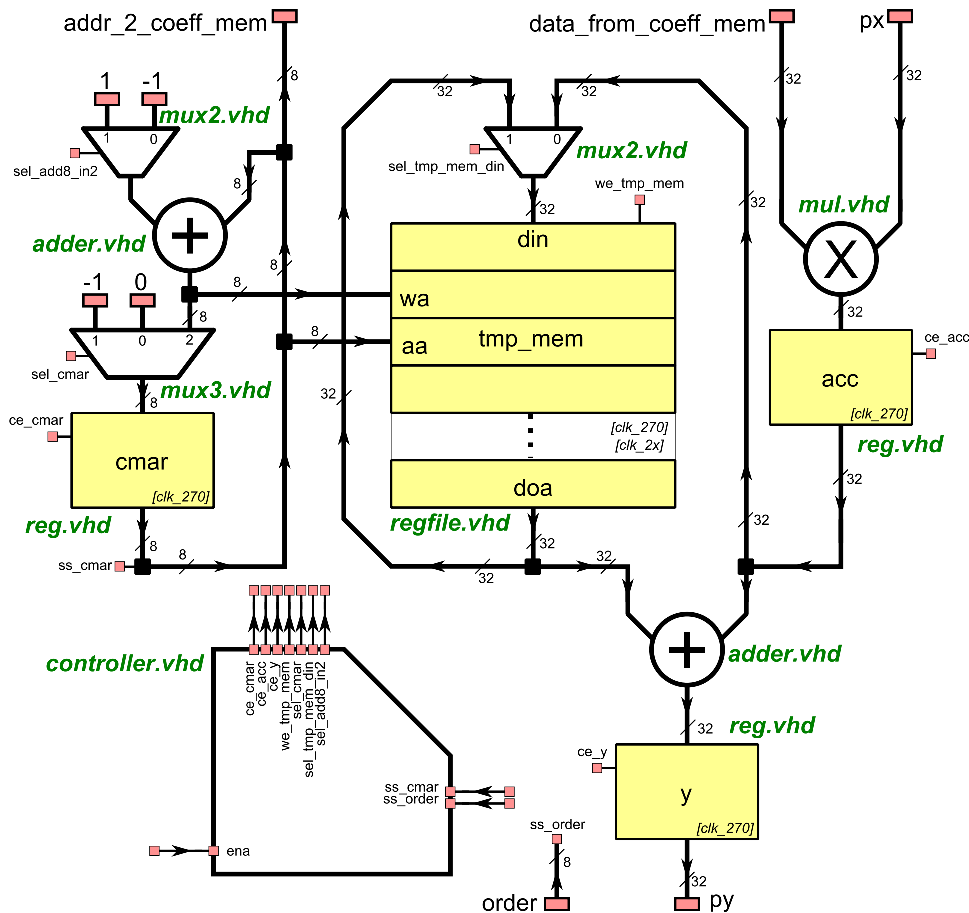

Fig. 5. Distributed structural RTL model structure of the FIR filter.

Table 6
VHDL language constructs implementing AMDL semantics.

AMDL resource/language construct	RTL (VHDL) implementation scheme		
	Structural		Behavioral
	Separated	Fused	
Dataport	Input/output port		
Reg	Pre-defined entity/architecture pair describing a register/register file	Pre-defined block describing a register/ register file	Signal
Regfile			Signal with a pre-defined array type
Asynchronous operator	Entity/architecture pair with a pre-defined interface	A process with a pre-defined sensitivity list inside a block	Procedure with a pre-defined interface
Synchronous/multicycle operator			Separated process inside an architecture body
Sequential instruction execution, loops, statement blocks	Specific FSMs		
Concurrent instructions	Multiple signal assignments in a single control state		
Conditional statements	If-then-else statement		

6. Experimental results

In this section, we present the test systems developed with the proposed design method. The first test system (μA_1) is a general-purpose single-core processor implementing a simple 3-address instruction set with a 3-stage 8-bit wide pipelined datapath. To minimize the control hazard occurrence the core performs 1-bit dynamic branch prediction with a 32-bit branch history table. To prevent data hazards the microarchitecture implements data forwarding. To decrease function call penalty, the register window technique is applied. The register file physically contains 16 registers which can be accessed as four overlapped register windows with 8 registers respectively. The datapath includes a shift&add multiplier unit (Table 8: mult) and an internal stack memory with 16 entries.

The second test system (μA_2) is a general-purpose multicore microprocessor. It comprises of a central core implementing an administrative instruction set with low performance microarchitecture and its datapath includes two instances of μA_1. The central core is responsible for data memory access and interrupt-management and the internal pipelined cores assure the computation performance of the system.

The third test system (μA_3) is an ASIP template optimized for DSP applications. It consists of programmable DSP pipelines and

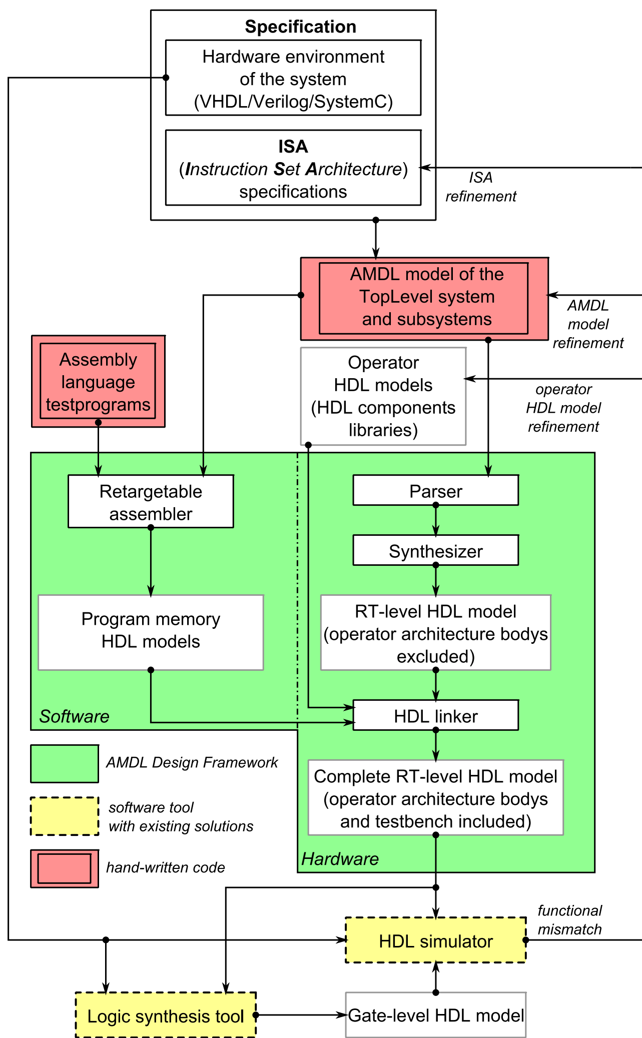

Fig. 6. AMDL design flow.

run-time configurable FIR filters (Table 8: FIR). The number of the pipelines and the filters are synthesis parameters and the FIR filter coefficient vector and the order of the FIR filters (up to 255) are run-time configurable parameters. The programmable DSP pipelines implement a 3-address instruction set with a 3-stage 32-bit wide pipelined datapath including a 32-word register file, a DSP ALU performing the special instructions shown in Table 7 and a 64-bit wide accumulator storing the results of the DSP instructions. The core provides high-speed external input and output FIFO interfaces.

To minimize the control hazard occurrence the core performs a 2-bit dynamic branch prediction with a 256-entry branch history table. To prevent data hazards the microarchitecture implements data forwarding.

Table 8 shows the architecture-elements applied and the synthesis results of the above described test systems. The μA_3 template core has been implemented with a single DSP pipeline with distributed structural RTL coding style and a single FIR filter with behavioral RTL coding style (there is no difference between the fused structural and the distributed structural RTL implementations in terms of resource requirement and operation frequency). The number of DSP pipelines and FIR filters can be adapted to the application (e.g. multiple input streams) with a negligible modification in the AMDL model.

Based on the results presented in Table 8 the following conclusions can be drawn. The logic synthesis based on the structure-oriented HDL coding style is more efficient in terms of resource requirement while

Table 7
DSP instructions of μA_3.

Instruction	Function
ff2rf	Read word from FIFO
rf2ff	Write word into FIFO
wba	Write back acc to the register file
mul	$Acc = x \times y$
mula	$Acc = acc + x \times y$
mulneg	$Acc = -(x \times y)$
mulnega	$Acc = acc - x \times y$
subsqr	$Acc = (x-y)^2$

the behavioral approach leads to a faster implementation. It has to be considered that this feature can only be utilized in case of relatively simple functionalities. In case of microprocessors complex enough to use in practice, only the structural RTL model can be implemented with reasonable resource requirement.

They key concepts behind these significant differences are resource sharing and model granularity. The most salient difference between the behavioral RTL and structural RTL implementation schemes is how the operators are embedded into their language environment. On one hand in behavioral RTL models the operator calls are directly embedded into the controlling FSM, which limits the designer in controlling resource sharing (binding). The other disadvantage is that the clocking and reset scheme of the FSM itself directly influences the implementation of operators during the synthesis process which may affect the synthesis tool when selecting hard heterogeneous blocks to use. On the other hand, the fine-grained structure of structural RTL models makes it possible to exploit the hard heterogeneous blocks of FPGAs more efficiently. The synthesis tools infer dedicated functionalities such as multiplication or shift-registers easier because of the well-defined interface and the isolated HDL description. This issue is not FPGA-related; the proposed IP macros of standard cell ASIC synthesis tools are similar to those in FPGA environments. Moreover, in structural RTL models the synthesizer does not need to perform resource sharing, because the FSM does not include any resources to share. Only the datapath includes such resources but they cannot be shared because the synthesis tools do not perform inter-entity optimizations. They cannot be unnecessarily duplicated either, because they are represented by single entity instantiations in the HDL model. In case of fused structural RTL models the synthesizer may perform resource sharing in the datapath but the resource duplication is impossible there, too.

In case of large systems the power-consumption issues addressed by the heterogeneous resources of FPGAs also have to be taken into consideration. The dynamic power-consumption of these systems is proportional to the resource usage but the hard heterogeneous blocks of FPGAs need significantly more power than general-purpose resources. The advantage of structural RTL modeling is that the designer can decide on the usage of the hard heterogeneous blocks, which improves the design space exploration capabilities.

The last four columns in Table 8 indicate the effectiveness of the AMDL-based design flow. The development time values are not predicted based on the code size but they represent actual development times. The values in the column labeled AMDL mean the development times of the AMDL models and the manually created operators' VHDL models embedded in it. Since the output RTL HDL models of the ARTL pre-synthesis process are identical to those applied in the hand-written projects represented in the column labeled VHDL (hand-written), their development times can be directly compared. The exact equality of the AMDL-based synthesis and the hand-optimized designs is caused by the fact that these hand-optimized designs represent a subset of those investigated during the development of the implementation schemes

Table 8
Synthesis results.

Test system	RTL style	Resource requirement				f_{max} (MHz)	Appr. dev. time (person-hour)		Lines of VHDL code	Lines of AMDL code
		FF	LUT	BRAM (kiB)	DSP48 slice		AMDL	VHDL (hand-written)		
mult	str	67	94	0	0	230.31	1	5	350	40
	bhv	67	142	0	0	353.36			70	
FIR	str	54	144	18	7	38.52	1	5	380	50
	bhv	43	486	0	8	83.57			90	
μA_1	str	268	386	90	0	50.92	20–30	90–100	2200	300
	bhv	572	1070	0	0	132.8			780	
μA_2	str	615	933	198	0	54.03	30–40	120–130	2650	550
	bhv	1227	2335	0	0	138.24			1100	
μA_3	mixed	474	1487	54	26	50.14	50–60	280–300	2900	500

presented in Section 4.2. It means, that the quality of results characteristic to hand-optimized designs can be achieved by AMDL pre-synthesis in a more efficient, less time-consuming way.

7. Summary and conclusion

The paper summarized the different solutions for the emerging demand for flexibility and reusability in System on Chip (SoC) design, including a novel solution to increase the efficiency of the hardware model generation for ASIPs. The programmable nature and the high computation performance make the concept of Application-Specific Instruction Set Processors (ASIPs) a promising approach. The primary modeling tools of full-custom ASIPs are the Architecture Description Languages (ADLs). Although there are solutions for transforming these formal models into synthesizable hardware, the final RTL implementations usually need a high amount of hand-optimizations before the logic synthesis in order to achieve appropriate quality of results.

This paper has proposed an efficient method for the optimized RTL model generation. The two key features of the presented approach are the possibility of a high optimization level achieved by a broad, abstract target architecture model, and the algorithmic language environment making a fast, detailed RTL design possible.

Based on the experimental results, the paper has proved that the proposed pre-synthesis method generates RTL hardware models in a more efficient way than traditional manual RTL coding which is widely used in today's ASIP synthesis flows. The quality of the RTL models generated by the new pre-synthesis method is identical to the hand-optimized designs but the required design time is much less than in case of manual RTL coding. Therefore, the conclusion can be drawn that the presented approach provides an efficient improvement in the trade-off exploration between development time and optimization level.

Acknowledgment

The work reported in the paper has been developed in the framework of the project “Talent care and cultivation in the scientific workshops of BME”. This project is supported by the Grant TÁMOP—4.2.2.B-10/1–2010-0009.

References

[1] K. Keutzer, S. Malik and A. Newton, From ASIC to ASIP: the next design discontinuity, in: Proceedings of the International Conference on Computer Design (ICCD), Freiburg, Germany, 2002.
[2] H. Meyr, System-on-chip for communication: the dawn of ASIPs and the dusk of ASICs, in: Proceedings of the IEEE Workshop on Signal Processing Systems (SIPS), Soul, Korea, 2003.
[3] S. Vakili, J. Langlois, G. Bois, Customised soft processor design: a compromise between architecture description languages and parameterisable processors, Comput. Digit. Tech. IET 7 (3) (2013) 122–131.
[4] C. Tradowsky, T. Harbaum, S. Deyerle and J. Becker, LImbiC: an adaptable architecture description language model for developing an application-specific image processor, in: Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2013.
[5] P. Coussy, D. Gajski, M. Meredith, A. Takach, An introduction to high-level synthesis, IEEE Des. Test Comput. 26 (4) (2009) 8–17.
[6] G. Martin és, G. Smith, High-level synthesis: past, present, and future, IEEE Des. Test Comput. 26 (2009) 18–25.
[7] D. Gajski, L. Ramachandran, Introduction to high-level synthesis, IEEE Des. Test Comput. 11 (4) (1994) 44–54.
[8] H. Blume, H. Feldkämper, T. Noll, Model-based exploration of the design space for heterogeneous system-on-chip, J. VLSI Signal Process. 40 (1) (2005) 19–34.
[9] H. Blume, H. Hübert, H. FeldKämper and T. Noll, Model-based exploration of the design space for heterogeneous system on chip, in: Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors (ASAP), San Jose, USA, 2002.
[10] M. Huber, P. Figuli, R. Girardey, D. Soudris, K. Siozios and J. Becker, A heterogeneous multicore system on chip with run-time reconfigurable virtual FPGA architecture, in: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), Shanghai, 2011.
[11] D. Bortolotti, C. Pinto, A. Marongiu, M. Ruggiero and L. Benini, VirtualSoC: a full-system simulation environment for massively parallel heterogeneous system-on-chip, in: Proceedings of the IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), Cambridge, MA, 2013.
[12] A. Jafri, A. Baghdadi, M. Jézéquel, ASIP-based universal demapper for multi-wireless standards, IEEE Embed. Syst. Lett. 1 (1) (2009) 9–13.
[13] T. Vogt, N. Wehn, A reconfigurable ASIP for convolutional and turbo decoding in an SDR environment, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 16 (10) (2008) 1309–1320.
[14] O. Muller, A. Baghdadi, M. Jézéquel, From parallelism levels to a multi-ASIP architecture for turbo decoding, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 17 (1) (2009) 92–102.
[15] X. Guan, Y. Fei, H. Lin, Hierarchical design of an application-specific instruction set processor for high-throughput and scalable FFT processing,, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 20 (3) (2012) 551–563.
[16] S. Saponara, L. Fanucci, S. Marsi, G. Ramponi, D. Kammler, E. Witte, Application-specific instruction-set processor for retinex-like image and video processing, IEEE Trans. Circuits Syst. 54 (7) (2007) 596–600.
[17] Z. Liu, K. Dickson, J. McCanny, Application-specific instruction set processor for SoC implementation of modern signal processing algorithms, IEEE Trans. Circuits Syst. 52 (4) (2005) 755–765.
[18] H. Peters, R. Sethuraman, A. Berić, P. Meuwissen, S. Balakrishnan, C. Alba Pinto, W. Kruijtzer, F. Ernst, G. Alkadi, J. van Meerbergen, G. de Haan, Application specific instruction-set processor template for motion estimation in video applications, IEEE Trans. Circuits Syst. Video Technol. 15 (4) (2005) 508–527.
[19] T. Good, M. Benaissa, Very small FPGA application-specific instruction processor for AES, IEEE Trans. Circuits Syst. 53 (7) (2006) 1477–1486.
[20] N. Neves, N. Sebastiao, A. Patricio, D. Matos, P. Tomas, P. Flores and N. Roma, BioBlaze: multi-core SIMD ASIP for DNA sequence alignment, in: Proceedings of the IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2013.
[21] D. Shin, A. Gerstlauer, R. Dömer, D. Gajski, An interactive design environment for C-based high-level synthesis of RTL processors, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 16 (4) (2008) 466–475.
[22] E. Casseau, B. Le Gal, Design of multi-mode application-specifi ccores based on high-level synthesis, Integr. VLSI J. 45 (2012) 9–21.
[23] R. Sinha, H. Patel, synASM: a high-level synthesis framework with support for parallel and timed constructs, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 31 (10) (2012) 1508–1521.
[24] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt and A. Nicolau, EXPRESSION: a language for architecture exploration through compiler/simulator

retargetability, in: Proceedings of the European Conference on Design, Automation and Test (DATE), Munich, Germany, 1999.
[25] S. Rigo, G. Araujo, M. Bartholomeu and R. Azevedo, ArchC: a systemC-based architecture description language, in: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing, 2004.
[26] A. Fauth, J. Van Praet and M. Freericks, Describing Instruction Set Processors using nML, in: Proceedings of the European Design and Test Conference, Paris, 1995.
[27] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, H. Meyr, A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 20 (11) (2001) 1338–1354.
[28] M. Hartoog, J. Rowson, P. Reddy, S. Desai, D. Dunlop, E. Harcourt and N. Khullar, Generation of software tools from processor descriptions for hardware/software codesign, in: Proceedings of the Design Automation Conference, Anaheim, CA, USA, 1997.
[29] P. Mishra, A. Kejariwal and N. Dutt, Synthesis-driven exploration of pipelined embedded processors, in: Proceedings of the 17th International Conference on VLSI Design, 2004.
[30] O. Schliebusch, A. Chattopadhyay, E. Witte, D. Kammler, G. Ascheid, R. Leupers and H. Meyr, Optimization techniques for ADL-driven RTL processor synthesis, in: Proceedings of the 16th International Workshop on Rapid System Prototyping (RSP 2005), 2005.
[31] S. Basu and R. Moona, High level synthesis from Sim-nML processor models, in: Proceedings of the 16th International Conference on VLSI Design, 2003.
[32] P. Horváth, G. Hosszú, F. Kovács, A proposed novel description language in the digital system modeling, in: Mehdi Khosrow-Pour (Ed.), Encyclopedia of Information Science and Technology, third edition, IGI Global, Hershey, New York, 2014, pp. 22–37.