

A Novel Methodology for the Design of Application-Specific Instruction-Set Processors (ASIPs) Using a Machine Description Language

Andreas Hoffmann, *Member, IEEE*, Tim Kogel, Achim Nohl, Gunnar Braun, Oliver Schliebusch, Oliver Wahlen, Andreas Wieferink, and Heinrich Meyr, *Fellow, IEEE*

Abstract—The development of application-specific instruction-set processors (ASIP) is currently the exclusive domain of the semiconductor houses and core vendors. This is due to the fact that building such an architecture is a difficult task that requires expertise in different domains: application software development tools, processor hardware implementation, and system integration and verification. This article presents a retargetable framework for ASIP design which is based on machine descriptions in the LISA language. From that, software development tools can be generated automatically including high-level language C compiler, assembler, linker, simulator, and debugger frontend. Moreover, for architecture implementation, synthesizable hardware description language code can be derived, which can then be processed by standard synthesis tools. Implementation results for a low-power ASIP for digital video broadcasting terrestrial acquisition and tracking algorithms designed with the presented methodology will be given. To show the quality of the generated software development tools, they are compared in speed and functionality with commercially available tools of state-of-the-art digital signal processor and μC architectures.

Index Terms—Architecture exploration, machine description languages, system-on-chip.

I. INTRODUCTION

IN CONSUMER electronics and telecommunications, high product volumes are increasingly going along with short lifetimes. Driven by the advances in semiconductor technology combined with the need for new applications like digital television and wireless broadband communications, the amount of system functionality realized on a single chip is growing enormously. Higher integration and, thus, increasing miniaturization have led to a shift from using distributed hardware components toward heterogeneous system-on-chip (SOC) designs [1]. Due to the complexity introduced by such SOC designs and time-to-market constraints, the designer's productivity has become the vital factor for successful products. For this reason a growing amount of system functions and signal processing algorithms is implemented in software rather than in hardware by employing embedded processor cores.

Manuscript received April 9, 2001; revised June 28, 2001. This paper was recommended by Guest Editor P. Marwedel.

The authors are with the Institute for Integrated Signal Processing Systems, Aachen University of Technology (RWTH), 52056 Aachen, Germany (e-mail: hoffmann@iss.rwth-aachen.de; kogel@iss.rwth-aachen.de; nohl@iss.rwth-aachen.de; braung@iss.rwth-aachen.de; schliebusch@iss.rwth-aachen.de; wahlen@iss.rwth-aachen.de; wieferink@iss.rwth-aachen.de; meyr@iss.rwth-aachen.de).

Publisher Item Identifier S 0278-0070(01)09989-4.

In the current technical environment, embedded processors and the necessary development tools are designed manually, with very little automation. This is because the design and implementation of an embedded processor, such as a digital-signal-processor (DSP) device embedded in a cellular phone, is a highly complex process composed of the following phases:

- 1) architecture exploration;
- 2) architecture implementation;
- 3) application software design;
- 4) system integration and verification.

During the *architecture exploration phase*, software development tools [i.e., high-level language (HLL) compiler, assembler, linker, and cycle-accurate simulator] are required to profile and benchmark the target application on different architectural alternatives. This process is usually an iterative one that is repeated until a best fit between selected architecture and target application is obtained. Every change to the architecture specification requires a complete new set of software development tools. As these changes on the tools are carried out mainly manually, this results in a long, tedious, and extremely error-prone process. Furthermore, the lack of automation makes it very difficult to match the profiling tools to an abstract specification of the target architecture. In the *architecture implementation phase*, the specified processor has to be converted into a synthesizable hardware description language (HDL) model. With this additional manual transformation, it is quite obvious that considerable consistency problems arise between the architecture specification, the software development tools, and the hardware implementation. During the *software application design phase*, software designers need a set of production-quality software development tools. Since the demands of the software application designer and the hardware processor designer place different requirements on software development tools, new tools are required. For example, the processor designer needs a cycle/phase-accurate simulator for hardware-software partitioning and profiling, which is very accurate, but inevitably slow, whereas the application designer demands more simulation speed than accuracy. At this point, the complete software development tool suite is usually reimplemented by hand—consistency problems are self-evident. In the *system integration and verification phase*, cosimulation interfaces must be developed to integrate the software simulator for the chosen architecture into a system simulation environment. These interfaces vary with the architecture that

is currently under test. Again, manual modification of the interfaces is required with each change of the architecture.

The efforts of designing a new architecture can be reduced significantly by using a retargetable approach based on a machine description. The language for instruction-set architectures (LISA) [2], [3] was developed for the automatic generation of consistent software development tools and synthesizable HDL code. A LISA processor description covers the instruction-set, behavioral, and timing models of the underlying hardware, thus, providing all essential information for the generation of a complete set of development tools including compiler, assembler, linker, and simulator. Moreover, it contains enough microarchitectural details to generate synthesizable HDL code of the modeled architecture. Changes on the architecture are easily transferred to the LISA model and are applied automatically to the generated tools and hardware implementation. In addition, speed and functionality of the generated tools allow usage even after the product development has been finished. Consequently, there is no need to rewrite the tools to upgrade them to production quality standard. In its predicate to represent an unambiguous abstraction of the real hardware, a LISA model description bridges the gap between hardware and software design. It provides the software developer with all required information and enables the hardware designer to synthesize the architecture from the same specification the software tools are based on.

The article is organized as follows. Section II reviews existing approaches on machine description languages and discusses their applicability for the design of application-specific instruction-set processor (ASIPs). Section III presents an overview on a typical ASIP design flow using LISA from specification to implementation. Moreover, different processor models are worked out which contain the required information the tools need for their retargetation. Besides, sample LISA code segments are presented showing how the different models are expressed in the LISA language. Section IV introduces the LISA processor design platform (LPDP). Following that, the different areas of application are illuminated in more detail. In Section V, the generated software development tools are presented in more detail with a focus on different simulation techniques that are applicable. Section VI shows the path to implementation and gives results for a case study that was carried out using the presented methodology. To prove the quality of the generated software development tools, in Section VII, simulation benchmark results are shown for modeled state-of-the-art processors. In Section VIII, requirements and limitations of the presented approach are explained. Section IX summarizes the article and gives an outlook on future research topics.

II. RELATED WORK

HDLs like VHDL or Verilog are widely used to model and simulate processors, but mainly with the goal to develop hardware. Using these models for architecture exploration and production quality software development tool generation has a number of disadvantages, especially for cycle-based or instruction-level processor simulation. They cover a huge amount

of hardware implementation details that are not needed for performance evaluation, cycle-based simulation, and software verification. Moreover, the description of detailed hardware structures has a significant impact on simulation speed [4], [5]. Another problem is that the extraction of the instruction set is a highly complex manual task and some instruction-set information, e.g., assembly syntax, cannot be obtained from HDL descriptions at all.

There are many publications on machine description languages providing instruction-set models. Most approaches using such models are addressing retargetable code generation [6]–[9]. Other approaches address retargetable code generation and simulation. The approaches of Maril [10] as part of the Marion environment and a system for very long instruction word (VLIW) compilation [11] are both using latency annotation and reservation tables for code generation. However, models based on operation latencies are too coarse for cycle-accurate simulation or even generation of synthesizable HDL code. The language nML was developed at Technische Universität Berlin [12], [13] and adopted in several projects [14]–[17]. However, the underlying instruction sequencer does not allow to describe the mechanisms of pipelining as required for cycle-based models. Processors with more complex execution schemes and instruction-level parallelism like the Texas Instruments (TI) TMS320C6x cannot be described even at the instruction-set level because of the numerous combinations of instructions. The same restriction applies to the language ISDL [18], which is very similar to nML. ISDL is an enhanced version of the nML formalism and allows the generation of a complete tool suite consisting of HLL compiler, assembler, linker, and simulator. Even the possibility of generating synthesizable HDL code is reported, but no results on the efficiency of the generated tools nor on the generated HDL code are given. The language EXPRESSION [19] allows the cycle-accurate processor description based on a mixed behavioral/structural approach. However, no results on simulation speed have been published nor is it clear if it is feasible to generate synthesizable HDL code automatically. The FlexWare2 environment [20] is capable of generating assembler, linker, simulator, and debugger from the Insulin formalism. A link to implementation is nonexistent, but test vectors can be extracted from the Insulin description to verify the HDL model. The HLL compiler is derived from a separate description targeting the CoSy [21] framework.

Recently, various ASIP development systems have been introduced [22]–[24] for systematic codesign of instruction-set and microarchitecture implementation using a given set of application benchmarks. The PEAS-III system [25] is an ASIP development environment based on a microoperation description of instructions that allows the generation of a complete tool suite consisting of HLL compiler, assembler, linker, and simulator including HDL code. However, no further information about the formalism is given that parameterizes the tool generators nor have any results been published on the efficiency of the generated tools. The MetaCore system [26] is a benchmark driven ASIP development system based on a formal representation language. The system accepts a set of benchmark programs and estimates the hardware cost and performance for

the configuration under test. Following that, software development tools and synthesizable HDL code are generated automatically. As the formal specification of the instruction-set architecture (ISA) is similar to the instruction-set processor specification formalism [27], complex pipeline operations as flushes and stalls can hardly be modeled. In addition, flexibility in designing the instruction set is limited to a predefined set of instructions. Tensilica Inc. customizes a reduced instruction-set computer (RISC) processor within the Xtensa system [28]. As the system is based on an architecture template comprising quite a number of base instructions, it is far too powerful and, thus, not suitable for highly application-specific processors, which only employ very few instructions in many cases.

Our interest in a complete retargetable tool suite for architecture exploration, production quality software development, architecture implementation, and system integration for a wide range of embedded processor architectures motivated the introduction of the language LISA used in our approach. In many aspects, LISA incorporates ideas that are similar to nML. However, it turned out from our experience with different DSP architectures that significant limitations of existing machine description languages must be overcome to allow the description of modern commercial embedded processors. For this reason, LISA includes improvements in the following areas:

- 1) capability to provide cycle-accurate processor models, including constructs to specify pipelines and their mechanisms including stalls, flushes, operation injection, etc.;
- 2) extension of the target class of processors including single instruction multiple data, VLIW, and superscalar architectures of real-world processor architectures;
- 3) explicit language statements addressing compiled simulation techniques;
- 4) distinction between the detailed bit-true description of operation behavior including side effects for the simulation and implementation on the one hand and assignment to arithmetical functions for the instruction selection task of the compiler on the other hand, which allows to determine freely the abstraction level of the behavioral part of the processor model;
- 5) strong orientation on the programming languages C/C++ (LISA is a framework that encloses pure C/C++ behavioral operation description);
- 6) support for instruction aliasing and complex instruction coding schemes.

III. ASIP DESIGN FLOW

Powerful application-specific programmable architectures are increasingly required in the DSP, multimedia, and networking application domains in order to meet demanding cost and performance requirements. The complexity of algorithms and architectures in these application domains prohibits an ad-hoc implementation and requests for an elaborated design methodology with efficient support in tooling. In this section, a seamless ASIP design methodology based on LISA will be introduced. Moreover, it will be demonstrated how the outlined concepts are captured by the LISA language elements. The

	application	LISA model	exploration result
1	assembly algorithm kernel	datapath model	ISA accurate profiling (data)
2	assembly program	instruction model	ISA accurate profiling (data+control)
3	revised assembly program	cycle-true model	cycle accurate profiling (data+control)
4	assembly program	RTL model	Hw cost + timing

Fig. 1. LISA-based ASIP development flow.

expressiveness of the LISA formalism providing high flexibility with respect to abstraction level and architecture category is especially valuable for the design of high-performance processors.

A. Architecture Exploration

The LISA-based methodology sets in after the algorithms, which are intended for execution on the programmable platform, are selected. The algorithm design is beyond the scope of LISA and is typically performed in an application-specific system-level design environment, e.g., COSSAP [29] for wireless communications or OPNET [30] for networking. The outcome of the algorithmic exploration is a pure functional specification usually represented by means of an executable prototype written in an HLL like C, together with a requirement document specifying cost and performance parameters. In the following, the steps of our proposed design flow depicted in Fig. 1 are described, where the ASIP designer refines successively the application jointly with the LISA model of the programmable target architecture.

First, the performance critical *algorithmic kernels* of the functional specification have to be identified. This task can be easily performed with a standard profiling tool, which instrumentalizes the application code in order to generate HLL execution statistics during the simulation of the functional prototype. Thus, the designer becomes aware of the performance critical parts of the application and is therefore prepared to define the data path of the programmable architecture on the assembly instruction level. Starting from a LISA processor model, which implements an arbitrary basic instruction set, the LISA model can be enhanced with parallel resources, special-purpose instructions, and registers in order to improve the performance of the considered application. At the same time, the algorithmic kernel of the application code is translated into assembly by making use of the specified special purpose instructions. By employing assembler, linker, and processor simulators derived from the LISA model (cf. Section V), the designer can iteratively profile and modify the programmable architecture in cadence with the application until both fulfill the performance requirements.

After the processing intensive algorithmic kernels are considered and optimized, the instruction set needs to be completed. This is accomplished by adding instructions to the LISA model that are dedicated to the low-speed control and configuration parts of the application. However, while these parts usually represent major portions of the application in terms of code amount,

they have only negligible influence on the overall performance. Therefore, it is very often feasible to employ the HLL C compiler derived from the LISA model and accept suboptimal assembly code quality in return for a significant cut in design time.

So far, the optimization has only been performed with respect to the software related aspects, while neglecting the influence of the microarchitecture. For this purpose, the LISA language provides capabilities to model cycle-accurate behavior of pipelined architectures. The LISA model is supplemented by the instruction pipeline and the execution of all instructions is assigned to the respective pipeline stage. If the architecture does not provide automatic interlocking mechanisms, the application code has to be revised to take pipeline effects into account. Now, the designer is able to verify that the cycle true processor model still satisfies the performance requirements.

At the last stage of the design flow, the HDL generator (see Section VI) can be employed to generate synthesizable HDL code for the base structure and the control path of the architecture. After implementing the dedicated execution units of the data path, strainable numbers on hardware cost and performance parameters (e.g., design size, power consumption, clock frequency) can be derived by running the HDL processor model through the standard synthesis flow. On this high level of detail, the designer can tweak the computational efficiency of the architecture by applying different implementations of the data path execution units.

B. LISA Language

The language LISA [2], [3] is aiming at the formalized description of programmable architectures, their peripherals, and interfaces. LISA closes the gap between purely structural oriented languages (VHDL, Verilog) and instruction-set languages.

LISA descriptions are composed of *resources* and *operations*. The declared resources represent the storage objects of the hardware architecture (e.g., registers, memories, pipelines), which capture the state of the system. Operations are the basic objects in LISA. They represent the designer's view of the behavior, the structure, and the instruction set of the programmable architecture. A detailed reference of the LISA language can be found in [31].

The process of generating software development tools and synthesizing the architecture requires information on architectural properties and the instruction-set definition as depicted in Fig. 2. These requirements can be grouped into different architectural models—the entirety of these models constitutes the abstract model of the target architecture. The LISA machine description provides information consisting of the following model components.

- The *memory model* lists the registers and memories of the system with their respective bit widths, ranges, and aliasing. The compiler gets information on available registers and memory spaces. The memory configuration is provided to perform object code linking. During simulation, the entirety of storage elements represents the state of the processor, which can be displayed in the debugger. The

	memory model	resource model	behavioral model	instruction set model	timing model	micro-architecture model
HLL-compiler	register allocation	instruction scheduling	instruction selection	-	instruction scheduling	-
assembler	-	-	-	instruction translation	-	-
linker	memory allocation	-	-	-	-	-
simulator	simulation of storage	-	operation simulation	decoder/classifier	operation scheduling	-
debugger	display configuration	profiling	-	-	-	-
HDL generator	basic structure	write conflict resolution	-	instruction decoder	operation scheduling	operation grouping

Fig. 2. Model requirements for ASIP design

```

RESOURCE
{
    PROGRAM_COUNTER    int           PC;
    REGISTER            signed int   R[0..7];

    DATA_MEMORY       signed int    RAM[0..255];
    PROGRAM_MEMORY     unsigned int  ROM[0..255];

    PIPELINE ppu_pipe = { FI; ID; EX; WB };

    PIPELINE_REGISTER IN ppu_pipe
    {
        bit[6]    Opcode;
        short     operandA;
        short     operandB;
    };
}

```

Fig. 3. Specification of the *memory model*.

HDL code generator derives the basic architecture structure.

In LISA, the resource section lists the definitions of all objects which are required to build the memory model. A sample resource section of the ICORE architecture described in [32] is shown in Fig. 3. The resource section begins with the keyword RESOURCE followed by (curly) braces enclosing all object definitions. The definitions are made in C style and can be attributed with keywords such as REGISTER, PROG_COUNTER, etc. These keywords are not mandatory, but they are used to classify the definitions in order to configure the debugger display. The resource section in Fig. 3 shows the declaration of program counter, register file, memories, the four-stage instruction pipeline, and pipeline registers.

- The *resource model* describes the available hardware resources and the resource requirements of operations. Resources reflect properties of hardware structures that can be accessed exclusively by one operation at a time. The instruction scheduling of the compiler depends on this information. The HDL code generator uses this information for resource conflict resolution.

Besides the definition of all objects, the resource section in a LISA processor description provides information about the availability of hardware resources. By this, the property of several ports, e.g., to a register bank or a memory is reflected. Moreover, the behavior section within LISA operations announces the use of processor resources. This takes place in the section header using the keyword USES in conjunction with the resource name and

```

RESOURCE
{
  REGISTER    unsigned int    R([0..7])6;
  DATA_MEMORY signed int    RAM([0..15]);
}

OPERATION NEG_RM {
  BEHAVIOR
  USES (IN R[];
        OUT RAM[]);
  {
    /* C-code */
    RAM[address] = (-1) * R[index];
  }
}

```

Fig. 4. Specification of the resource model.

```

OPERATION COMPARE_IMM {
  DECLARE {
    LABEL index;
    GROUP src1, dest = { register };
  }
  CODING { 0b10011 index=0bx[5] src1 dest }
  SYNTAX { "CMP" src1 "-", "index", "-", "dest" }
  SEMANTICS { CMP (dest,src1,index) }
}

```

Fig. 5. Specification of the instruction-set model.

the information if the used resource is read, written, or both (IN, OUT, or INOUT, respectively).

For illustration purposes, a sample LISA code taken from the ICORE architecture is shown in Fig. 4. The definition of the availability of resources is carried out by enclosing the C-style resource definition with round braces followed by the number of simultaneously allowed accesses. If the number is omitted, one allowed access is assumed. The figure shows the declaration of a register bank and a memory with six and one ports, respectively. Furthermore, the behavior section of the operation announces the use of these hardware resources for read and write.

- The *instruction-set model* identifies valid combinations of hardware operations and admissible operands. It is expressed by the assembly syntax, instruction-word coding, and the specification of legal operands and addressing modes for each instruction. Compilers and assemblers can identify instructions based on this model. The same information is used at the reverse process of decoding and disassembling.

In LISA, the instruction-set model is captured within operations. Operation definitions collect the description of different properties of the instruction-set model, which are defined in several sections.

- 1) The CODING section describes the binary image of the instruction word.
- 2) The SYNTAX section describes the assembly syntax of instructions, operands, and execution modes.
- 3) The SEMANTICS section specifies the transition function of the instruction.

Fig. 5 shows an excerpt of the ICORE LISA model contributing to the instruction-set model information on the compare immediate instruction. The DECLARE section contains local declarations of identifiers and admissible operands. Operation *register* is not shown in the figure, but

```

OPERATION register
{
  DECLARE { LABEL index; }
  CODING { index=0bx[4] }
  EXPRESSION { R[index] }
}

OPERATION ADD {
  DECLARE { GROUP src1,src2,dest = { register }; }
  CODING { 0b010010 src1 src2 dest }
  BEHAVIOR
  {
    /* C-code */
    dest = src1 + src2;
    saturate(&dest);
  }
}

```

Fig. 6. Specification of the behavioral model.

comprises the definition of the valid coding and syntax for *src1* and *dest*, respectively.

- The *behavioral model* abstracts the activities of hardware structures to operations changing the state of the processor for simulation purposes. The abstraction level of this model can range widely between the hardware implementation level and the level of HLL statements.

The BEHAVIOR and expression sections within LISA operations describe components of the behavioral model. Here, the behavior section contains pure C code that is executed during simulation, whereas the expression section defines the operands and execution modes used in the context of operations. An excerpt of the ICORE LISA model is shown in Fig. 6. Depending on the coding of the *src1*, *src2*, and *dest* field, the behavior code of operation ADD works with the respective registers of register bank *R*. As arbitrary C code is allowed, function calls can be made to libraries that are later linked to the executable software simulator.

- The *timing model* specifies the activation sequence of hardware operations and units. The instruction latency information lets the compiler find an appropriate schedule and provides timing relations between operations for simulation and implementation.

Several parts within a LISA model contribute to the timing model. First, the declaration of pipelines in the resource section. The declaration starts with the keyword PIPELINE, followed by an identifying name and the list of stages. Second, operations are assigned to pipeline stages by using the keyword IN and providing the name of the pipeline and the identifier of the respective stage, such as

```
OPERATION name_of_operation IN ppu_pipe.EX. (1)
```

Third, the ACTIVATION section in the operation description is used to activate other operations in the context of the current instruction. The activated operations are launched as soon as the instruction enters the pipeline stage to which the activated operation is assigned. Nonassigned operations are launched in the pipeline stage of their activation.

To exemplify this, Fig. 7 shows sample LISA code taken from the ICORE architecture. Operations *CORDIC* and *WriteBack* are assigned to stages EX and WB of pipeline *ppu_pipe*, respectively. Here, operation *CORDIC* activates

```

RESOURCE
{
  PIPELINE ppu_pipe = { FI; ID; EX; WB };
}

OPERATION CORDIC IN ppu_pipe.EX
{
  ACTIVATION { WriteBack }
  BEHAVIOR {
    PIPELINE_REGISTER(ppu_pipe, EX/WB).ResultE = cordic();
  }
}

OPERATION WriteBack IN ppu_pipe.WB {
  BEHAVIOR {
    R[value] = PIPELINE_REGISTER(ppu_pipe, EX/WB).ResultE;
  }
}

```

Fig.7. Specification of the timing model.

operation *WriteBack*, which will be launched in the following cycle (in correspondence to the spacial ordering of pipeline stages) in case of an undisturbed flow of the pipeline.

Moreover, in the activation section, pipelines are controlled by means of predefined functions *stall*, *shift*, *flush*, *insert*, and *execute*, which are automatically provided by the LISA environment for each pipeline declared in the resource section. All these pipeline control functions can be applied to single stages as well as whole pipelines, for example

```
PIPELINE(ppu_pipe,EX/WB).stall();
```

 (2)

Using this very flexible mechanism, arbitrary pipelines, hazards and mechanisms like forwarding can be modeled in LISA.

- The *microarchitecture model* allows grouping of hardware operations to functional units and contains the exact microarchitecture implementation of structural components such as adders, multipliers, etc. This enables the HDL generator to generate the appropriate HDL code from a more abstract specification.

Analogous to the syntax of the VHDL language, operation grouping to functional units is formalized using the keyword *ENTITY* in the resource section of the LISA model, for example

```

ENTITY Alu
{
  Add, Sub
}.

```

 (3)

Here, LISA operations *Add* and *Sub* are assigned to the functional unit *Alu*. Information on the exact microarchitectural implementation of structural components can be included into the LISA model, e.g., by calling DesignWare components [33] from within the behavior section or by inlining HDL code.

IV. LISA PROCESSOR DESIGN PLATFORM

The LPDP is an environment that allows the automatic generation of software development tools for architecture exploration, hardware implementation, software development tools

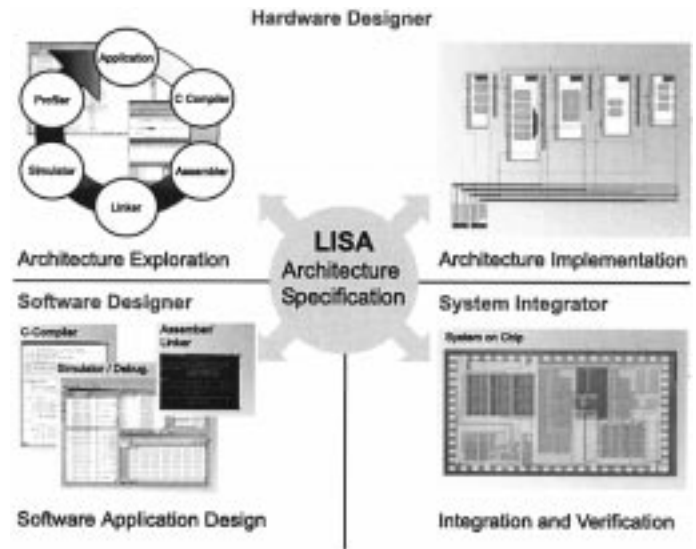


Fig.8. LISA processor design environment.

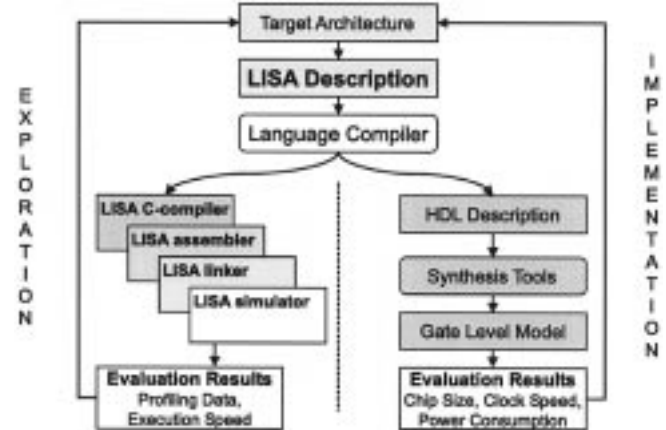


Fig. 9. Exploration and implementation.

for application design, and hardware–software cosimulation interfaces from one sole specification of the target architecture in the LISA language. Fig. 8 shows the components of the LPDP environment.

A. Hardware Designer Platform—for Exploration and Processor Generation

As indicated in Section III, architecture design requires the designer to work in two fields (see Fig. 9): on the one hand, the development of the software part including compiler, assembler, linker, and simulator and, on the other hand, the development of the target architecture itself.

The software simulator produces profiling data and, thus, may answer questions concerning the instruction set, the performance of an algorithm, and the required size of memory and registers. The required silicon area or power consumption can only be determined in conjunction with a synthesizable HDL model. To accommodate these requirements, the LISA hardware designer platform can generate the following tools:

- 1) LISA language debugger for debugging the instruction-set with a graphical debugger frontend;

- 2) exploration C compiler for the noncritical parts of the application;
- 3) exploration assembler, which translates text-based instructions into object code for the respective programmable architecture;
- 4) exploration linker, which is controlled by a dedicated linker command file;
- 5) ISA simulator providing extensive profiling capabilities, such as instruction execution statistics and resource utilization.

Besides the ability to generate a set of software development tools, synthesizable HDL code (both VHDL and Verilog) for the processors control path and instruction decoder can be generated automatically from the LISA processor description. This also comprises the pipeline and pipeline controller including complex interlocking mechanisms, forwarding, etc. For the data path, hand-optimized HDL code has to be inserted manually into the generated model. This approach has been chosen as the data path typically represents the critical part of the architecture in terms of power consumption and speed (critical path).

It is obvious that deriving both software tools and hardware implementation model from one sole specification of the architecture in the LISA language has significant advantages: only one model needs to be maintained, changes on the architecture are applied automatically to the software tools, and the implementation model and the consistency problem among the software tools and between software tools and implementation model is reduced significantly.

B. Software Designer Platform—for Software Application Design

To cope with the requirements of functionality and speed in the software design phase, the tools generated for this purpose are an enhanced version of the tools generated during architecture exploration phase. The generated simulation tools are enhanced in speed by applying the compiled simulation principle [34]—where applicable—and are faster by one to two orders of magnitude than the tools currently provided by architecture vendors. As the compiled simulation principle requires the content of the program memory not to be changed during the simulation run, this holds true for most DSPs. However, for architectures running the program from external memory or working with operating systems that load/unload applications to/from internal program memory, this simulation technique is not suitable. For this purpose, an interpretive simulator is also provided.

C. System Integrator Platform—for System Integration and Verification

Once the processor software simulator is available, it must be integrated and verified in the context of the whole system (SOC), which can include a mixture of different processors, memories, and interconnect components. In order to support the system integration and verification, the LPDP system integrator platform provides a well-defined application programming interface (API) to interconnect the instruction-set simulator generated from the LISA specification with other simulators. The API allows to control the simulator by stepping, running, and

setting breakpoints in the application code and by providing access to the processor resources.

Sections V–VII will present the different areas addressed by the LPDP in more detail—software development tools and HDL code generation. Additionally, Section VII will prove the high quality of the generated software development tools by comparing them to those shipped by the processor vendors.

V. SOFTWARE DEVELOPMENT TOOLS

The feasibility to generate automatically HLL C compilers, assemblers, linkers, and ISA simulators from LISA processor models enables the designer to explore the design space rapidly. In this section, specialties and requirements of these tools are discussed with particular focus on different simulation techniques.

A. Assembler and Linker

The LISA assembler processes textual assembly source code and transforms it into linkable object code for the target architecture. The transformation is characterized by the instruction-set information defined in a LISA processor description. Besides the processor-specific instruction set, the generated assembler provides a set of pseudoinstructions (directives) to control the assembling process and initialize data. Section directives enable the grouping of assembled code into sections which can be positioned separately in the memory by the linker. Symbolic identifiers for numeric values and addresses are standard assembler features and are supported as well. Moreover, besides mnemonic-based instruction formats, C-like algebraic assembly syntax can be processed by the LISA assembler.

The linking process is controlled by a linker command file that keeps a detailed model of the target memory environment and an assignment table of the module sections to their respective target memories. Moreover, it is suitable to provide the linker with an additional memory model that is separated from the memory configuration in the LISA description and allows linking code into external memories that are outside the architecture model.

B. Simulator

Due to the large variety of architectures and the facility to develop models on different levels of abstraction in the domain of time and architecture (see Section III), the LISA software simulator incorporates several simulation techniques ranging from the most flexible *interpretive* simulation to more application- and architecture-specific *compiled* simulation techniques.

Compiled simulators offer a significant increase in instruction (cycle) throughput. However, the compiled simulation technique is not applicable in any case. To cope with this problem, the most appropriate simulation technique for the desired purpose (debugging, profiling, verification), architecture (instruction-accurate, cycle-accurate), and application (DSP kernel, operating system) can be chosen before the simulation is run. An overview of the available simulation techniques in the generated LISA simulator is given in the following.

The *interpretive* simulation technique is employed in most commercially available instruction-set simulators. In general,

interpretive simulators run significantly slower than compiled simulators. However, unlike compiled simulation, this simulation technique can be applied to *any* LISA model and application.

Dynamically scheduled, compiled simulation reduces simulation time by performing the steps of instruction decoding and operation sequencing prior to simulation. This technique cannot be applied to models using external memories or applications consisting of self-modifying program code.

Besides the compilation steps performed in dynamic scheduling, *static scheduling* and *code translation* additionally implement operation instantiation. While the latter technique is used for instruction-accurate models, the former is suitable for cycle-accurate models including instruction pipelines. Beyond, the same restrictions apply as for dynamically scheduled simulation.

A detailed discussion of the different compiled simulation techniques is given in the following sections, while performance results are given in Section VII. The interpretive simulator is not discussed.

1) *Compiled Simulation*: The objective of compiled simulation is to reduce the simulation time. Considering instruction-set simulation, efficient runtime reduction can be achieved by performing repeatedly executed operations only once *before* the actual simulation is run, thus inserting an additional translation step between application load and simulation. The preprocessing of the application code can be split into three major steps [35].

- 1) Within the step of *instruction decoding*, instructions, operands, and modes are determined for each instruction word found in the executable object file. In compiled simulation, the instruction decoding is only performed once for each instruction, whereas interpretive simulators decode the same instruction multiple times, e.g., if it is part of a loop. This way, the instruction decoding is completely omitted at runtime, thus reducing simulation time significantly.
- 2) *Operation sequencing* is the process of determining all operations to be executed for the accomplishment of each instruction found in the application program. During this step, the program is translated into a table-like structure indexed by the instruction addresses. The table lines contain pointers to functions representing the behavioral code of the respective LISA operations. Although all involved operations are identified during this step, their temporal execution order is still unknown.
- 3) The determination of the operation timing (scheduling) is performed within the step of *operation instantiation and simulation loop unfolding*. Here, the behavior code of the operations is instantiated by generating the respective function calls for each instruction in the application program, thus, unfolding the simulation loop that drives the simulation into the next state.

Besides fully compiled simulation, which incorporates all of the above steps, partial implementations of the compiled principle are possible by performing only some of these steps. The accomplishment of each of these steps gives a further runtime

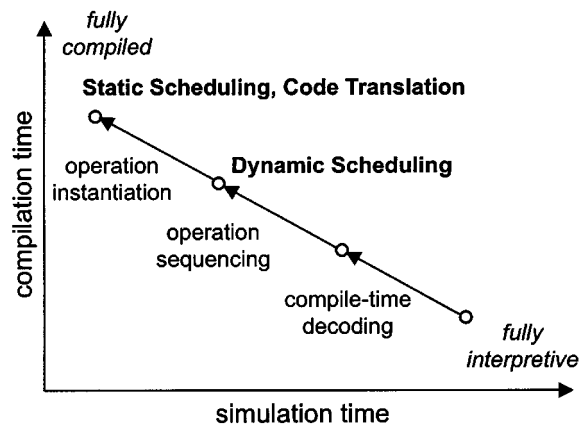


Fig. 10. Levels of compiled simulation.

reduction, but also requires a nonneglectable amount of compilation time. The tradeoff between compilation time and simulation time is (qualitatively) shown in Fig. 10.

There are two levels of compiled simulation that are of particular interest—*dynamic scheduling* and *static scheduling* respective *code translation*. In the case of dynamic scheduling, the task of selecting operations from overlapping instructions in the pipeline is performed at runtime of the simulation. The static scheduling already schedules the operations at compile time.

2) *Dynamic Scheduling*: As shown in Fig. 10, the dynamic scheduling performs instruction decoding and operation sequencing at compile time. However, the temporal execution order of LISA operations is determined at simulator runtime. While the operation scheduling is rather simple for instruction-accurate models, it becomes a complex task for models with instruction pipelines.

In order to reflect the instructions' timing exactly and to consider all possibly occurring pipeline effects such as flushes and stalls, a generic pipeline model is employed simulating the instruction pipeline at runtime. The pipeline model is parameterized by the LISA model description and can be controlled via predefined LISA operations. These operations include:

- 1) insertion of operations into the pipeline (stages);
- 2) execution of all operations residing in the pipeline;
- 3) pipeline shift;
- 4) removal of operations (flush);
- 5) halt of entire pipeline or particular stages (stall).

Unlike for statically scheduled simulation, operations are inserted into and removed from the pipeline dynamically, which means that each operation injects further operations upon its execution. The information about operation timing is provided in the LISA description, i.e., by the activation section as well as the assignment of operations to pipeline stages (see *timing model* in Section III-B).

It is obvious that the maintenance of the pipeline model at simulation time is expensive. Execution profiling on the generated simulators for the TI TMS320C62xx [36] and TMS320C54x [37] revealed that more than 50% of the simulator's runtime is consumed by the simulation of the pipeline.

The situation could be improved by implementing the step of operation instantiation, consequently superseding the need for

pipeline simulation. This, in turn, implies *static* scheduling, in other words, the determination of the operation timing due to overlapping instructions in the pipeline taking place at compile time.

Although there is no pipeline model in instruction-accurate processor models, it will be shown that operation instantiation also gives a significant performance increase for these models. Beyond that, operation instantiation is relatively easy to implement for instruction-accurate models (in contrast to pipelined models).

3) *Static Scheduling*: Generally, operation instantiation can be described as the generation of an individual piece of (behavioral) simulator code for each instruction found in the application program. While this is straightforward for instruction-accurate processor models, cycle-true pipelined models require a more sophisticated approach.

Considering instruction-accurate models, the shortest temporal unit that can be executed is an instruction. That means, the actions to be performed for the execution of an individual instruction are determined by the instruction alone. In the simulation of pipelined models, the granularity is defined by cycles. However, since several instructions might be active at the same time due to overlapping execution, the actions performed during a single cycle are determined by the respective state of the instruction pipeline. As a consequence, instead of instantiating operations for each single instruction of the application program, behavioral code for each occurring *pipeline state* has to be generated. Several of such pipeline states might exist for each instruction, depending on the execution context of the instruction, i.e., the instructions executed in the preceding and following cycles.

As pointed out previously, the principle of compiled simulation relies on an additional translation step taking place before the simulation is run. This step is performed by a so-called *simulation compiler*, which implements the three steps presented in Section V-A. Obviously, the simulation compiler is a highly architecture-specific tool, which is, therefore, retargeted from the LISA model description.

Operation instantiation: The objective of static scheduling is the determination of all possible pipeline states according to the instructions found in the application program. For purely sequential pipeline flow, i.e., in case no control hazards occur, the determination of the pipeline states can be achieved simply by overlapping consecutive instructions subject to the structure of the pipeline. In order to store the generated pipeline states, *pipeline state tables* are used, providing an intuitive representation of the instruction flow in the pipeline. Inserting instructions into pipeline state tables is referred to as *scheduling* in the following.

A pipeline state table is a two-dimensional array storing pointers to LISA operations. One dimension represents the location within the application, the other the location within the pipeline, i.e., the stage in which the operation is executed. When a new instruction has to be inserted into the state table, both intrainstruction and interinstruction precedence must be considered to determine the table elements in which the corresponding operations will be entered. Consequently, the actual time an operation is executed at depends on the scheduling

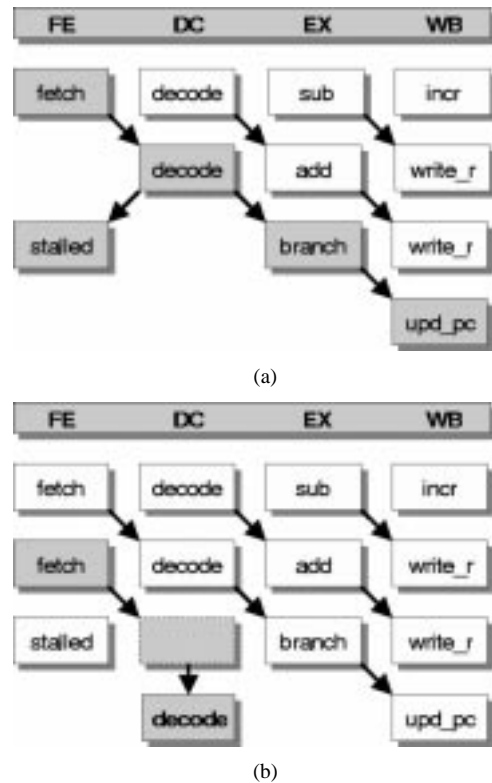


Fig. 11. Inserting instructions into pipeline state table. (a) Pipeline state table after branch instruction insertion. (b) Pipeline state table for next instruction considering the branch stall cycle.

of the preceding instruction as well as the scheduling of the operation(s) assigned to the preceding pipeline stage within the current instruction. Furthermore, control hazards causing pipeline stalls and/or flushes influence the scheduling of the instruction following the occurrence of the hazard.

A simplified illustration of the scheduling process is given in Fig. 11. Fig. 11(a) shows the pipeline state table after a branch instruction has been inserted, composed of the operations fetch, decode, branch, and upd_pc as well as a stall operation. The table columns represent the pipeline stages, the rows represent consecutive cycles (with earlier cycles in upper rows). The arrows indicate activation chains.

The scheduling of a new instruction always follows the intrainstruction precedence, which means that fetch is scheduled before decode, decode before branch, and so on. The appropriate array element for fetch is determined by its assigned pipeline stage (FE) and according to interinstruction precedences. Since the branch instruction follows the add instruction (which has already been scheduled), the fetch operation is inserted below the first operation of add [not shown in Fig. 11(a)]. The other operations are inserted according to their precedences.

The stall of pipeline stage FE, which is issued from the decode operation of branch, is processed by tagging the respective table element as stalled. When the next instruction is scheduled, the stall is accounted for by moving the decode operation to the next table row respective next cycle [see Fig. 11(b)]. Pipeline flushes are handled in a similar manner: if a selected table element is marked as flushed, the scheduling of the current instruction is abandoned.

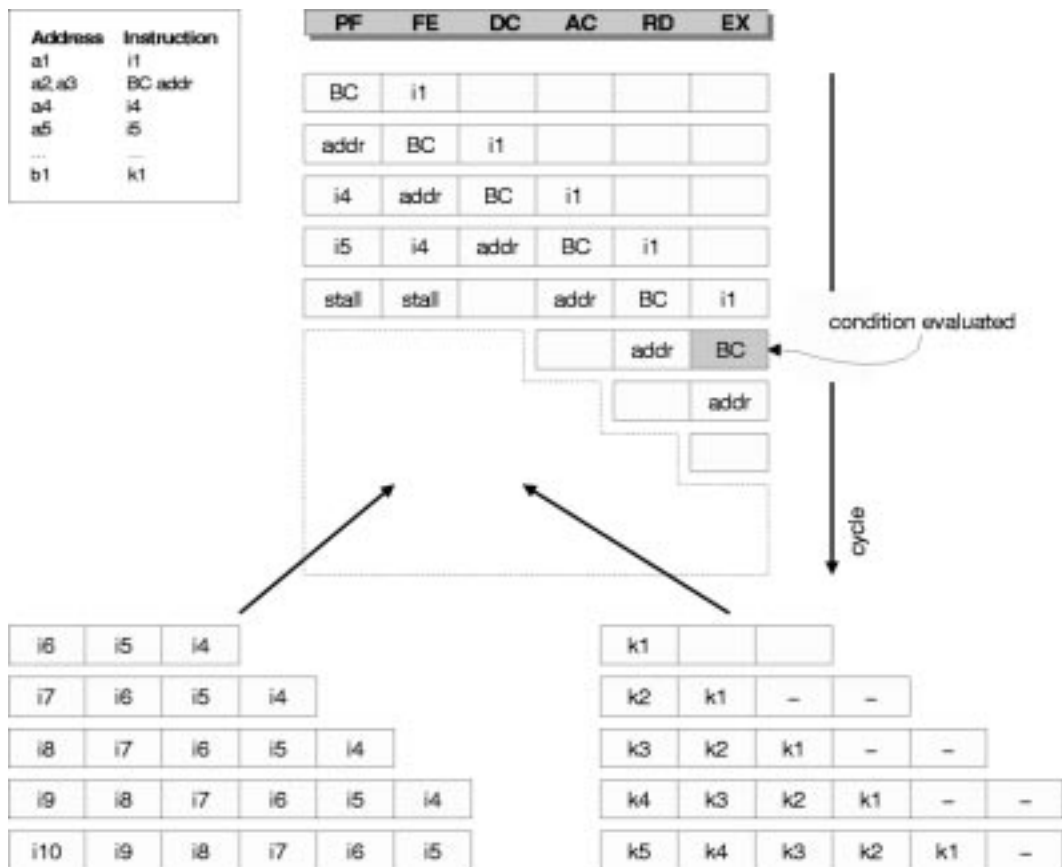


Fig. 12. Pipeline behavior for a conditional branch.

Assuming purely sequential instruction flow, the task of establishing a pipeline state table for the entire application program is very straightforward. However, every (sensible) application contains a certain amount of control flow (e.g., loops) interrupting this sequential execution. The occurrence of such control flow instructions makes the scheduling process extremely difficult or in a few cases even impossible.

Generally, all instructions modifying the program counter cause interrupts in the control flow. Furthermore, only instructions providing an immediate target address, i.e., branches and calls whose target address is known at compile time, can be scheduled statically. If indirect branches or calls occur, it is inevitable to switch back to dynamic scheduling at runtime.

Fortunately, most control flow instructions can be scheduled statically. Fig. 12 exemplarily shows the pipeline states for a conditional branch instruction as found in the TMS320C54x’s instruction set. Since the respective condition cannot be evaluated until the instruction is executed, scheduling has to be performed for both eventualities (condition true respective false), splitting the program into alternative execution paths. The selection of the appropriate block of prescheduled pipeline states is performed by switching among different state tables at simulator runtime. In order to prevent from doubling the entire pipeline state table each time a conditional branch occurs, alternative execution paths are left as soon as an already generated state has been reached. Unless several conditional instructions reside in the pipeline at the same time, these usually have the length of a few rows.

```

switch (pc) {
case 0x1584: fetch(); decode(); sub(); write_registers();
case 0x1585: fetch(); decode(); test_condition(); add();
case 0x1586: branch(); write_registers();
              fetch(); update_pc();
              fetch(); decode();
              fetch(); decode(); load(); goto _0x1400_;
}
    
```

Fig. 13. Generated simulator code.

Simulator instantiation: After all instructions of the application program have been processed, and, thus, the entire operation schedule has been established, the simulator code can be instantiated. The simulation compiler backend thereby generates either C code or an operation table with the respective function pointers, both describing alternative representations of the application program. Fig. 13 shows a simplified excerpt of the generated C code for a branch instruction. Cases represent instructions, while a new line starts a new cycle.

4) *Instruction-Based Code Translation:* The need for a scheduling mechanism arises from the presence of an instruction pipeline in the LISA model. However, even instruction-accurate processor models without pipeline benefit from the step of operation instantiation. The technique applied here is called *instruction-based code translation*. Due to the absence of instruction overlap, simulator code can be instantiated for each instruction independently, thus, simplifying simulator generation to the concatenation of the respective behavioral code specified in the LISA description.

In contrast to direct binary-to-binary translation techniques [38], the translation of target-specific into host-specific machine code uses C source code as intermediate format. This keeps the simulator portable and, thus, independent from the simulation host.

Since the instruction-based code translation generates program code that linearly increases in size with the number of instructions in the application, the use of this simulation technique is restricted to small- and medium-sized applications (less than $\approx 10k$ instructions, depending on model complexity). For large applications, the resultant worse cache utilization on the simulation host reduces the performance of the simulator significantly.

VI. ARCHITECTURE IMPLEMENTATION

As we are targeting the development of ASIPs, which are highly optimized for one specific application domain, the HDL code generated from a LISA processor description has to fulfill tight constraints to be an acceptable replacement for handwritten HDL code by experienced designers. Especially power consumption, chip area, and execution speed are critical points for this class of architectures. For this reason, the LPDP platform does not claim to be able to efficiently synthesize the complete HDL code of the target architecture. Especially the data path of an architecture is highly critical and must in most cases be optimized manually. Frequently, full-custom design techniques must be used to meet power consumption and clock speed constraints. For this reason, the generated HDL code is limited to the following parts of the architecture:

- 1) coarse processor structures such as register set, pipeline, pipeline registers, and test interface;
- 2) instruction decoder setting data and control signals that are carried through the pipeline and activating the respective functional units executed in context of the decoded instruction;
- 3) pipeline controller handling different pipeline interlocks, pipeline register flushes, and supporting mechanisms such as data forwarding.

Additionally, hardware operations as they are described in the LISA model can be grouped to functional units (see *microarchitecture model* in Section III-B). Those functional units are generated as *wrappers*, i.e., the ports of the functional units as well as the interconnects to the pipeline registers and other functional units are generated automatically while the content needs to be filled manually with code. Emerging driver conflicts in context with the interconnects are resolved automatically by the insertion of multiplexers.

The disadvantage of rewriting the data path in the HDL description by hand is that the behavior of hardware operations within those functional units has to be described and maintained twice—on the one hand, in the LISA model and, on the other hand, in the HDL model of the target architecture. Consequently, a problem here is verification, which will be addressed in future research.

A. LISA Language Elements for HDL Synthesis

The following sections will show in detail how different parts of the LISA model contribute to the generated HDL model of the target architecture.

```
RESOURCE
{
  REGISTER S32      R([0..7])6; /* GP Registers */
  REGISTER bit[11]  AR([0..3]); /* Address Registers */

  DATA_MEMORY S32 RAM([0..255]); /* Memory Space */
  PROGRAM_MEMORY U32 ROM([0..255]); /* Instruction ROM */

  PORT bit[1]      RESET;      /* Reset pin */
  PORT bit[32]     STATE_BUS;  /* Processor state bus */

  PIPELINE ppu_pipe = { FI; ID; EX; WB };
  PIPELINE_REGISTER IN ppu_pipe {
    bit[6] Opcode;
    ...
  };
}
```

Fig. 14. Resource declaration in the LISA model of the ICORE architecture.

1) *Resource Section*: The resource section provides general information about the structure of the architecture (e.g., registers, memories, and pipelines, see *resource/memory model* in Section III-B). Based on this information, the coarse structure of the architecture can be generated automatically. Fig. 14 shows an excerpt resource declaration of the LISA model of the ICORE architecture [32], which was used in our case study.

The ICORE architecture has two different register sets—one for general-purpose use named *R*, consisting of eight separate registers with 32 bits, and one for the address registers named *AR*, consisting of four elements each with 11 bits. The round brackets indicate the maximum number of simultaneously accesses allowed for the respective register bank—six for the general purpose register *R* and one for the address register set. From that, the respective number of access ports to the register banks can be generated automatically. With this information—bit-true widths, ranges, and access ports—the register banks can be easily synthesized. Moreover, a data and program memory resource are declared—both 32-bits wide and with just one allowed access per cycle. Since various memory types are known and are generally very technology-dependent, yet cannot be further specified in the LISA model, wrappers are generated with the appropriate number of access ports. Before synthesis, the wrappers need to be filled manually with code for the respective technology. The resources labeled as *PORT* are accessible from outside the model and can be attached to a testbench—in the ICORE, the *RESET* and the *STATE_BUS*.

Besides the processor resources such as memories, ports, and registers, also pipelines and pipeline registers are declared. The ICORE architecture contains a four-stage instruction pipeline consisting of the stage instructions fetch (FI), instruction decode (ID), instruction execution (EX), and write back to registers (WB). In between those pipeline stages, pipeline registers are located, which forward information about the instruction such as instruction opcode, operand registers, etc. The declared pipeline registers are multiple instanced between each stage and are completely generated from the LISA model. For the pipeline and the stages, entities are created, which are in a subsequent phase of the HDL generator run filled with code for functional units, instruction decoder, pipeline controller, etc.

2) *Grouping Operations to Functional Units*: As the LISA language describes the target architecture's behavior and timing on the granularity of hardware operations, however, the synthesis requires the grouping of hardware operations to

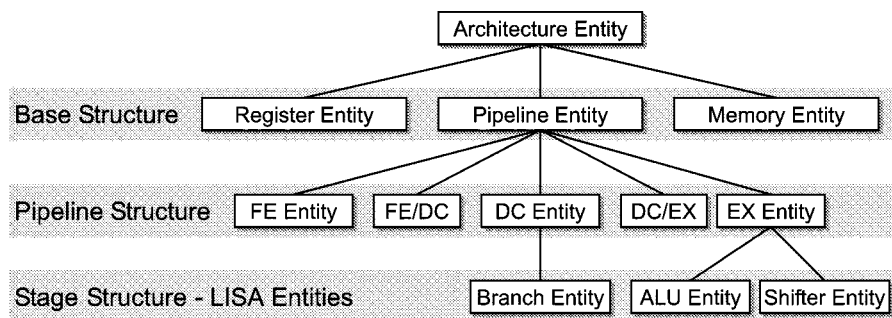


Fig. 15. Entity hierarchy in generated HDL model.

functional units that can then be filled with hand-optimized HDL code for the data path, a well-known construct from the VHDL language was adopted for this purpose: the ENTITY (see *microarchitecture model* in Section III-B). Using the entity to group hardware operations to a functional unit is not only an essential information for the HDL code generator, but also for retargeting the HLL C compiler, which requires information about the availability of hardware resources to schedule instructions.

As indicated in Section VI-A, the HDL code derived from the LISA resource section already comprises a pipeline entity including further entities for each pipeline stage and the respective pipeline registers. The entities defined in the LISA model are now part of the respective pipeline stages, as shown in Fig. 15. Here, a *Branch* entity is placed into the entity of the *Decode* stage. Moreover, the EX stage contains an arithmetic logic unit and a *Shifter* entity. As it is possible in LISA to assign hardware operations to pipeline stages, this information is sufficient to locate the functional units within the pipeline to which they are assigned.

As already pointed out, the entities of the functional units are *wrappers*, which need to be filled with HDL code by hand. Nevertheless, in Section VI-B, it will be shown that by far, the largest part of the target architecture can be generated automatically from a LISA model.

3) *Generation of the Instruction Decoder*: The generated HDL decoder is derived from information in the LISA model on the coding of instructions (see *instruction-set model* in Section III-B). Depending on the structuring of the LISA architecture description, decoder processes are generated in several pipeline stages. The specified signal paths within the target architecture can be divided into data signals and control signals. The control signals are a straight forward derivation of the operation activation tree, which is part of the LISA timing model (see *timing model* in Section III-B). The data signals are explicitly modeled by the designer by writing values into pipeline registers and implicitly fixed by the declaration of used resources in the behavior sections of LISA operations.

B. Implementation Results

The ICORE, which was used in our case study, is a low-power ASIP for digital video broadcasting terrestrial acquisition and tracking algorithms. It has been developed in cooperation with Infineon Technologies. The primary tasks of this architecture are the fast-Fourier-transformation window position, sampling-

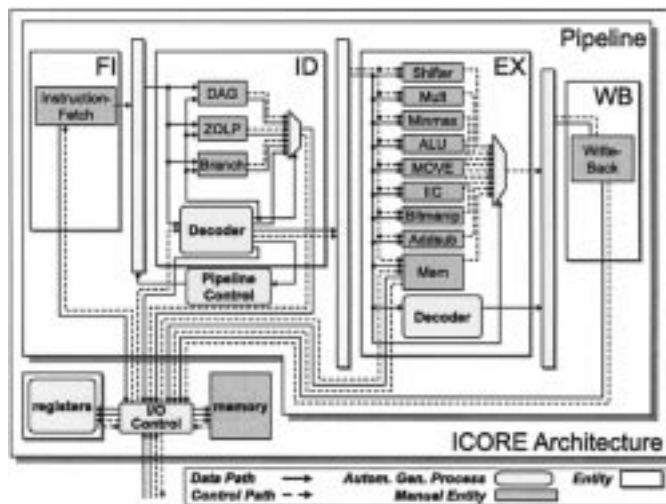


Fig. 16. Complete generated HDL model.

clock synchronization for interpolation/decimation, and carrier frequency offset estimation. In a previous project, this architecture was completely designed by hand using semicustom design. Therefore, a large amount of effort was spent in optimizing the architecture toward extremely low-power consumption while keeping up the clock frequency at 120 MHz. At that time, a LISA model was already realized for architecture exploration purposes and for verifying the model against the handwritten HDL implementation.

Except for the data path within functional units, the HDL code of the architecture has been generated completely. Fig. 16 shows the composition of the model.

The dark boxes have been filled manually with HDL code, whereas the light boxes and interconnects are the result of the generation process.

1) *Comparison of Development Time*: The LISA model of the ICORE as well as the original handwritten HDL model of the ICORE architecture have been developed by one designer. The initial manual realization of the HDL model (without the time needed for architecture exploration) took approximately three months. As already indicated, a LISA model was built in this first realization of the ICORE for architecture exploration and verification purposes. It took the designer approximately one month to learn the LISA language and to create a cycle-accurate LISA model.

After completion of the HDL generator, it took another two days to refine the LISA model to register transfer level accuracy.

The handwritten functional units (data path), which were added manually to the generated HDL model, could be completed in less than a week.

This comparison clearly indicates that the time expensive work in realizing the HDL model was to create structure, controller, and decoder of the architecture. In addition, a major decrease of total architecture design time can be seen, as the LISA model results from the design exploration phase.

2) *Gate-Level Synthesis*: To verify the feasibility of generating automatically HDL code from LISA architecture descriptions in terms of power-consumption, clock speed, and chip area, a gate-level synthesis was carried out. The model has not been changed (i.e., manually optimized) to enhance the results.

a) *Timing and size comparison*: The results of the gate-level synthesis affecting timing and area optimization were compared to the handwritten ICORE model, which comprised the same architectural features. Moreover, the same synthesis scripts were used for both models. It shall be emphasized that the performance values are nearly the same for both models. Furthermore, it is interesting that the same critical paths were found in both the handwritten and the generated model. The critical paths occur exclusively in the data path, which confirms the presumption that the data path is the most critical part of the architecture and, thus, should not be generated automatically from an abstract processor model.

b) *Critical path*: The synthesis has been performed with a clock of 8 ns, which equals a frequency of 125 MHz. The critical path, starting from the pipeline register to the shifter unit and multiplexer to the next pipeline register, violates this timing constraints by 0.36 ns. This matches the handwritten ICORE model, which has been improved from this point of state manually at gate level.

The longest combinatoric path of the ID stage runs through the decoder and the data address generator entity and counts 3.7 ns. Therefore, the generated decoder does not affect the critical path in any way.

c) *Area*: The synthesized area has been a minor criteria, due to the fact that the constrains for the handwritten ICORE model are not area sensitive. The total area of the generated ICORE model is 59 009 gates. The combinational area takes 57% of the total area. The handwritten ICORE model takes a total area of 58 473 gates.

The most complex part of the generated ICORE is the decoder. The area of the automatically generated decoder in the ID stage is 4693 gates, whereas the area of the handwritten equivalent is 5500 gates. This result must be considered carefully as the control logic varies in some implemented features, e.g., the handwritten decoder and program flow controller support an *idle* and *suspended* state of the core.

d) *Power consumption comparison*: Fig. 17 shows the comparison of power consumption of the handwritten versus the generated ICORE realization.

The handwritten model consumes 12.64 mW, whereas the implementation generated from a LISA model consumes 14.51 mW. The reason for the slightly worse numbers in power consumption of the generated model versus the handwritten is due to the early version of the LISA HDL generator, which in its current state allows access to all registers and memories within the

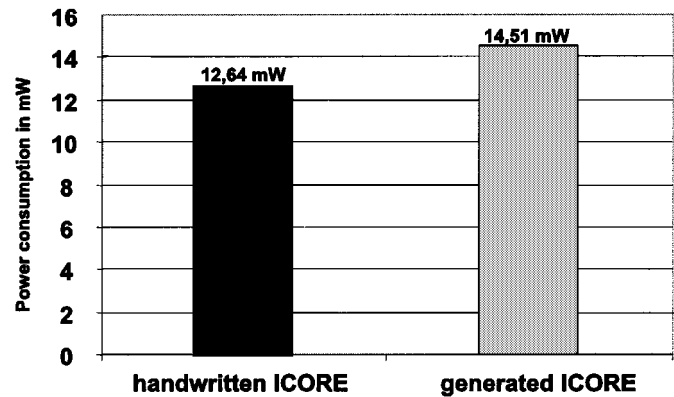


Fig. 17. Power consumption of different ICORE realization.

model via the test interface. Without this unnecessary overhead, the same results as for the hand-optimized model are achievable.

To summarize, it could be shown in this section that it is feasible to generate efficient HDL code from architecture descriptions in the LISA language.

VII. TOOLS FOR APPLICATION DEVELOPMENT

The LPDP application software development tool suite includes HLL C compiler, assembler, linker, and simulator as well as a graphical debugger frontend. Providing these tools, a complete software development environment is available which ranges from the C/assembly source file up to simulation within a comfortable graphical debugger frontend.

The tools are an enhanced version of those tools used for architecture exploration. The enhancements for the software simulate the ability to graphically visualize the debugging process of the application under test. The LISA debugger frontend *ldb* is a generic graphical user interface for the generated LISA simulator (see Fig. 18). It visualizes the internal state of the simulation process. Both the C source code and the disassembly of the application as well as all configured memories and (pipeline) registers are displayed. All contents can be changed in the frontend at runtime of the application. The progress of the simulator can be controlled by stepping and running through the application and setting breakpoints.

The code generation tools (assembler and linker) are enhanced in functionality as well. The assembler supports more than 30 common assembler directives, labels, and symbols, named user sections, generation of source listing, and symbol table and provides detailed error report and debugging facilities, whereas the linker is driven by a powerful linker command file with the ability to link sections into different address spaces and paging support and the possibility to define user-specific memory models.

A. Examined Architectures

To examine the quality of the generated software development tools, four different architectures have been considered. The architectures were carefully chosen to cover a broad range of architectural characteristics and are widely used in the field of digital signal processing and microcontrollers (μ C). Moreover,

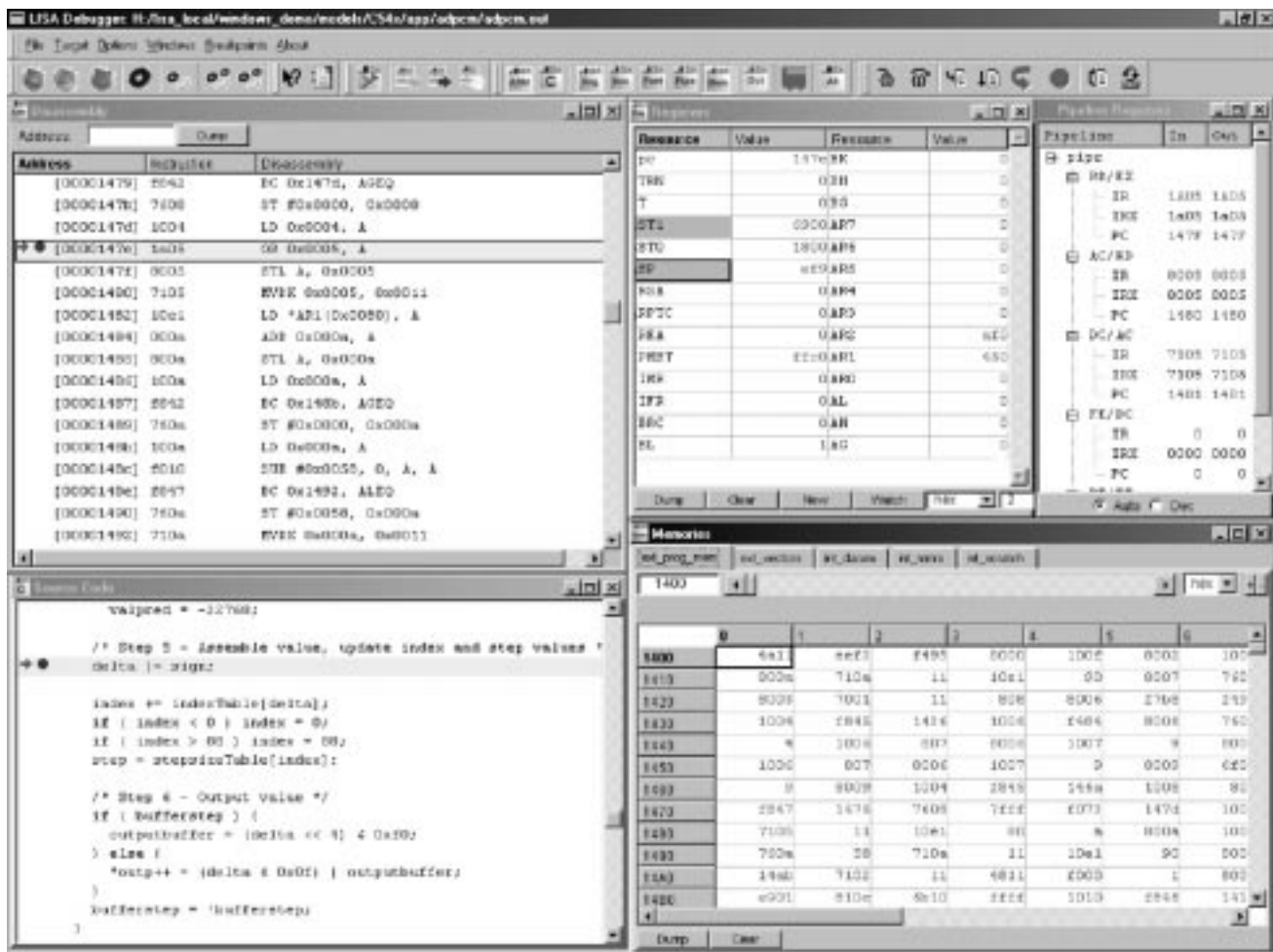


Fig. 18. Graphical debugger frontend.

the abstraction level of the models ranges from phase-accuracy (TMS320C62x) to instruction-set accuracy (ARM7).

- 1) *ARM7*: The ARM7 core is a 32-bit μ C of Advanced RISC Machines Ltd. [39]. The realization of a LISA model of the ARM7 μ C at instruction-set accuracy took approximately two weeks.
- 2) *ADSP2101*: The Analog Devices ADSP2101 is a 16-bit fixed-point DSP with 20-bit instruction-word width [40]. The realization of the LISA model of the ADSP2101 at cycle accuracy took approximately three weeks.
- 3) *TMS320C54x*: The TI TMS320C54x is a high-performance 16-bit fixed-point DSP with a six-stage instruction pipeline [37]. The realization of the model at cycle accuracy (including pipeline behavior) took approximately eight weeks.
- 4) *TMS320C62x*: The TI TMS320C62x is a general-purpose fixed-point DSP based on a VLIW architecture containing an 11-stage pipeline [36]. The realization of the model at phase-accuracy (including pipeline behavior) took approximately six weeks.

These architectures were modeled on the respective abstraction level with LISA and software development tools were generated successfully. The speed of the generated tools was then compared with the tools shipped by the respective tools of the architecture vendor. Of course, the LISA tools are working on

the same level of accuracy as the vendor tools. The vendor tools are using exclusively the interpretive simulation technique.

B. Efficiency of the Generated Tools

Measurements took place on an AMD Athlon system with a clock frequency of 800 MHz. The system is equipped with 256 MB of random access memory and is part of the networking system. It runs under the operating system Linux, kernel version 2.2.14. Tool compilation was performed with GNU GCC, version 2.92.

The generation of the complete tool suite (HLL C compiler, simulator, assembler, linker, and debugger frontend) takes, depending on the complexity of the considered model, between 12 s (ARM7 μ C instruction-set accurate) and 67 s (C6x DSP phase-accurate). Due to the early stage in research on the re-targetable compiler (see Section VIII), no results on code quality are presented.

1) *Performance of the Simulator*: Figs. 19– 22 show the speed of the generated simulators in instructions per second and cycles per second, respectively. Simulation speed was quantified by running an application on the respective simulator and counting the number of processed instructions per cycle.

The set of simulated applications on the architectures comprises a simple 20-tap finite-impulse-response filter, an adaptive differential pulse code modulation G.721 coder/decoder, and a

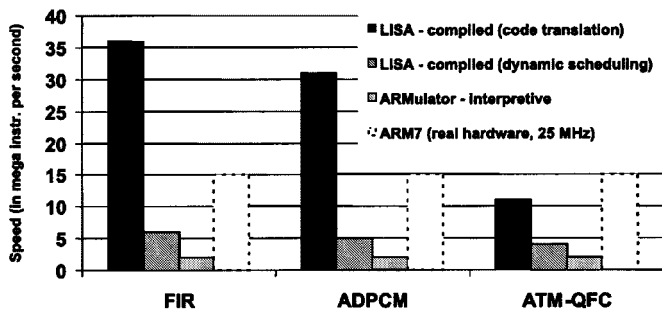
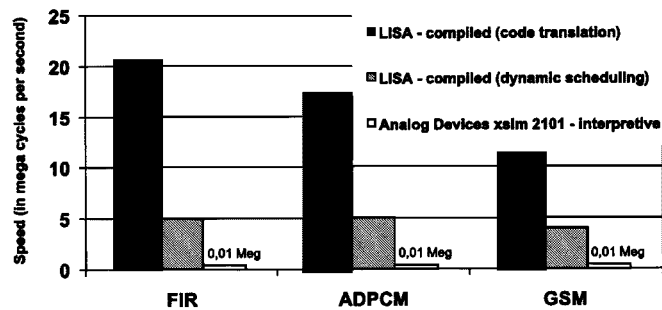
Fig. 19. Spend of the ARM7 at μC instruction accuracy.

Fig. 20. Spend of the ADSP2101 DSP at cycle accuracy.

global system for mobile communications speech codec. For the ARM7, an asynchronous transfer mode (ATM) quantum flow control protocol application was additionally run, which is responsible for flow control and configuration in an ATM port-processor chip.

As expected, the compiled simulation technique applied by the generated LISA simulators outperforms the vendor simulators by one to two orders of magnitude.

As both the ARM7 and ADSP2101 LISA model contain no instruction pipeline, two different flavors of compiled simulation are applied in the benchmarks—instruction-based code translation and dynamic scheduling (see Section V-B4). It shows that the highest possible degree of simulation compilation offers an additional speedup of a factor of 2–7 compared to dynamically scheduled compiled simulation. As explained in Section V-B4, the speedup decreases with bigger applications due to cache misses on the simulating host. It is interesting to see that, considering an ARM7 μC running at a frequency of 25 MHz, the software simulator running at 31 MIPS outperforms even the real hardware. This makes application development suitable before the actual silicon is at hand.

The LISA model of the C54x DSP is cycle accurate and contains an instruction pipeline. Therefore, compiled simulation with static scheduling is applied (see Section V-B3). This pays off with an additional speedup of a factor of 5, compared to a dynamically scheduled compiled simulator.

Due to the superscalar instruction dispatching mechanism used in the C62x architecture, which is highly runtime dependent, the LISA simulator for the C62x DSP uses only compiled simulation with dynamic scheduling. However, the dynamic scheduled compiled simulator still offers a significant speedup of a factor of 65 compared to the native TI simulator.

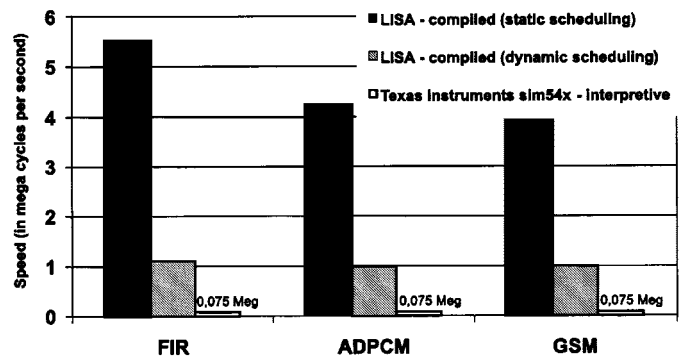


Fig. 21. Spend of the C54x DSP at cycle accuracy.

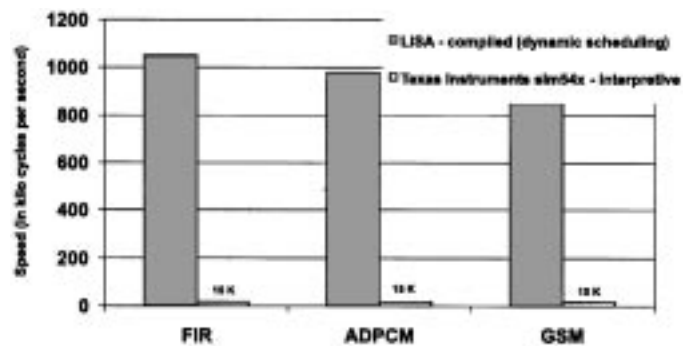


Fig. 22. Spend of the C6x DSP at cycle accuracy.

2) *Performance of Assembler and Linker*: The generated assembler and linker are not as time critical as the simulator is. It shall be mentioned though that the performance (i.e., the number of assembled/linked instructions per second) of the automatically generated tools is comparable to that of the vendor tools.

VIII. REQUIREMENTS AND LIMITATIONS

In this section, the requirements and current limitations of different aspects of the processor design using the LISA language are discussed. These affect the modeling capabilities of the language itself as well as the generated tools.

A. LISA Language

Common to all models described in LISA is the underlying zero-delay model. This means that all transitions are provided correctly at each control step. Control steps may be clock phases, clock cycles, instruction cycles, or even higher levels. Events between these control steps are not regarded. However, this property meets requirements of current cosimulation environments [41]–[43] on processor simulators to be used for hardware–software codesign [44], [45]. Besides, the LISA language currently contains no formalism to describe memory hierarchies such as multilevel caches. However, existing C/C++ models of memory hierarchies can easily be integrated into the LISA architecture model.

B. HLL C Compiler

Due to the early stage of research, no further details on the re-targetable compiler are presented within the scope of this paper.

At the current status, the quality of the generated code is only fair. However, it is evident that the proposed new ASIP design methodology can only be carried out efficiently at the presence of an efficient retargetable compiler. In the case study presented in Section VI, major parts of the application were realized in assembly code.

C. HDL Generator

As LISA allows modeling the architecture using a combination of both LISA language elements and pure C/C++ code, certain coding guidelines need to be obeyed in order to generate synthesizable HDL code of the target architecture. First, only the LISA language elements are considered—thus, the usage of C code in the model needs to be limited to the description of the data path, which is not taken into account for HDL code generation anyway. Second, architectural properties, which can be modeled in LISA, but are not synthesizable, include pipelined functional units and multiple instruction word decoders.

IX. CONCLUSION AND FUTURE WORK

In this paper, we presented the LPDP—a novel framework for the design of ASIPs. The LPDP platform helps the architecture designer in different domains: architecture exploration, implementation, application software design, and system integration/verification.

In a case study, it was shown that an ASIP, the ICORE architecture, was completely realized using this novel design methodology—from exploration to implementation. The implementation results concerning maximum frequency, area, and power consumption were comparable to those of the hand-optimized version of the same architecture realized in a previous project.

Moreover, the quality of the generated software development tools was compared to those of the semiconductor vendors. LISA models were realized and tools successfully generated for the ARM7 μ C, the Analog Devices ADSP2101, the TI C62x, and the TI C54x on instruction set, cycle, and phase accuracy, respectively. Due to the usage of the compiled simulation principle, the generated simulators run by one to two orders of magnitude faster than the vendor simulators. In addition, the generated assembler and linker can compete well in speed with the vendor tools.

Our future work will focus on modeling further real-world processor architectures and improving the quality of our retargetable C compiler. In addition, formal ways to model memory hierarchies will be addressed. For the HDL generator, data path synthesis will be examined in context of the SystemC modeling language.

REFERENCES

- [1] M. Birnbaum and H. Sachs, "How VSIA answers the SOC dilemma," *IEEE Comput.*, vol. 32, pp. 42–50, June 1999.
- [2] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, "LISA—Machine description language for cycle-accurate models of programmable DSP architectures," in *Proc. Design Automation Conf.*, New Orleans, LA, June 1999, pp. 933–938.
- [3] V. Zivojnović, S. Pees, and H. Meyr, "LISA—Machine description language and generic machine model for HW/SW co-design," in *Proc. IEEE Workshop VLSI Signal Processing*, San Francisco, CA, Oct. 1996, pp. 127–136.

- [4] K. Olukotun, M. Heinrich, and D. Ofelt, "Digital system simulation: Methodologies and examples," in *Proc. Design Automation Conf.*, June 1998, pp. 658–663.
- [5] J. Rowson, "Hardware/software co-simulation," in *Proc. Design Automation Conf.*, June 1994, pp. 439–440.
- [6] R. Stallman, *Using and Porting the GNU Compiler Collection*, gcc-2.95 ed. Boston, MA: Free Software Foundation, 1999.
- [7] G. Araujo, A. Sudarsanam, and S. Malik, "Instruction set design and optimization for address computation in DSP architectures," in *Proc. Int. Symp. System Synthesis*, Nov. 1997, pp. 31–37.
- [8] C. Liem *et al.*, "Industrial experience using rule-driven retargetable code generation for multimedia applications," in *Proc. Int. Symposium System Synthesis*, Sept. 1995, pp. 60–68.
- [9] D. Engler, "VCODE: A retargetable, extensible, very fast dynamic code generation system," in *Proc. Int. Conf. Programming Language Design and Implementation*, May 1996, pp. 160–170.
- [10] D. Bradlee, R. Henry, and S. Eggers, "The Marion system for retargetable instruction scheduling," in *Proc. ACM SIGPLAN'91 Conf. Programming Language Design and Implementation*, Toronto, ON, Canada, June 1991, pp. 229–240.
- [11] B. Rau, "VLIW compilation driven by a machine description database," in *Proc. 2nd Code Generation Workshop*, Leuven, Belgium, Mar. 1996.
- [12] M. Freericks, "The nML machine description formalism," Fachbereich Informatik, Tech. Univ. Berlin, Berlin, Germany, Tech. Rep. 1991/15, 1991.
- [13] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proc. Eur. Design and Test Conf.*, Paris, France, Mar. 1995, pp. 503–507.
- [14] M. Hartoog *et al.*, "Generation of software tools from processor descriptions for hardware/software codesign," in *Proc. Design Automation Conf.*, June 1997, pp. 303–306.
- [15] W. Geurts *et al.*, "Design of DSP systems with Chess/Checkers," in *Proc. 2nd Int. Workshop on Code Generation for Embedded Processors*, Leuven, Belgium, Mar. 1996.
- [16] J. Van Praet, "A graph based processor model for retargetable code generation," in *Proc. Eur. Design and Test Conf.*, Mar. 1996, pp. 102–107.
- [17] V. Rajesh and R. Moona, "Processor modeling for hardware software codesign," in *Proc. Int. Conf. VLSI Design*, Goa, India, Jan. 1999, pp. 81–87.
- [18] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An instruction set description language for retargetability," in *Proc. Design Automation Conf.*, June 1997, pp. 299–302.
- [19] A. Halambi, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," in *Proc. Conf. Design, Automation, and Test Eur.*, Mar. 1999, pp. 100–104.
- [20] P. Paulin, "Design automation challenges for application-specific architecture platforms," in *Proc. SCOPES 2001 Workshop Software and Compilers for Embedded Systems*, Mar. 2001.
- [21] The COSY compilation system, Associated Compiler Experts (ACE). (2001). [Online]. Available: <http://www.ace.nl/products/cosy.html>
- [22] T. Morimoto, K. Saito, H. Nakamura, T. Boku, and K. Nakazawa, "Advanced processor design using hardware description language AIDL," in *Proc. Asia South Pacific Design Automation Conf.*, Mar. 1997, pp. 387–390.
- [23] I. Huang, B. Holmer, and A. Despain, "ASIA: Automatic synthesis of instruction-set architectures," in *Proc. SASIMI Workshop*, Oct. 1993.
- [24] M. Gschwind, "Instruction set selection for ASIP design," in *Proc. Int. Workshop Hardware/Software Codesign*, May 1999, pp. 7–9.
- [25] S. Kobayashi *et al.*, "Compiler generation in PEAS-III: An ASIP development system," in *Proc. SCOPES 2001 Workshop Software and Compilers for Embedded Systems*, Mar. 2001.
- [26] C.-M. E. A. Kyung, "Metacore: An application specific DSP development system," in *Proc. Design Automation Conf.*, June 1998, pp. 173–184.
- [27] M. Barbacci, "Instruction set processor specifications (ISPS): The notation and its application," *IEEE Trans. Comput.*, vol. C-30, pp. 24–40, Jan. 1981.
- [28] R. Gonzales, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, pp. 60–70, Mar. 2000.
- [29] COSSAP, Synopsys, Inc. [Online]. Available: <http://www.synopsys.com>
- [30] OPNET [Online]. Available: <http://www.opnet.com>
- [31] LISA home page, Inst. Integrated Signal Processing Systems, The Aachen Univ. Technol. (2001). [Online]. Available: <http://www.iss.rwth-aachen.de/lisa>
- [32] T. Gloekler, S. Bitterlich, and H. Meyr, "Increasing the power efficiency of application specific instruction set processors using datapath optimization," in *Proc. IEEE Workshop Signal Processing Systems*, Lafayette, LA, Oct. 2001, pp. 563–570.

- [33] DesignWare, Synopsys, Inc. (1999). [Online]. Available: <http://www.synopsys.com/products/designware/designware.html>
- [34] A. Hoffmann, A. Nohl, G. Braun, and H. Meyr, "Generating production quality software development tools using a machine description language," in *Proc. Conf. Design, Automation, and Test Eur.*, Mar. 2001, pp. 674–679.
- [35] S. Pees, A. Hoffmann, and H. Meyr, "Retargeting of compiled simulators for digital signal processors using a machine description language," in *Proc. Conf. Design, Automation, and Test Eur.*, Mar. 2000, pp. 933–938.
- [36] *TMS320C62x/C67x CPU and Instruction Set Reference Guide*, Texas Instruments Incorporated, Dallas, TX, Mar. 1998.
- [37] *TMS320C54x CPU and Instruction Set Reference Guide*, Texas Instruments Incorporated, Dallas, TX, Oct. 1996.
- [38] R. Sites *et al.*, "Binary translation," *Commun. ACM*, vol. 36, pp. 69–81, Feb. 1993.
- [39] *ARM7 Data Sheet*, Advanced Risc Machines Ltd., Cambridge, U.K., Dec. 1994.
- [40] *ADSP2101 Users Manual*, Analog Devices, Inc, Norwood, MA, Sept. 1993.
- [41] EagleI, Synopsys, Inc. (1999). [Online]. Available: <http://www.synopsys.com/products/hsw>
- [42] Cierto, Cadence Design Systems, Inc. (1999). [Online]. Available: <http://www.cadence.com/technology/hsw>
- [43] Seamless, Mentor Graphics. (1999). [Online]. Available: <http://www.mentor.com/seamless>
- [44] L. Guerra *et al.*, "Cycle and phase accurate DSP modeling and integration for HW/SW co-verification," in *Proc. Design Automation Conf.*, June 1999, pp. 964–969.
- [45] R. Earnshaw, L. Smith, and K. Welton, "Challenges in cross-development," *IEEE Micro*, vol. 17, pp. 28–36, July/Aug. 1997.



Andreas Hoffmann (M'01) received the Dipl. Ing. degree in electrical engineering from Ruhr-University-Bochum, Bochum, Germany, in 1997.

In 1997, he joined the Institute for Integrated Signal Processing Systems, Aachen University of Technology, Aachen, Germany, as a Research Assistant, where he has been working on retargetable software development tools based on processor descriptions in the LISA language. His current research interests include novel processor design techniques as well as retargetable simulation and

code generation for embedded processors.



Tim Kogel received the Dipl. Ing. degree in electrical engineering from Aachen University of Technology (RWTH), Aachen, Germany in 1999. He is currently working toward the Ph.D. degree in electrical engineering at the same university.

His current research interests include system-level design methodologies, hardware/software cosimulation, and architectures for networking applications.



Achim Nohl received the Dipl. Ing. degree in electrical engineering from Aachen University of Technology (RWTH), Aachen, Germany in 2000. He is currently working toward the Ph.D. degree in electrical engineering at the same university.

His current research interests include hybrid simulation techniques and code generation for embedded processors.



Gunnar Braun received the Dipl. Ing. degree in electrical engineering from Aachen University of Technology (RWTH), Aachen, Germany in 2000. He is currently working toward the Ph.D. degree in electrical engineering at the same university.

His current research interests include retargetable simulation and code generation as well as design methodology for embedded processors.



Oliver Schliebusch received the Dipl. Ing. degree in electrical engineering from Aachen University of Technology (RWTH), Aachen, Germany in 2001. He is currently working toward the Ph.D. degree in electrical engineering at the same university.

His current research interests include the design of processor architectures and the HDL code generation from LISA processor models.



Oliver Wahlen received the Dipl. Ing. degree in electrical engineering from Aachen University of Technology (RWTH), Aachen, Germany, in 1998. He is currently working toward the Ph.D. degree in electrical engineering at the same university.

His current research interests include compiler/architecture codesign with the main focus on retargetable code generation for ASIP and DSP architectures.



Andreas Wieferink received the Dipl. Ing. degree in electrical engineering from Aachen University of Technology (RWTH), Aachen, Germany, in 2000. He is currently working toward the Ph.D. degree in electrical engineering at the same university.

His current research interests include hardware–software cosimulation and processor modeling.



Heinrich Meyr (M'75–SM'83–F'86) received the M.S. and Ph.D. degrees from the Swiss Federal Institute of Technology (ETH) Zurich, Zurich, Switzerland, in 1967 and 1973, respectively.

He spent over 12 years in various research and management positions in industry before accepting a professorship in electrical engineering at the Aachen University of Technology (RWTH), Aachen, Germany, in 1977, where he currently heads an institute involved in the analysis and design of complex signal processing systems for communication

applications. He is a Cofounder CADIS GmbH (acquired by Synopsys, Inc., Mountain View, CA, in 1993), which commercialized the tool suite COSSAP, extensively used in industry worldwide. He is a Member of the Board of Directors of two companies in the communications industry. He has authored or coauthored numerous papers for the IEEE and two books, *Synchronization in Digital Communications* (New York: Wiley, 1990) and *Digital Communication Receivers, Synchronization, Channel Estimation, and Signal Processing* (New York: Wiley, 1997), and holds many patents. He has worked extensively in the areas of communication theory, synchronization, and digital signal processing for the last 30 years and his research has been applied to the design of many industrial products.

Dr. Meyr served as Vice President for International Affairs of the IEEE Communications Society.