Architecture Description Languages for Systems-on-Chip Design †

Hiroyuki Tomiyama ‡
tomiyama@ics.uci.edu

Ashok Halambi
ahalambi@ics.uci.edu

Peter Grun
pgrun@ics.uci.edu

Nikil Dutt
dutt@ics.uci.edu

Alex Nicolau
nicolau@ics.uci.edu

Center for Embedded Computer Systems
University of California, Irvine, CA 92697–3425, USA
http://www.cecs.uci.edu/~aces

Abstract

Advances in semiconductor technology coupled with the increasing availability of soft and hard IP libraries enable embedded system designers to develop Systems-on-Chip (SOCs) containing highly customized processors and memories for their specific applications. However, there is a strong demand for a methodology and tools that support efficient Design Space Exploration (DSE) of SOC architectures. Architecture Description Language (ADL)-based SOC codesign is a promising approach to efficient DSE of SOC architectures. ADLs are languages designed for specification of SOC architecture templates, and are used to perform early validation of SOC architectures, as well as to automatically generate software toolkits required to complete the integrated, and concurrent hardware and software design of the SOCs. In this paper we survey recent efforts in the use of ADLs. We conclude with a discussion of several major challenges facing ADL-based codesign of future SOCs.

1. Introduction

Traditionally, embedded systems are implemented as a system board on which several chips (e.g., processors, memories, ASICs, and I/O interface circuits) are connected to each other. In most cases of embedded system design, a designer selects processors, memories, and I/O interface circuits from standard product libraries, and designs ASICs. Customization of processors and memories is hardly allowed because of the high hardware cost and long time-to-market. However, the advent of System-on-Chip (SOC) technology which enables integration of processors, memories, ASICs, and I/O interface circuits on a single chip, coupled with the availability of soft and hard IP libraries, opens up the possibility for realizing a variety of new embedded system architectures that can be customized for a particular application. Specifically, customization of the processor architecture and the memory system which permits efficient implementation of SOCs in terms of performance, chip area and power consumption is enabled. The future trend is towards SOCs that contain heterogeneous multi-processors (including general purpose processors, ASIPs, and DSPs), and hierarchical or distributed memories (including SRAMs, DRAMs, and flash memories), which have never been possible in the past. Figure 1 shows an example of such an SOC.

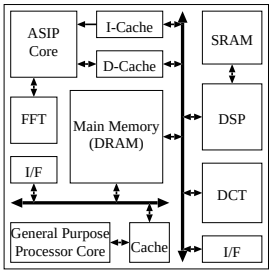

Figure 1. An example SOC

Since each application has its own functionality and design constraints such as performance, chip area, and power consumption, the optimal SOC architecture can be quite different depending on the application. In order to find the optimal SOC architecture, a large range of architecture candidates have to be explored in an early phase of SOC design. In this phase, the detailed structure of each block (processor, memory, ASIC, etc.)

† This work is supported in part by grants from NSF (MIP-9708067 and ONR (N00014-93-1-1348).

‡ Research at UCI is supported by JSPS postdoctoral fellowships for research abroad.

has not been fixed, and will be fixed in the later (therefore, low level) design phase. When exploring SOC architectures, the architecture template (not the detailed structure) of the SOC must be specified at a system level, and its efficiency in terms of performance, chip area, and power, must be estimated. When a designer customizes the processor architecture and the memory system, the designer requires a software toolkit including at least a compiler and a simulator for the processor-memory system. However, manual development of the toolkit is generally a time-consuming bottleneck in SOC design. Therefore, an ideal environment for SOC design should support automatic generation of a software toolkit from the specification of the processor and the memory system.

Although existing HDLs are well suited to describe the structure of blocks in SOCs, they do not provide sufficient ability to easily specify SOC architecture templates because of their low levels of abstraction. Furthermore, automatic generation of a production-quality software toolkit from processor and memory description in HDLs is hardly possible because of the lack of information about the instruction set. Therefore, there is a strong need for new specification languages (for SOC architecture templates) that support automatic software toolkit generation.

Architecture Description Languages (ADLs) are languages designed for specification of architectural templates of SOCs, and ADL-based SOC codesign is a promising approach to efficient Design Space Exploration (DSE) of SOC architectures. In this paper we discuss ADLs and an ADL-based codesign methodology which allows efficient DSE and optimization of SOC architectures. Section 2 describes requirements of ideal ADLs and an ADL-based codesign methodology. Section 3 surveys existing ADLs, and Section 4 summarizes future challenges to be addressed. Section 5 concludes this paper.

2. Architecture Description Languages

2.1. What is an ADL?

An ADL is a language designed to specify architecture templates of SOCs. An architecture template includes (a) blocks/components which reside on the SOC, (b) how they are connected or interact, and (c) functionality of each block/component. Detailed structure of each block (processor, memory, ASIC, etc.) is not described in ADLs because it is determined in the later design phase. Although several ADLs have been proposed recently as we will see in Section 3, no ADL has been standardized. ADLs are rather new research topics and should be further studied in order to realize efficient DSE of SOC architectures.

Next, we discuss the requirements of ideal ADLs. First of all, ideal ADLs must have an ability to specify architecture templates of SOCs (i.e., blocks/components which reside on the SOC, how they are connected or interact, and functionality of each block/component) in a natural and easy manner. Natural and easy specification shrinks specification time and also improves the readability of the specification. Blocks in SOCs may include processors, memories, ASICs, I/O interface circuits, and interconnections (e.g., buses). Especially, an ADL must be able to capture wide classes of processor architectures ranging from simple RISC processors to VLIWs, advanced superscalar processors, ASIPs and DSPs with irregular datapath structures. Also, memory hierarchies must be modeled which may be composed of various memory types such as DRAMs, flash memories, frame buffers, and so on.

ADLs must be formal languages. From the underlying formalism, an SOC designer can formally verify or validate several properties which the specified SOC must satisfy in an early phase of SOC design.

ADLs must support automatic generation of a production-quality software toolkit including at least an Instruction-Level Parallelizing (ILP[1]) compiler and simulators. Information on conflicts between instructions is necessary for both ILP compilers and run-time schedulers to detect dependencies, hazards, and so on. This conflict information can be specified in many ways and the ADL should support a natural specification which eases the cumbersome approach of detailed conflict specification. The simulators are used for various purposes such as estimation of performance and power, analysis of resource utilization, functional validation of hardware, debugging of software, hardware/software co-simulation, and so on. A variety of simulators need to be generated to support both fast DSE (e.g., function- and instruction-level simulators) and detailed design (e.g., cycle-, phase-, and pin-accurate simulators).

2.2. ADL-Based SOC Codesign

The dotted box in Figure 2 shows a contemporary hardware/software codesign flow for the design of traditional embedded systems consisting of a programmable processor, application specific integrate circuits (ASICs), memories, I/O interfaces, etc., which are connected on a system board. This contemporary codesign flow starts from specifying an application in a system design language. The application is generally composed of several tasks. Next, each task is assigned to either software (the processor) or hardware (an ASIC) such that design constraints (e.g., performance, power consumption, chip area, etc.) are satisfied. This design process is called hardware/software

[1] In the rest of this paper, the term ILP also stands for Instruction Level Parallelism, depending on the context.

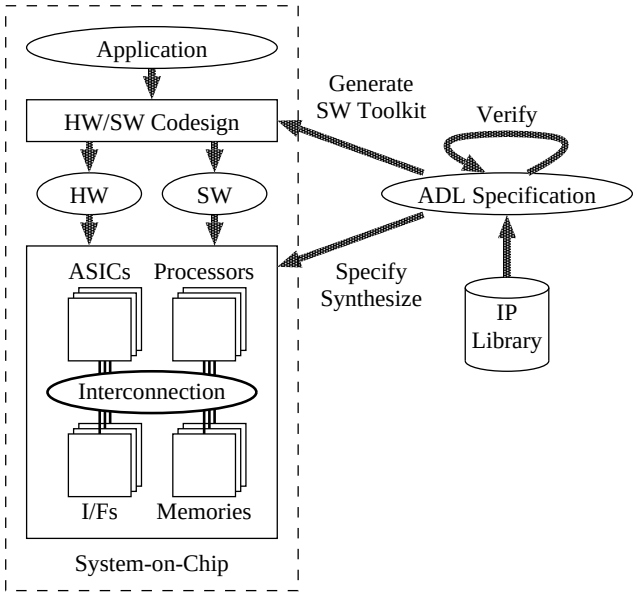

Figure 2. ADL-based codesign flow for SOCs

partitioning. After partitioning, tasks assigned to software are translated into programs in high-level languages (HLLs) such as C/C++, and then compiled into object code which will finally be stored in memories. Tasks assigned to hardware are translated into HDL descriptions, and then synthesized into ASICs. In traditional codesign systems, the target architecture template is pre-defined. Specifically, a processor is fixed or can be selected from a library of pre-designed processors, but customization of the processor architecture is not allowed. Even in codesign systems allowing customization of the processor, the fundamental architecture can rarely be changed.

Advances in SOC technology which permit customization of processor architectures and memory systems make the contemporary codesign flow ineffective. Indeed, customization of processors and memories permits efficient implementation of SOCs in terms of performance, chip area and power consumption. However, the time-to-market tends to be long because of two main reasons: One is the extended design space explored by SOC designers and the other is the lack of software toolkits including at least a compiler and a simulator for the custom processor. The latter difficulty does not arise in the contemporary codesign flow in which pre-designed processors are employed, because software toolkits are usually supplied by the processor vendors. However, custom processor design requires development of software toolkit which is a time-consuming bottleneck in SOC design. Therefore, there is a strong demand for a new codesign methodology which permits efficient design space exploration (DSE) of SOC architectures with automatic software toolkit generation.

ADL-based SOC codesign is a promising approach for efficient DSE of SOC architectures as shown in Figure 2. An ADL-based SOC codesign flow starts from specifying an application in a system design language and an architecture template of the SOC in an ADL. The architecture template is verified or validated using a formal method. After verification, a software toolkit is automatically generated. After hardware/software partitioning, software tasks are compiled with the generated compiler, while hardware tasks are mapped to ASICs (which is defined in the ADL specification). Then, hardware/software co-simulation is performed using the generated simulator. At this point, performance, chip area, and power consumption are estimated. If all design constraints are satisfied, the design phase goes into more detailed abstraction level. Otherwise, the ADL specification is modified to further explore the design space of the SOC architecture.

The ADL-based codesign approach provides the ability to *customize* the SOC architecture template for a given (family of) application(s), while the traditional codesign does not. Especially, the following advantages of ADLs drive efficient DSE of SOC architectures: First, due to the high abstraction level of the ADLs, the architecture template can be specified in a natural and easy manner, resulting in a short time for specification. Second, the underlying formalism of the ADLs enables early and formal verification/validation of several properties which the SOC must satisfy. The other major advantages include automatic generation of a software toolkit from ADL descriptions.

3. Survey of ADLs

This section surveys a number of existing ADLs or ADL-like languages. Most existing ADLs are devoted to specification of processor architectures. Up to now, there exist no ADLs which satisfy all of the requirements listed in the previous section.

We classify existing ADLs into five categories based on their main objective: synthesis, compiler generation, simulator generation, validation, and other approaches.

3.1. Synthesis-Oriented ADLs

MIMOLA is an HDL rather than an ADL proposed at University of Dortmund [1]. MIMOLA is used to describe target machines in the MSSQ compiler [2] and the RECORD compiler [3], both of which have been developed at the same university. MSSQ translates programs written in a Pascal-like HLL into microcode for microprogrammable controllers, while RECORD

translates programs written in a DSP-specific programming language, called Data Flow Language (DFL), into machine code for the target DSP. In MIMOLA, the processor structure is modeled as a net-list at an RT level, and MIMOLA does not contain information on the instruction set or the assembly language. In MSSQ, programs are mapped to the components in the datapath. On the other hand, in RECORD, the instruction set is extracted from the MIMOLA description [4], and then, the extracted instruction set is used for compiler generation. MIMOLA does not support simulator generation, but MIMOLA descriptions can be simulated using the MSSB/U simulators. The largest advantage in this approach is that a single MIMOLA description can be used for synthesis, simulation, and test generation as well as code generation. However, MIMOLA does not support explicit description of processor pipelines or resource conflicts, resulting in poor code quality for some classes of VLIW or deeply pipelined processors. In [5], a processor modeling formalism has been proposed which enables automatic analysis of constraints on ILP.

COACH is a CAD system developed at Kyushu University for ASIP design supporting automatic generation of compilers and instruction level simulators [6]. The **UDL/I** HDL [7] is used for processor description in which processors are described at an RT level on a per-cycle basis. The instruction set is automatically extracted from the UDL/I description [8], and then it is used for generation of a compiler and a simulator. COACH assumes simple RISC processors and does not explicitly support processor pipelines or ILP. The simulator is at the instruction level and is not cycle-accurate. For concise simulation, the UDL/I simulator [9] can be used. Of course, the processor description is synthesizable with the UDL/I synthesis system [10]. The largest advantage of the COACH system is that COACH requires only a single description of the processor used for synthesis, concise simulation, instruction-level simulation, and compilation.

3.2. Compiler-Oriented ADLs

nML was proposed at TU Berlin [11], and used in the SIGH/SIM instruction-set simulator [12] and the CBC code generator [13]. nML is also used in some other institutes. At IMEC, nML is also used in the CHESS code generator [14] and the CHECKERS instruction set simulator. At Cadence Design Systems, nML is used to generate an instruction-set simulator, an assembler, and a disassembler [15]. The main target processors of nML are DSPs and ASIPs. In nML, the behavior (i.e., the instruction set) of a processor is described in the form of an attribute grammar, i.e., declarative description with attributes added onto the rules of the grammar. The information on the instruction set includes behavior (called action), assembly syntax, and object code image for each each instruction. nML explicitly supports several addressing modes and hierarchical description of the instruction set. However, it does not explicitly support multi-cycle functional units (multi-cycle FUs) or pipelined FUs since nML assumes that FUs do not have delay but storages do. Instead, transitory storages are introduced to model timing in the datapath [16]. Other weaknesses in nML include no explicit support of multi-word instructions. Resource conflicts between instructions are described implicitly by the grammar in a form of sets of legal combinations of operations. A good critique of nML is given in [15].

ISDL is an ADL proposed at MIT [17]. ISDL is mainly targeted towards VLIW processors. ISDL is used for automatic generation of compilers [18], assemblers, and cycle-accurate simulators [19]. Moreover, ISDL descriptions can be translated into synthesizable Verilog code. Similar to nML, ISDL mainly describes the instruction set of processors. The instruction set description includes action, assembly format, cost (reflecting execution cycles, code size, resource conflict, etc.), and timing (e.g., latency) for each instruction. The pipeline structure is not explicitly described. Constraints on ILP posed by resource conflicts and so on, are explicitly described in a form of a set of Boolean rules all of which must be satisfied for an instruction to be valid.

MDES is an ADL proposed at UIUC for DSE of high-performance processors used in the Trimaran system, supporting automatic compiler generation [20]. MDES captures both structure and behavior of target processors. Information is broken down into sections (such as format, resource usage, latency, operation, register, etc.), based on a high-level classification of the information being represented. MDES has good constructs for preprocessing which enable concise description. However, MDES allows only a restricted retargetability of the cycle-accurate simulator to the HPL-PD processor family. MDES permits description of memory systems, but limited to the traditional hierarchy, i.e., register files, caches, and main memory. MDES captures constraints on ILP with explicit description of reservation tables using a hierarchical description for compactness [21].

FlexWare is a CAD system for DSP or ASIP design [22]. The FlexWare system includes the CodeSyn code generator and the Insulin simulator. Both behavior and structure are captured in the target processor description. The machine description for CodeSyn consists of three components: instruction set, available resources (and their classification), and an interconnect graph representing the datapath structure. The instruction set description is a list of generic processor macro instructions to execute each target processor instruction. The simulator uses a VHDL model of

a generic parameterizable machine. The parameters include bit-width, number of registers and ALUs, etc. The application is translated from the user-defined target instruction set to the instruction set of this generic machine. Then, the code is executed on the generic machine. It is not clear how the resource conflicts between instructions are specified. Explicit specification of memory systems does not appear to be possible.

Valen-C is an embedded software programming language proposed at Kyushu University [23, 24]. Valen-C is an extended C language which supports explicit and exact bit-width for integer type declarations A retargetable compiler, called Valen-CC, which takes C or Valen-C programs as an input and generate the assembly code has also been developed. Although Valen-CC assumes simple RISC architectures, it has retargetability to some extent. The most interesting feature of Valen-CC is that the processor can have any datapath bit-width (e.g., 14 bits or 29 bits). The Valen-C system aims at optimization of datapath width. The target processor description for Valen-CC includes the instruction set consisting of behavior and assembly syntax of each instruction as well as the processor datapath width. Valen-CC does not explicitly support processor pipelines or ILP. In addition to Valen-CC, a cycle-accurate simulator has been developed [25]. The processor description for the simulator is different from one used for Valen-CC. The processor description for the simulator is an RT-level net-list written in a hybrid language of Verilog and Lisp. The processor description language for the simulator and that for Valen-CC are being integrated in order to improve ease of specification and to support a wider range of processor architectures.

CSDL is a set of machine description languages used in the Zephyr compiler system developed at University of Virginia [26]. Zephyr is an infrastructure for compiler writers rather than a retargetable compiler, consisting of several machine-independent modules (e.g., HLL parsers) and retargetable modules (e.g., code optimizers). A retargetable code optimizer, called VPO, plays a central role in Zephyr, which performs instruction selection, register allocation, code motion, and so on. A compiler writer can develop a compiler for his/her target processor by retargeting VPO and some other pre-designed modules, and by newly describing some specific modules. CSDL consists of four languages: SLED, λ-RTL, CCL, and PLUNGE. SLED describes assembly and binary representations of instructions [27], while λ-RTL describes the behavior of instructions in a form of register transfers [28]. CCL specifies the convention of function calls [29]. PLUNGE is a graphical notation for specifying the pipeline structure. Since there is no literature on details of PLUNGE, it is not clear whether CSDL has an ability to automatically generate cycle-accurate simulators and/or detect conflicts between instructions.

EXPRESSION which has been developed at UC Irvine is an ADL designed to support DSE of a wide class of processor architectures ranging from RISCs, DSPs, ASIPs, and VLIWs, coupled with a variety of memory system organization and hierarchies [30]. EXPRESSION contains an integrated specification of both structure and behavior of processor-memory systems. The structure contains a net-list of components (i.e., units, storages, ports, and connections), as well as the valid unit-to-storage or storage-to-unit data transfers. The pipeline architecture is described as the ordering of units which comprise the pipeline stages, plus the timing of the multi-cycled units. The behavior describes the instruction set. Each operation is defined in terms of its opcode, operands, and format which is used to indicate the relative ordering of various operation fields. Each instruction is viewed as a list of slots to be filled with operations. From the EXPRESSION description, an ILP compiler and a cycle-accurate simulator are automatically generated. In EXPRESSION, resource conflicts between instructions are not explicitly described, but reservation tables specifying the conflicts are automatically generated and passed to the ILP compiler [31]. Since manual description of reservation tables for deeply pipelined or VLIW processors is cumbersome and error-prone, this ability to automatically generate reservation tables drives rapid DSE of SOC architectures. V-SAT is a graphical front-end for rapid capture and SOC DSE using EXPRESSION [32]. Another key feature of EXPRESSION is the ability to specify novel memory systems including memory hierarchies, on-chip DRAMs, frame buffers, partitioned memory address spaces, and so on.

3.3. Simulation-Oriented ADLs

LISA is an ADL developed at RWTH Aachen which supports bit-true cycle-accurate simulator generation for DSPs [33, 34]. In LISA, both structure and behavior can be modeled. The main characteristics of LISA is the operation-level description of the pipeline which is able to model complex interlocking and bypassing techniques. Each instruction consists of multiple operations which are defined as register transfers during a single control step (i.e., instruction-, cycle-, or phase-level depending on the required accuracy). LISA provides explicit support of some pipeline behaviors such as pipeline stalls and flushes. However, LISA does not provide explicit description of resource conflicts for ILP compilers. Resource conflicts are modeled with signals which capture the occurrence of certain opcodes in the pipeline stages and trigger different strategies to stall, shift, or freeze the pipeline. There exist no mechanism to generate compilers from LISA descriptions.

RADL is an extension of the LISA approach proposed at Rockwell Semiconductor Systems [35]. Cycle-

and phase-accurate simulators can be generated for DSPs. RADL supports more detailed pipeline modeling such as delay slots, interrupts, zero-overhead loops, hazards, and multiple pipelines. Similar to LISA, resource conflicts for ILP compilers are not explicitly captured.

3.4. Validation-Oriented ADL

AIDL is an ADL developed at University of Tsukuba for design of high-performance superscalar processors [36]. It seems that AIDL does not aim at datapath optimization but aims at validation of the pipeline behavior such as data-forwarding and out-of-order completion. In AIDL, timing behavior of pipeline is described using *interval temporal logic*. AIDL does not support software toolkit generation. However, AIDL descriptions can be simulated using the AIDL simulator. Although there exist no synthesis tools for AIDL, AIDL descriptions can be translated into synthesizable VHDL code [37]. However, no methodology for translation is presented.

3.5. Other Approaches

PEAS-I is a CAD system for ASIP design supporting automatic instruction set optimization, compiler generation, and instruction level simulator generation [38]. In the PEAS-I system, the GNU C compiler is used, and the machine description of GCC is automatically generated. Therefore, there exists no specific ADL in PEAS-I. Inputs to PEAS-I include an application program written in C and input data to the program. Then, the instruction set is automatically selected in such a way that the performance is maximized or the gate count is minimized. Based on the instruction set, GNU CC and an instruction level simulator is automatically retargeted. PEAS-I assumes a 4-stage single pipelined processor architecture, and does not assume multi-word instructions.

3.6. Comparison between ADLs

Table 1 summarizes the comparison between some existing ADLs in terms of their supports for compiler generation (and ability of ILP optimization), ILP constraint detection, simulator generation (and accuracy), synthesis and formal verification. Also, information captured by the ADLs is compared.

Since MIMOLA and UDL/I are originally HDLs, their descriptions are synthesizable and can be simulated using HDL simulators. MIMOLA appears to be successful for retargetable compilation for DSPs with irregular datapaths. However, since its abstraction level is rather low, MIMOLA is laborious to write. COACH supports generation of both compilers and simulators. However, it assumes too simple processor architectures without pipelines or ILP. nML and ISDL supports ILP compiler generation. However, due to the lack of structural information, it appears to be impossible to automatically detect conflicts between instructions. MDES supports simulator generation only for the HPL-PD processor family. It is not clear whether MDES has enough information to generate simulators for the other processor families and to generate conflicts between instructions. EXPRESSION has ability to automatically generate ILP compilers, reservation tables, and cycle-accurate simulators. Furthermore, description of memory hierarchies is supported. LISA and RADL are dedicated to simulator generation and do not support compiler generation. Although AIDL descriptions are executable on the AIDL simulator, AIDL does not support compiler generation.

From the above comparison as well as our experiences of research/development, we believe that ADLs should capture both behavior (instruction set) and structure (net-list) information in order to generate high-quality software toolkit automatically and efficiently. Behavior information which is necessary for compiler generation should be explicitly specified because of mainly two reasons. One reason is that instruction set extraction from net-lists described in synthesis-oriented ADLs or HDLs does not seem to be applicable to a wide range of processors. The other reason is that synthesis-oriented ADLs or HDLs are generally tedious to write for the purpose of DSE. Also, structure information is necessary not only to generate cycle-accurate simulators but also to generate ILP constraints which are necessary for high-quality ILP compiler generation.

4. Challenges

Most existing ADLs are designed to specify processor architectures of uni-processor systems and generate software toolkits automatically. There are several future research directions to pursue, for example, formal verification, synthesis, runtime environment generation, implication to IP libraries, and support of wide ranges of SOC architectures.

As described in Section 2.2, ADLs are used for not only software toolkit generation but also verification and synthesis. Up to now, formal verification of ADL descriptions has not been addressed explicitly. However, formal verification of processor pipelines in an early design phase is becoming essential as ILP in processors increases and pipelines become deeper. Especially, pipeline conflicts (or hazards) have to be detected automatically. In general, there exist three kinds of pipeline conflicts: resource conflicts, data conflicts, and control conflicts. In EXPRESSION, resource conflicts are automatically detected to generate reservation tables [31]. Although their main objective of reservation table generation is high-quality ILP compiler gen-

Table 1. Comparison between different ADLs

	MIMOLA	COACH-UDL/I	nML	ISDL	MDES	EXPRESSION	LISA	RADL	AIDL
Compiler gen.	√	√	√	√	√	√			
Is ILP?	√		√	√	√	√			
ILP const. detect.	△					√			
Simulator gen.	△	√	√	√	△	√	√	√	△
Is cycle-accurate?	△	△		√	△	√	√	√	△
Synthesis	√	√		△					△
Formal verification						△			△
Inst. set info.			√	√	√	√	√	√	
Net-list info.	√	√				√	√	√	
Memory hier. info.						√			

√ Supported △ Supported with restrictions

eration, their techniques can be used for formal verification of resource conflicts. In addition, verification of data and control hazards should be addressed in future. Formal verification of consistency in ADL descriptions is also important. As mentioned in Section 3.6, ADLs should contain both behavior (instruction set) and structure (net-list) information. We should check whether the instruction set is implemented by the structure or not. Furthermore, it should be examined whether the structure has no redundant components.

Synthesis of processors from ADL descriptions should be supported since designing processors is generally very complex and time-consuming. In ADL descriptions, detailed structures of control paths are not captured, and therefore, should be synthesized automatically.

Several existing ADLs support automatic generation of compilers and simulators, but no ADL supports generation of runtime software environments such as Real-Time Operating Systems (RTOSs). Since RTOSs have a large impact on system performance and cost, automatic generation or optimization of RTOSs is one of important future research avenues. Special attention must be paid to task scheduling, interrupt handling, memory management, device drivers, and so on.

In order to realize efficient DSE of SOC architectures, ADL descriptions must be reusable by storing them into IP libraries. To achieve this, standard mechanisms to specify SOC architectures in ADLs and to encapsulate their design attributes (e.g., performance, power consumption, feature size, etc.) are essential.

ADLs must specify architecture templates including (a) blocks/components which reside on the SOC, (b) how they are connected or interact, and (c) functionality of each block/component, as describe in Section 2.1. Future SOCs will contain heterogeneous multiple processors with multi-threaded architectures, ASICs, and various types of memories (e.g., DRAMs and flash memories), and so on, which are irregularly connected in order to efficiently execute specific applications. Since most existing ADLs are tailored to specify processor architectures of uni-processor systems, they should be extended to capture wider ranges of SOC architectures. Special attention should be paid to specification of heterogeneous multi-processors and memory hierarchies with various memory technologies.

5. Conclusions

In this paper we described an ADL-based SOC codesign methodology which enables efficient DSE of SOC architectures. In the codesign methodology, ADLs are used to specify architecture templates of SOC architectures at a system level, verify the SOC architectures, generate software toolkits, and synthesize the SOCs. We surveyed recent efforts in the use of ADLs, and addressed several major challenges facing ADL-based codesign of future SOCs.

We believe that ADL-based codesign is a key technology to efficiently design complex SOCs containing programmable processors and memory hierarchies both of which are highly customized to specific applications.

References

[1] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. *The MIMOLA Language Version 4.1.* University of Dortmund, 1994.

[2] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems, Kluwer Academic Publishers*, 3(1):1–36, January 1998.

[3] R. Leupers. *Retargetable Code Generation for Digital Signal Processors.* Kluwer Academic Publishers, 1997.

[4] R. Leupers and P. Marwedel. A BDD-based front-end for retargetable compilers. In *Proc. of ED&TC95*, 1995.

[5] R. Leupers. HDL-based modeling of embedded processor behavior for retargetable compilation. In *Proc. of 11th ISSS*, 1998.

[6] H. Akaboshi. *A Study on Design Support for Computer Architecture Design.* PhD thesis, Dept. of Information Systems, Kyushu University, Jan. 1996.

[7] UDL/I Committee. *UDL/I Language Reference Manual Version 2.1.0a*, 1994.

[8] H. Akaboshi and H. Yasuura. Behavior extraction of MPU from HDL description. In *Proc. of APCHDL*, 1994.

[9] Fintronic USA Inc. *UDL/I 2.0 Simulation Environment User's Guide*, 1995.

[10] ASTEM RI. *UDL/I Logic Synthesis System User's Guide Ver. 1.1.0*, 1995.

[11] M. Freericks. The nML machine description formalism. Technical Report 1991/15, Fachbereich Informatik, TU Berlin, 1991.

[12] F. Lohr, A. Fauth, and M. Freericks. SIGH/SIM: An environment for retargetable instruction set simulation. Technical Report 1993/43, Fachbereich Informatik, TU Berlin, 1993.

[13] A. Fauth and A. Knoll. Automatic generation of DSP program development tools. In *Proc. of ICASSP*, 1993.

[14] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. *CHESS: Retargetable Code Generation for Embedded DSP Processors,* In *Code Generation for Embedded Processors* (P. Marwedel and G. Goossens, ed.), Kluwer Academic Publishers, 1995.

[15] M. R. Hartoog, J. A. Rowson, P. D. Reddy, S. Desai, D. D. Dunlop, E. A. Harcourt, and N. Khullar. Generation of software tools from processor descriptions for hardware/software codesign. In *Proc. of 34th DAC*, 1997.

[16] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proc. of ED&TC*, 1995.

[17] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proc. of 34th DAC*, 1997.

[18] S. Hanono and S. Devadas. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In *Proc. of 35th DAC*, 1998.

[19] G. Hadjiyiannis, P. Russo, and S. Devadas. A methodology for accurate performance evaluation in architecture exploration. In *Proc. of 36th DAC*, 1999.

[20] J. C. Gyllenhaal. A machine description language for compilation. Master's thesis, Dept. of ECE, UIUC, Sept. 1994.

[21] J. C. Gyllenhaal, W. W. Hwu, and B. R Rau. Optimization of machine descriptions for efficient use. *Int'l Journal of Parallel Programming*, 26(4):417–447, August 1998.

[22] P. Paulin, C. Liem, T. May, and S. Sutarwala. *FlexWare: A Flexible Firmware Development Environment for Embedded Systems,* In *Code Generation for Embedded Processors* (P. Marwedel and G. Goossens, ed.), Kluwer Academic Publishers, 1995.

[23] A. Inoue, H. Tomiyama, F. N. Eko, H. Kanbara, and H. Yasuura. A programming language for processor based embedded systems. In *Proc. of APCHDL*, 1998.

[24] A. Inoue, H. Tomiyama, H. Okuma, H. Kanbara, and H. Yasuura. Language and compiler for optimizing datapath widths of embedded systems. *IEICE Trans. Fundamentals*, E81–A(12):2595–2604, Dec. 1998.

[25] F. N. Eko and H. Yasuura. A cycle-accurate simulator toolkit for soft-core processors. In *Proc. of 6th APCHDL*, 1999.

[26] A. Appel, J. Davidson, and N. Ramsey. The Zephyr compiler infrastructure. Internal Report, `http://www.RCS.virginia.edu/zephyr`, 1998.

[27] N. Ramsey and M. F. Fernández. Specifying representations of machine instructions. *ACM Trans. Programming Languages and Systems*, 19(3):492–524, May 1997.

[28] N. Ramsey and J. W. Davidson. Machine descriptions to build tools for embedded systems. In *Proc. of LCTES*, 1998.

[29] M. W. Bailey and J. W. Davidson. A formal model and specification language for procedure calling conventions. In *Proc. of 22th POPL*, 1995.

[30] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. of DATE*, 1999.

[31] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. to appear at *12th ISSS*, 1999.

[32] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. V-SAT: A visual specification and analysis tool for system-on-chip exploration. In *Proc. of EUROMICRO DSD*, 1999.

[33] S. Pees, A. Hoffmann, V. Živojnović, and H. Meyr. LISA: Machine description language for cycle-accurate models of programmable DSP architectures. In *Proc. of 35th DAC*, 1999.

[34] V. Živojnović, S. Pees, and H. Meyr. LISA: Machine description language and generic machine model for HW/SW co-design. In *Proc. of Workshop on VLSI Signal Processing*, 1996.

[35] C. Siska. A processor description language supporting retargetable multi-pipeline DSP program development tools. In *Proc. of 11th ISSS*, 1998.

[36] T. Morimoto, K. Yamazaki, H. Nakamura, T. Boku, and K. Nakazawa. Superscalar processor design with hardware description language AIDL. In *Proc. of APCHDL*, 1994.

[37] T. Morimoto, K. Saito, H. Nakamura, T. Boku, and K. Nakazawa. Advanced processor design using hardware description language AIDL. In *Proc. of ASP-DAC*, 1997.

[38] J. Sato, A. Y. Alomary, Y. Honma, T. Nakata, A. Shiomi, N. Hikichi, and M. Imai. PEAS-I: A hardware/software codesign system for ASIP development. *IEICE Trans. Fundamentals*, E77–A(3):483–491, March 1994.