

A Study of Architecture Description Languages from a Model-based Perspective

Wei Qin
Boston University
Boston, MA 02215
Email: wqin@bu.edu

Sharad Malik
Princeton University
Princeton, NJ 08544
Email: sharad@princeton.edu

Abstract—Owing to the recent trend of using application-specific instruction-set processors (ASIP), many Architecture Description Languages (ADLs) have been created. They specify architectures or microarchitectures of processors, and automate tasks including circuit implementation, simulation, retargetable compilation and formal verification. Despite the large amount of effort dedicated to the field in the past two decades, we see no sign of convergence in it. In other words, the existing research results are barely reused as the foundation of new research projects. This situation makes it hard for new researchers to enter the field. To nurture a continuous growth of the field, this paper advocates two moves – the creation of formal foundations of ADLs and the adoption of the open-source development model. We argue that for an ADL to be capable of rigorously specifying a processor, it must be based on a solid foundation which we call the architecture model. The existing ADLs feature a wide variety of formal and ad-hoc architecture models which confines the flexibility and analyzability of the ADLs in one way or another. The paper discusses the Operation State Machine (OSM) model, the result of our first attempt to create high-level processor models. The model has features balanced flexibility and analyzability for use in architecture space exploration frameworks for ASIPs. The paper also describes the use of the OSM model in the Mescal Architecture Description Language (MADL), an open-source ADL framework that we developed. Lastly, it points out the potential application of formal verification techniques on OSM.

I. INTRODUCTION

Architecture description languages (ADLs) represent the instruction-set architecture (ISA) and/or microarchitecture of processors. Owing to the recent trend of using application-specific instruction-set processors (ASIPs), ADLs have received substantial interest in both academia and industry. They have been used to assist the generation of retargetable software development tools such as compilers, instruction-set and microarchitecture simulators, as well as the synthesis of circuit implementation for ASIPs. More recently, ADLs start to receive the attention from the testing and formal verification fields. They have been used to generate functional test programs, to validate the equivalence between the ISA and microarchitecture, and to verify functional properties of processors.

The diverse applications of ADLs place diverging requirements on their semantics. To most ADL designers, it is important that the ADL can express a wide range of processors accurately. This allows its users to explore a large design space and to represent important characteristics faithfully. We define this aspect as the *flexibility* of an ADL. For ADLs targeting compiler generation, testing, or formal verification applications, it is important that the abstraction level of the ADL be high enough so that architectural properties can be extracted and analyzed. We define this aspect as the *analyzability* of the ADL. We view flexibility and analyzability as the most important features of the semantics of an ADL. However, in reality, they are often conflicting with each other – an analyzable ADL requires a high level of abstraction for its semantics but a flexible ADL normally prefers low level semantics. It remains a huge challenge to balance these two aspects in creating an ADL.

In order to gain deep understanding of the flexibility and analyzability of an ADL, we advocate using *architecture model* – the computation model based on which the semantics of an ADL is defined – to study these two aspects. It defines how the architectural or microarchitectural components operate and how they interact with each other. Separating the architecture model from the remaining part of an ADL, mainly its syntax and type system, allows us to focus on the information that is directly related to the above two aspects.

Existing ADLs use a wide variety of architecture models from the general discrete-event (DE) model to ad-hoc architecture templates. Due to the diverse requirements from the applications that the ADLs intend to support, and also due to the conflicting nature of flexibility and analyzability, it is extremely difficult, if ever possible, to find an architecture model that suffices all purposes. In this paper, we will give a comprehensive survey of the existing ADLs and their architecture models. We will also present our effort toward creating a flexible and analyzable model, which resulted in the operation state machine (OSM) model.

This paper is organized as follows. Section II sur-

veys the existing ADLs and their architecture models. Section III presents the operation state machine (OSM) model and briefly discusses the Mescal Architecture Description Language (MADL) which is based on the OSM model. We then discuss the potential of using the OSM model for formal verification purposes in Section IV and conclude the paper in Section V.

II. A SURVEY OF ADLS

A processor may be viewed from a variety of angles, including the microarchitecture, the memory organization, the instruction semantics, the assembly syntax, the instruction encoding, and the abstract binary interface (ABI). According to the supported views, ADLs are traditionally classified into three categories: structural ADLs, behavioral ADLs, and mixed ADLs [40], [31]. Structural ADLs utilize component netlists to describe the structural details of processors at the logic or the microarchitecture level. ADLs in this category include MIMOLA [43] and UDL/I [1]. They are suitable for synthesizing hardware and generating cycle-accurate simulators (CAS)'s. In contrast, behavioral ADLs focus on describing the instruction sets of processors. They are more suitable for generating compilers or instruction-set simulators (ISS)'s. Behavioral ADLs include ISDL [15], CSDL [33] and the early version of nML [13]. Mixed ADLs stand in between and contain both instruction set and coarse-grained structural information. Most ADLs belong to this category, including UPFAST [26], BUILDABONG [38], LISA [27], RADL [36], ArchC [35], EXPRESSION [16], PRMDL [39], HMDDES [14], Maril [4], TIE [42], BABEL [24], and the latest version of nML [37].

One drawback of the above classification approach is its low resolution – most existing ADLs fall into the mixed ADL category and cannot be differentiated. To address this problem, we proposed to use architecture model to classify ADLs. An architecture model represents the concurrent behavior of a processor at the instruction-set and/or the microarchitecture levels, which is the key semantics that an ADL needs to capture. A classification method based on it can provide an incisive understanding of the semantics of the ADLs. It also helps evaluate those processor modeling frameworks that are based on general programming languages such as C and C++. These frameworks include LSE [41], HASE [7], and Asim [12]. Below we describe the architecture models and related modeling frameworks.

A. Discrete Event Model

The discrete event (DE) model is the standard MoC for modeling digital circuits. It has been used by MIMOLA [43] whose creation preceded Verilog [18] and VHDL [17]. MIMOLA contains two parts: the hardware description part and the high-level programming part.

The former describes a processor at the register transfer level (RTL level). MIMOLA has been used by a series of tools including the MSSH hardware synthesizer, the MSSQ code generator, the MSST self-test program compiler, the MSSB functional simulator, and the MSSU RTL simulator [21].

Another DE-based processor description language is UDL/I developed at Kyushu University in Japan [1]. It serves as the input to the COACH ASIP design automation system. The COACH system can extract the instruction set from the RTL level UDL/I description for a small class of processors. The instruction set information can be used to generate a target-specific compiler and an ISS.

HASE is a processor modeling environment based on the DE MoC [7]. Its modules are described in C++ as light-weight threads, which are scheduled by a DE simulation engine. Compared to MIMOLA and UDL/I, the components in HASE are of much coarser granularity.

B. Synchronous Structural Model

Most processors are implemented as synchronous logic circuits. Therefore, a synchronous MoC is sufficient to model the vast majority of processors. Synchronous MoCs do not need the expensive event calendar and therefore are more efficient for simulation. A few processor modeling frameworks take advantage of this fact and use synchronous computation models.

Asim [12] is a processor modeling environment developed at Compaq for performance modeling of high-end processors. In an Asim model, the hardware modules communicate with each other in a clock-driven fashion. No combinational path between any input and any output of a module is allowed. This limits the way of drawing the boundary between the modules in a processor model: all combinational components must be folded into some sequential modules with registered output.

The Liberty Simulation Environment (LSE) [41] is based on a sophisticated synchronous model, the Heterogeneous Synchronous/Reactive (HSR) MoC [11]. In an HSR system, structural components are viewed as black boxes connected by unidirectional unbuffered channels. When the system receives any input, the components connected to the input channel will react instantaneously according to their operation semantics and change their outputs. Such outputs will further trigger downstream components, which will in turn update their outputs instantaneously. The process continues until all outputs stabilize.

The synchronous structural modeling approaches target the generation of CAS's, but not ISS's or compilers for high-level languages. For the modeling of logic hardware, the synchronous structural models are less flexible than DE models since they cannot model

cyclic combination paths. However, to many computer architects, the improved simulation efficiency of these approaches over DE models overshadows this limitation.

C. Synchronous Behavioral Model

The synchronous structural models use component netlists to model hardware structures. The communication of data values between modules is through copying between ports and channels. Such copying introduces significant runtime overhead in simulation.

UPFAST achieves faster simulation speed by avoiding structural netlists [26]. In an UPFAST model, the communication between microarchitecture modules is achieved through directly referencing the states, which are declared as global variables. This implementation style trades modularity for simulation speed.

Another difference of UPFAST from the above-mentioned structural approaches is its explicit notion of instructions. It describes instruction semantics as a number of time annotated procedures (TAP). When an instruction arrives at a pipeline stage, its TAPs corresponding to the stage will be evaluated. Under this scheme, UPFAST users need to explicitly schedule all TAPs into pipeline stages and clock phases. For pipelining behaviors involving the dependency of multiple instructions, this scheme is difficult to use.

D. Domain-specific Models

Computer architects often abstract pipeline stages as place holders for instructions. This view point is exploited by some ADLs to simplify the modeling of microprocessors. These include LISA, RADL and ArchC.

LISA was developed at Aachen University of Technology in Germany [27]. It has been used in commercial tools of AXYS [3] and CoWare [9]. The atomic functional entity in LISA is the *operation*, which roughly corresponds to the behavior of an instruction inside of a pipeline stage. The LISA simulation kernel utilizes the Gantt-Chart (similar to the pipeline diagram) to control the execution progress of instructions. In a CAS generated from LISA, each pipeline stage is simplified as an operation buffer. When an instruction is fetched and decoded, it is decomposed into several operations, which are inserted to the operation buffers of the corresponding pipeline stages. An operation will be evaluated when its parent instruction advances to the pipeline stage holding the operation. In summary, the MoC adopted by LISA is the combination of the Gantt-Chart and its operation scheduling mechanism. It is limited to modeling in-order pipelines.

Similar to LISA, RADL [36] and ArchC [35] also utilized the Gantt-Chart notion. RADL was derived from an early version of LISA and therefore very similar to LISA. ArchC is based on SystemC. It is designed for

simulator generation. Except for several special syntax constructs, the main part of an ArchC description is in C++. The MoC of ArchC can be viewed as the combination of Gantt-Charts and DE.

E. Petri-net-based Models

Petri-net [25] is a mathematical model for asynchronous and nondeterministic concurrent systems. Several experiments have been made to use its various extensions to model the microarchitecture of processors. These include the use of Colored Petri-net (CPN) [5], Timed Colored Petri-net (TCPN) [44], Petri-net for Digital Systems (PNDS) [10], and Reduced Colored Petri-net (RCPN) [34]. The last two were newly created for modeling processors.

In Petri-net-based processor models, the *tokens* are often used to represent instructions and data. The *places* are often used to represent pipeline stages. And the *transitions* are used to represent evaluation semantics of the instructions.

F. Architecture Templates

Many ADLs were created as configuration systems for the software tools that they intended to support. In a typical design environment based on such an ADL, a generic processor template serves as the foundation of the description. The ADL files provide parameters to configure some aspects of the processor template that are of interest to the software tools.

nML was originally developed to specify ISAs [13]. It was later commercialized and extended to include structural and timing information [37]. In its new version, instruction semantics can be described with regard to the pipeline stages that it goes through. The specification of pipeline control and timing information in the new nML is largely based on parameters. Due to this constraint, nML is mainly used to design DSPs with simple pipeline control.

ISDL is a behavioral ADL for specifying ISAs of DSPs [15]. Limited instruction timing information such as execution latency can be specified in ISDL, but not any pipeline control information. ISDL intends to assist the retargetable compilation and simulation of simple DSPs. It supports describing irregular instruction set constraints with Boolean expressions.

Similar to ISDL, TDL utilizes some parameters to specify instruction timing information [19]. It was used in a post-pass optimization (assembly optimization) framework named PROPAN, which targets VLIW DSPs with simple control paths. Another similar work is the Maril ADL [4] which helps to configure the Marion retargetable compiler. Its target scope is general purpose processors with RISC style instruction sets.

EXPRESSION is an ADL developed to assist the generation of both simulators and high-level language (HLL) compilers [16]. It utilizes coarse-grained netlists (similar to sketch diagrams) of pipeline stages and storage components to specify the structure of a processor. Each pipeline stage is configured with several parameters including the name of the output latch, the names of the ports, the capacity of the stage, the operations that can go through the stage, and the latency. In the early version of EXPRESSION [16], pipeline control information, such as the conditions to stall and to flush, is implicit. This limits its capability to describe a large range of processors. A later paper [23] reported a more flexible template for explicit modeling of pipeline control.

Other ADLs in this category include the HMDES language [14] for the IMPACT research compiler, and the TIE language [42] describing instruction extensions for Tensilica’s Xtensa processors. All architecture template approaches are based on ad-hoc architecture models. One advantage of these ADLs is the conciseness of processor descriptions since a large portion of architecture information is implicitly encoded into their generic templates. Another advantage is that their relative ease to extract required information from these templates for the use of software tools, especially HLL compilers. The drawback of architecture templates, however, is their significantly limited architecture range. It is also difficult to convey the range limitation to general users of the ADLs since the information can hardly be formulated in a mathematical way.

G. Other Models

Some ADLs do not fit into any of the above categories. Below we briefly describe them.

BUILDABONG is a design environment for ASIP design [38]. It models the hardware structures of processors based on the Abstract State Machine formalism (ASM). An ASM model simply contains a set of synchronous transition rules in the form of

$$if \langle cond \rangle then \langle updates \rangle .$$

At every cycle, all rules will be evaluated simultaneously. Each rule tests its Boolean condition “cond”, and then evaluates the “updates” statements if the condition is true. The rules represent the hardware implementation of the processor at the RTL level.

Mathews *et al.* designed a functional language named HAWK for describing synchronous processors models at the RTL level [22]. One potential advantage of a functional language is the reduced number of software bugs due to rigorous type checking. However, it is not intuitive for most computer architects to specify hardware in such a language. The practicality of the approach remains a question.

CSDL is a family of machine description languages for the Zephyr compiler infrastructure [33]. It contains CCL, a function calling convention specification language; SLED, a formalism describing instruction assembly syntax and binary encoding; and λ -RTL, a register transfer language for instruction semantics description. These languages only describe the instruction set and the programming interface of a processor.

BABEL [24] is an ADL for retargeting the popular processor simulation framework SimpleScalar [2]. Similar to CSDL, it describes the instruction set and the programming interface of a processor. No detail was released on its microarchitecture modeling and description approach.

H. Summary

This section surveyed previous effort in presenting in the field of microprocessor modeling and description. The structural models, including the DE, HSR, ASM, and Petri-net models, focus on hardware structures and hence are suitable to be used for the synthesis of hardware and CAS’s. While the Gantt-Chart, the synchronous behavioral model and most architecture templates contain both high-level structural and instruction set information. Therefore, they may be used to generate cycle-accurate simulators, instruction set simulators, and compiler components.

In general, designing an ADL is a time-consuming engineering task, so does accurately describing a real-world processor in the ADL. The two tasks are often intertwined with each other. Alongside the creation process of an ADL, a few processor models are implemented as test cases. Feedbacks from the test cases are used to guide the design process of the ADL. And modifications to the ADL require the test cases to be updated. The design loop can take a long time to converge. In some cases, it took tens of man-years for an ADL to reach satisfactory quality.

Complexity can be overcome by diligent engineering effort and good management. But a more serious problem is that there seems no sign of convergence of the field. In other words, existing research results or engineering efforts are rarely used as the foundation of new ADL projects. This may be attributed to the fundamental diversity in the field: existing ADLs use a variety of formal and informal architecture models, target a variety of processor classes, serve a variety of purposes, and employ a variety of algorithms. This may be also be due to the lack of formal theory, or commonly recognized abstraction layers, in the field. Without such abstraction layers, it is hard to share development effort among researchers: each group of designers have to implement the whole infrastructure and the machine descriptions from scratch. The consequences are wasted engineering efforts

in the repeated implementation basic infrastructure, and the intimidating roadblocks for new researchers to enter this field. This situation thwarts the continuously fast development of the field.

To overcome the above-mentioned difficulties, we believe that some moves need to be taken by researchers. First of all, we believe that more attention needs be paid to the development of formal models and standard abstraction interfaces. Secondly, we advocate sharing of implementation details of ADLs and processor descriptions. Source code is an important complement to technical publications for researchers to understand each other’s work. It also allow researchers to reuse the processor models that would otherwise take years to re-develop. The good news is that there exists a growing trend of open-sourcing led by projects such as ArchC, Liberty and MESCAL.

III. OSM MODEL

In this section we introduce the Operation State Machine (OSM) model that was developed during the course of the MESCAL (Modern Embedded Systems – Compilers, Architectures and Languages) project. It represents our first effort in developing theoretical foundations for ADLs and in implementing open-source infrastructures.

The OSM model was initially proposed in a conference paper in 2003 [32]. Since then a significant amount of effort has been spent in refining it along the development course of MADL (the Mescal Architecture Description Language). The initial purpose of the OSM model and MADL was to support the generation of software development tools including ISS’s, CAS’s, and compilers. It was designed with balanced flexibility and analyzability to support these tools. In this section, we give an introduction to the major aspects of the OSM model.

A. Abstractions

The OSM model views processors at two levels, the operation level and the hardware level. The operation level contains the ISA and the dynamic execution behavior of the operations. The hardware level represents the simplified microarchitecture as a result of the abstraction mechanisms used in the OSM model. Separation of the two levels allows us to analyze relevant information at each level.

a) Abstraction of Operation: At the operation level, we use extended finite state machines (EFSM) to model the execution of operations.¹ Each EFSM represents one operation in the pipeline. Thus the name *operation state machine* (OSM) is used for these special-purpose

¹An EFSM is a traditional finite state machine (FSM) with internal state variables [6]

EFSMs. The states of an OSM stand for the execution statuses of the operation that it represents; while its edges stand for the valid execution steps. Each edge of the OSM is controlled by a *condition*, representing the readiness of the operation to progress along the edge. Such readiness is expressed as the availability of execution resources, including structural resources, data resources and artificial resources created for modeling purposes. Example resources include pipeline stages, reorder-buffer entries and the availability of operands.

b) Abstraction of Resource: The execution resources are maintained in the hardware level of the OSM model. They are modeled as *tokens*. A token may optionally contain a data value. A *token manager* controls a number of tokens of the same type. It grants the tokens to the OSMs according to its token management policy and the requests from the OSMs. In essence, a token manager is an abstract implementation of a control policy in the processor. Depending on the control semantics of the microarchitecture components, token managers can have different policies.

We defined a set of token transaction primitives for an OSM to exchange tokens with the token managers. They include *allocate*, *inquire*, *release* and *discard*. Once a token has been allocated to an OSM, it cannot be *allocated* to another one until the first OSM *releases* or *discards* the token. But other OSMs may still be able to *inquire* about the allocated token.² Except for *discards*, each token transaction contains a condition and a side-effect. *Discard* may be viewed as an unconditional version of *release* and has only the side-effect. It is useful to model the flushing of an operation.

c) Putting Things Together: Each edge of an OSM is associated with a set of token transaction primitives. The conjunction of the conditions of all token transactions on an edge forms the above-mentioned *condition* of the edge. If the condition is true, then all side-effects of the transactions can take place and the OSM transitions along the edge to its next state. Otherwise, no side-effect can take place and the OSM either stays in the original state or transitions along an alternative edge that has a true condition.

d) Data-flow: The OSMs can evaluate instruction semantics and communicate data values with token managers. In addition to the above-mentioned token transaction primitives, arithmetic and logic expressions can be associated to the edges. We also defined two data transaction primitives *read* and *write* to allow an OSM to exchange data value with tokens. They can also be annotated onto edges.

²In the the initial version of the OSM model in [32], other OSMs are not allowed to inquire about the allocated token. The restriction has been removed to improve flexibility.

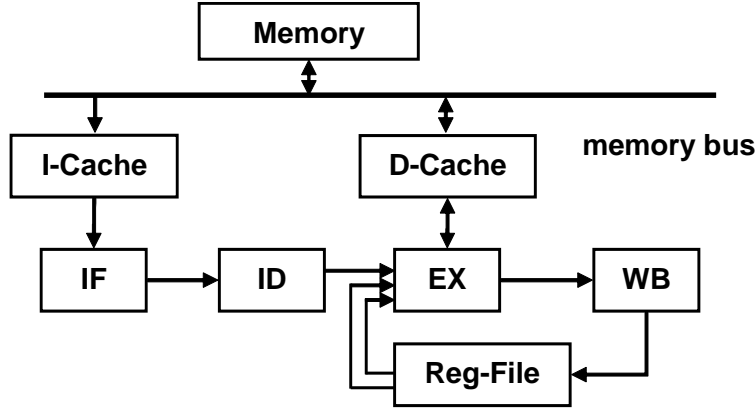


Fig. 1. Example scalar processor

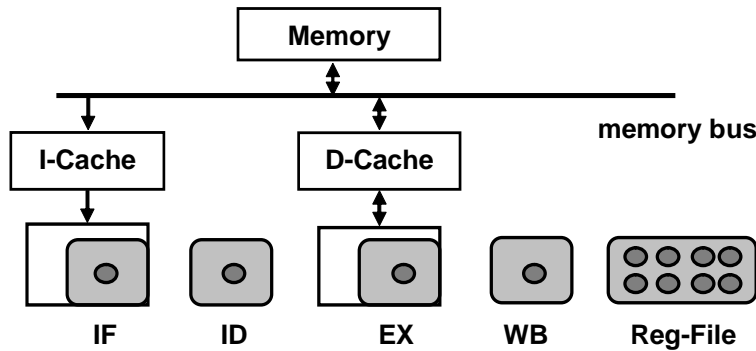


Fig. 2. Hardware level model for the example processor

B. An Example

We use the scalar processor shown in Figure 1 as a modeling example. The microarchitecture of the processor contains four pipeline stages, the register file, the instruction cache, the data cache, the memory bus, and the main memory.

Following the OSM modeling scheme, a two-level OSM model is created for the scalar processor. The hardware level of the model, shown in Figure 2, contains 8 components corresponding to those in Figure 1. Each of the components IF, ID, EX, WB, and Reg-File now contains a token manager indicated by the shaded boxes. ID, EX, WB, and Reg-File do not communicate with other components and therefore are reduced to standalone token managers. The others still have a DE interface to communicate with the caches or with each other.

In Figure 2, each of the IF, ID, EX, and WB token managers contains one token which represents the corresponding pipeline stage resource. The ownership of such a pipeline stage token by an OSM means that the operation represented by the OSM is at the pipeline stage. Since a token can be owned by at most one OSM at a time, no two operations can be at the same pipeline stage

simultaneously. This properly models the scalar-issuing behavior of the processor.

The Reg-File token manager contains the same number of tokens as its registers. Each token represents one register. It also has an associated value as the data content of the register. The token manager accepts integer-typed token identifiers, which are simply interpreted as register indexes. It helps to preserve data-dependency in the pipeline. Figure 3 shows an OSM used to model the “add” operation for the processor. It contains seven internal state variables iw , $v1$, $v2$, $v3$, rd , $rs1$, and $rs2$. We next consider an example “add $r1$, $r2$, $r7$ ” operation with semantics of storing the sum of the values of register $r1$ and $r2$ to register $r7$.

Prior to the fetching of the “add” operation, its corresponding OSM resides in its *dormant* state I .³ In the first cycle of the life range of the operation, it posts a token allocation request to the IF manager, which is the only action associated with edge e_0 . If the fetching stage (IF) is empty, the IF token is available. The OSM will be

³The dormant state refers to the status when the operation is not in the processor pipeline, either because it is not fetched yet or it has retired. An OSM has one and only one dormant state.

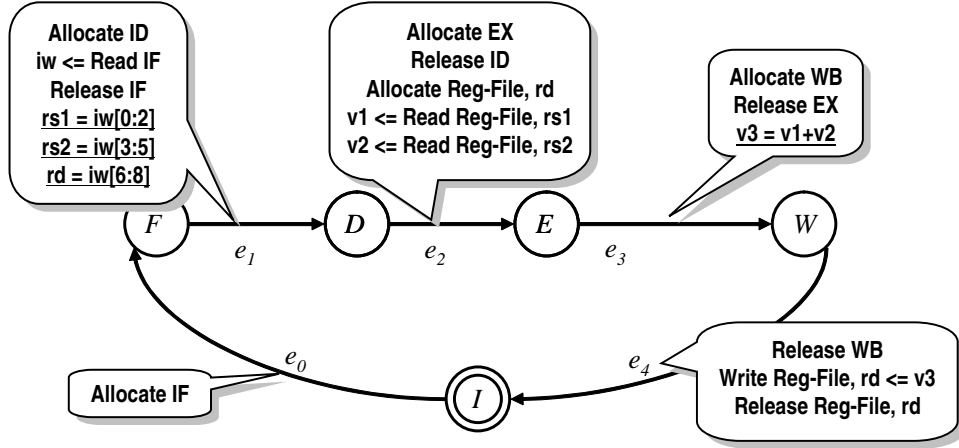


Fig. 3. OSM for the add operation

successful with its request and will obtain the ownership of the token. It will also progress along e_0 to state F , indicating the entrance of the operation into the fetching stage.

In the following cycle, the operation tries to advance further to the decoding stage (ID). In the model, the newly activated OSM sends an allocation request to the ID manager and a release request to the IF manager, which are control actions associated with edge e_1 . The former tests the availability of the decoding stage and the latter tests if the fetching stage has completed loading the operation from the instruction cache. If both tests are positive, the OSM will obtain the ID token, read the IF token value into the instruction word iw , release the IF token, and enter state D . This indicates that the operation leaves the fetching stage and enters the decoding stage. A few computation expressions are also associated with edge e_1 . They extract the operand fields from iw and place the values into rd , $rs1$, $rs2$. In this case, the field values are 7, 1 and 2, respectively.

In the next cycle, the operation tries to enter the execution stage (EX). In the model, the OSM sends an allocation request to the EX token manager asking for its token and a release request to the ID manager asking for the permission to leave. In order to get its source operands for computation, the OSM also sends inquiry requests to the Reg-File manager to test the availability of the source operands $r1$ and $r2$.⁴ To obtain the right to update its destination operand $r7$, the OSM also sends a token allocation request to the Reg-File manager to get the corresponding token. If all requests are successful, the OSM obtains the EX token and the Reg-File token, releases the ID token and enters state E , indicating that

⁴The *inquire* requests are not explicitly shown in Figure ?? since they are implicitly contained in the *read* actions.

the operation enters the execution stage. Meanwhile, the OSM reads its source operands from the Reg-File manager into $v1$ and $v2$.

In the fourth cycle, the operation attempts to move on to the write back (WB) stage. In the OSM model, this involves an allocation action with the WB manager and a release action with the BF manager. If all actions succeed, the OSM enters state W cycle and the operation gets into the WB stage. The computation expression exists on edge e_3 of evaluates the semantics of the operation.

In the last cycle of its life range, the operation retires from the pipeline. In the processor model, the OSM sends release requests to the WB manager and the Reg-File manager. If both are successful, the OSM writes its computation result in $v3$ to the destination token and releases both tokens. It then goes back to the dormant state I , finishing modeling the “add” operation.

The above description explains the correlation between an OSM and the “add” operation that it models. In [30], we documented more details of modeling a full instruction set and modeling pipeline behaviors such as all types of hazard and data-dependent latency. We also documented the scheduling issues for multiple OSMs and the interoperability of the OSM scheduler and the general DE scheduler.

C. Comparison with Other Models

Compared to the DE-based and the synchronous structural modeling approaches, the OSM model is more analyzable and compact but no less flexible. First of all, the abstraction of the operation level makes the model analyzable since operation behaviors are explicitly represented on the edges of OSMs. It allows model developers to separate the specification of the ISA from

low level microarchitectural details. Secondly, the abstraction mechanisms of token managers greatly simplified the modeling of the microarchitecture. This effect can be seen by comparing Figure 1 and Figure 2. The connectivity between the microarchitecture components is significantly reduced. An added benefit because of this is the improved simulation efficiency over structural models. Lastly, a portion of microarchitecture remains in the DE domain. This gives the model equal flexibility as the DE-based model.

Similar to the OSM model, the Petri-net-based modeling approaches reduce the connectivity between pipeline stages. However, they do not have a separate operation level. Therefore it is not straightforward for users to separate operation behaviors from hardware behaviors. In this sense, they are not as analyzable as the OSM model.

The Gantt-Chart model used by LISA [27] models operation flow control in pipelines. It also features the explicit notion of *operation* and therefore is of a similar abstraction level as the OSM model. However, due to its limited expressiveness in modeling the execution statuses of operations, it can model only in-order pipelines. Therefore, it is less flexible than the OSM model. Moreover, it is less analyzable since many operation properties are coded in an imperative style in LISA.

The architecture templates adopted by the many ADLs are less flexible than the OSM model. The OSM model does not assume any predefined architectural feature of the processor. In principle, it can be used to model the pipeline of any type of processors.

D. MADL

The Mescal Architecture Description Language (MADL) is designed as a description language of OSMs. The main intended usage of MADL is to support the generation of software tools that are involved in the design process of a processor. Such software tools include the CAS, the ISS, the HLL compiler, the assembler, and the disassembler. The current MADL implementation does not include the specification of the hardware level. Extending MADL to cover the complete processor is a part of our ongoing research.

One notable issue is that in order to make MADL descriptions compact, we created the notion of the dynamic-OSM model. In the dynamic OSM model, the token transaction requests are dynamically bound to the edges of the OSMs depending on run-time values. An MADL description specifies a dynamic-OSM model. The MADL compiler will convert it to the original (static) version. In [30], we documented the semantics of the dynamic OSM model and the process to convert it to the original OSM model. We also described the flow to generate the software tools from MADL. The syntax is documented

in the reference manual[29]. The manual, along with an MADL compiler and example processor models, has also been released at [28].

IV. FORMAL VERIFICATION APPLICATIONS

MADL is naturally suitable for simulation-based verification of microprocessors since the OSM model is executable. However, since simulation-based verification cannot provide complete assurance of correctness, we are currently investigating approaches to apply formal verification means to MADL descriptions. The analyzability of the OSM model makes automated formal verification feasible. We can perform model property checking and equivalence checking by looking into the operation level.

A. Model Property Checking

This task verifies the soundness of a given OSM model in MADL. The following properties of an OSM model can be analyzed with common graph algorithms.

- **Strong connectedness** An OSM is a directed graph and must be strongly connected in order to ensure that there is no dead-end state and there is no unreachable isolated components. This property can be easily verified with linear time complexity [8] on the state diagram.
- **Conservation of resources** A requirement for an OSM is that it releases all resources when it retires. This property can be verified by traversing all paths from the dormant state back to itself and verify that there exists a *release* or *discard* for every *allocate*. This algorithm also has linear time complexity.
- **Liveness** A deadlock situation may occur when two or more OSMs form a cycle of dependence of resources. We define that state α depends on state β if an outgoing edge of α has an allocation request to the same token manager as an release request on an outgoing edge of β . By checking all state pairs, we can draw a resource dependency graph among all states. A cycle in the dependency graph implies a potential deadlock situation.

B. Algebraic Equivalence Checking

This task partially verifies the equivalence between an ISA specification and an OSM model. We can first extract operation semantics from the operation level of the OSM model. The extracted semantics can then be compared to the semantics in the ISA specification by using the algebraic equivalence checking algorithm [20]. This approach is also applicable to checking the architectural equivalence between the OSM models of two different implementations of the same ISA.

Algebraic equivalence checking is only useful for comparing the semantics of individual operations. It does not tell anything regarding the interaction of the operations in

a pipeline. To further check if the OSM model correctly implements the ISA specification, we need to verify if all pipeline hazards are correctly resolved. In order to determine this, more sophisticated analysis involving the internals of token managers must be performed. This requires an analyzable description scheme of the token managers, which is a subject of our ongoing investigation.

V. CONCLUSIONS

In this paper, we conducted a survey of existing ADLs from the perspective of their architecture models, which we believe is the key element defining the quality of an ADL. We pointed out that flexibility and analyzability are two important aspects for an architecture model. We believe that to facilitate a healthy development of the ADL field, it is important for researchers to focus on developing formal models and to share implementations. Our first effort along this direction, the OSM model and the MADL, are presented in this paper. In addition to the simulation-based verification tasks that are already performed based on MADL, we expect to employ formal approaches in new applications utilizing MADL.

REFERENCES

- [1] H. Akaboshi. *A Study on Design Support for Computer Architecture Design*. PhD thesis, Department of Information Systems, Kyushu University, Japan, 1996.
- [2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, pages 59–67, Feb 2002.
- [3] AXYS Design Automation, Inc. <http://www.axysdesign.com> (current July 2004).
- [4] D. G. Bradlee, R. R. Henry, and S. J. Eggers. The Marion system for retargetable instruction scheduling. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 1991.
- [5] F. Burns, A. Koelmans, and A. Yakovlev. Modelling of superscalar processor architectures with design/CPN. In *Proceedings of Workshop on Practical Use of Coloured Petri Nets and Design*, June 1998.
- [6] K. Cheng and A. S. Krishnakumar. Automatic generation of functional vectors using the extended finite state machine model. *ACM Transactions on Design Automation of Electronic Systems*, 1(1):57–79, 1 1996.
- [7] P. S. Coe, F. W. Howell, R. N. Ibbett, and L. M. Williams. Technical note: A hierarchical computer architecture design and simulation environment. *ACM Transactions on Modeling and Computer Simulation*, pages 431–446, Oct 1998.
- [8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
- [9] CoWare, Inc. <http://www.coware.com> (current July 2005).
- [10] W. L. A. de Oliveira, N. Marranghello, and F. Damiani. Modeling a processor with a Petri Net extension for digital systems. In *Proceedings of the Conference on Design, Analysis, and Simulation of Distributed Systems*, 2004.
- [11] S. A. Edwards. *The specification and execution of heterogeneous synchronous reactive systems*. PhD thesis, University of California at Berkeley, Berkeley, CA, 1998.
- [12] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *IEEE Computer*, pages 68–76, February 2002.
- [13] A. Fauth, J. V. Praet, and M. Freericks. Describing instructions set processors using nML. In *Proceedings of Conference on Design Automation and Test in Europe*, pages 503–507, Paris, France, 1995.
- [14] J. C. Gyllenhaal, W. Hwu, and B. R. Rao. HMDDES version 2.0 specification. Technical Report IMPACT-96-3, University of Illinois at Urbana-Champaign, 1996.
- [15] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proceedings of Design Automation Conference*, pages 299–302, June 1997.
- [16] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of Conference on Design Automation and Test in Europe*, pages 485–490, 1999.
- [17] IEEE Inc., 3 Park Avenue, New York, NY 10016-5997, USA. *IEEE Standard VHDL Language Reference Manual (1076-2000)*, 2000.
- [18] IEEE Inc., 3 Park Avenue, New York, NY 10016-5997, USA. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (1364-2001)*, 2001.
- [19] D. Kästner. *Retargetable Postpass Optimization by Integer Linear Programming*. PhD thesis, Saarland University, Germany, 2000.
- [20] F. C. K.C. Shashidhar, Maurice Bruynooghe and G. Janssens. Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In *Proceedings of Conference on Design Automation and Test in Europe*, pages 1310–1315, 2005.
- [21] R. Leupers and P. Marwedel. Retargetable generation of code selectors from HDL processor models. In *Proceedings of Conference on Design Automation and Test in Europe*, pages 140–144, 1997.
- [22] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *Proceedings of the International Conference on Computer Languages*, pages 90–101, 1998.
- [23] P. Mishra, N. Dutt, and A. Nicolau. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *Proceedings of the International Symposium on System Synthesis*, pages 256–261, Oct 2001.
- [24] W. S. Mong and J. Zhu. A retargetable micro-architecture simulator. In *Proceedings of Design Automation Conference*, pages 752–757, 2003.
- [25] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 4 1989.
- [26] S. Önder and R. Gupta. Automatic generation of microarchitecture simulators. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 80–89, May 1998.
- [27] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of Design Automation Conference*, pages 933–938, 1999.
- [28] W. Qin. <http://people.bu.edu/wqin/pub/madl-1.0.tar.gz> (current Sept. 2004).
- [29] W. Qin. <http://www.princeton.edu/~mescal/madl.html> (current July 2004).
- [30] W. Qin. *Modeling and description of embedded processors for the development of software tools*. PhD thesis, Princeton University, 2005.
- [31] W. Qin and S. Malik. Architecture description languages for retargetable compilation. In Y. N. Srikant and P. Shankar, editors, *Compiler Design Handbook: Optimizations & Machine Code Generation*, pages 535–564. CRC Press, 2002.
- [32] W. Qin and S. Malik. Flexible and formal modeling of microprocessors with application to retargetable simulation. In *Proceedings of Conference on Design Automation and Test in Europe*, pages 556–561, 2003.
- [33] N. Ramsey and J. W. Davidson. Machine descriptions to build tools for embedded systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 176–192, 1998.

- [34] M. Reshadi and N. Dutt. Generic pipelined processor modeling and high performance cycle-accurate simulator generation. In *Proceedings of Conference on Design Automation and Test in Europe*, 2005.
- [35] S. Rigo, R. J. Azevedo, and G. Araujo. The ArchC architecture description language. Technical Report 15, Institute of Computing of the University of Campinas, Brazil, 2003.
- [36] C. Siska. A processor description language supporting re-targetable multi-pipeline DSP program development tools. In *Proceedings of the International Symposium on System Synthesis*, pages 31–36, 1998.
- [37] Target Compiler Technologies N.V. <http://www.retarget.com> (current July 2005).
- [38] J. Teich, R. Weper, D. Fischer, and S. Trinkert. A joined architecture/compiler environment for ASIPs. In *Proceedings of International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 26–33, San Jose, CA, Nov 2000.
- [39] A. Terechko, E. Pol, and J. van Eijndhoven. PRMDL: A machine description language for clustered VLIW architectures. In *Proceedings of Conference on Design Automation and Test in Europe*, page 821, 2001.
- [40] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Architecture description languages for system-on-chip design. In *Proceedings of The Sixth Asia Pacific Conference on Chip Design Language (APCHDL)*, 1999.
- [41] M. Vachharajani, N. Vachharajani, D. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of International Symposium on Microarchitecture*, pages 271–282, Nov 2002.
- [42] A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *Proceedings of Design Automation Conference*, pages 184–188, 2001.
- [43] G. Zimmerman. The MIMOLA design system: A computer-aided processor design method. In *Proceedings of Design Automation Conference*, pages 53–58, June 1979.
- [44] W. M. Zuberek, R. Govindarajan, and F. Suci. Timed Colored Petri Net models of distributed memory multithreaded processors. In *Proceedings of Workshop on Practical Use of Coloured Petri Nets and Design*, June 1998.