Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation

Wei Qin Sharad Malik
Princeton University
Princeton, NJ 08544, USA

Abstract

Given the growth in application-specific processors, there is a strong need for a retargetable modeling framework that is capable of accurately capturing complex processor behaviors and generating efficient simulators. We propose the operation state machine (OSM) computation model to serve as the foundation of such a modeling framework. The OSM model separates the processor into two interacting layers: the operation layer where operation semantics and timing are modeled, and the hardware layer where disciplined hardware units interact. This declarative model allows for direct synthesis of micro-architecture simulators as it encapsulates precise concurrency semantics of microprocessors. We illustrate the practical benefits of this model through two case studies - the StrongARM core and the PowerPC-750 superscalar processor. The experimental results demonstrate that the OSM model has excellent modeling productivity and model efficiency. Additional applications of this modeling framework include derivation of information required by compilers and formal analysis for processor validation.

1 Introduction

Microprocessor modeling is critical to the development of both hardware and software in the design cycle of new processors. With the growth in application-specific processors, there is a strong need for modeling environments based on precise semantics that can be used for rapid generation of detailed processor simulators. Microprocessors are well understood at two levels of abstraction: the instruction-set-architecture (ISA) and the micro-architecture. Accordingly, microprocessor simulation techniques can be classified into instruction set simulation and micro-architecture simulation. Instruction set simulators (ISS) emulate the functionality of programs and are useful for software development. Successful techniques in this category include interpreted simulation, statically-compiled simulation [17] and dynamically-compiled simulation [3]. Micro-architecture simulators are usually built on top of ISSs. Apart from simulating the functionality of programs, they provide performance metrics such as cycle counts and cache hit ratios – important feedback for both hardware and software developers. However, existing simulation techniques in this category are far less mature. To help characterize existing techniques in this category and to highlight the objectives of this research, we identify four important characteristics of a high-quality micro-architecture simulation framework:

Efficient Micro-architecture simulators are typically 3 to 4 orders of magnitude slower than real hardware. For practical simulation of real-world workloads, any possible speedup is highly desirable.

Expressive Microprocessors range from simple non-interlocking data-paths to complex superscalar or multi-threaded architectures. Flexible models capable of capturing a wide range of architectures precisely are essential to the retargetability and usability of the simulation frameworks.

Declarative Rule-based declarative models help expose important architecture properties for model analysis and validation purposes. Such models are also the foundation for simple and clean architecture description languages that are common in retargetable simulation frameworks.

Productive Fast development of microprocessor models enables the parallel development of both software and hardware and helps shorten time-to-market.

In reality, these requirements are often conflicting and are very hard to satisfy simultaneously by a single modeling framework. All existing retargetable simulation frameworks make certain trade-offs and emphasize some aspects at the cost of the others.

An ideal foundation of a high-quality modeling framework is a flexible and formal microprocessor model that is properly balanced in terms of the above characteristics. We address this need through the operation state machine (OSM) formalism described in this paper. We demonstrate that this model is efficient in that the resulting simulation speed is comparable to the popular micro-architecture simulator SimpleScalar [2], expressive in that it can model a wide range of architectures, declarative since it is a formal rule-based model, and productive in that it greatly reduces the simulator development effort.

This paper is organized as follows. Section 2 summarizes related work in the field. Section 3 presents the OSM formalism, and Section 4 applies it to microprocessor modeling, followed by two micro-architecture simulator case studies in Section 5. Other applications of the model are described in Section 6. Section 7 concludes the paper.

2 Related Work

Micro-architecture modeling has been studied for more than two decades. Various modeling frameworks and languages that automate the development of micro-architecture simulators have been reported.

nML [8], ISDL [10], and EXPRESSION [11] take an operation-centric approach by focusing on the execution of individual operations that comprise an instruction. These architecture description languages (ADL) automate the generation of code generators and instruction set simulators. However, detailed pipeline control-path specification is not explicitly supported by these ADLs. As a result, implicit control-path templates have to be used in order to synthesize micro-architecture simulators, which significantly limits the flexibility of these languages.

The alternative to the operation-centric model is the hardware-centric model. MIMOLA [21] is a computer design language based on the discrete event (DE) model of computation (MOC) and can model both combinational and sequential digital logic. However, it is not suitable for complex microprocessor modeling purposes since the abstraction level of the models is low and thus the simulation speed is very slow. HASE [4] is another architecture modeling environment based on the DE MOC. SystemC [9] is a C++ library supporting the same MOC and has been used for microprocessor modeling [15]. Both HASE and SystemC are based on the C++ programming language for module development and have improved efficiency and productivity compared with MIMOLA. Asim [7] is a microprocessor modeling framework based on cycle-driven simulation, which specializes the DE domain by aligning all events on clock boundaries, thus providing faster simulation speed. Liberty [20] is based on the heterogeneous synchronous-reactive MOC [6] and allows for intra-cycle communication. Both Asim and Liberty emphasize clean structuring of the modules and their interfaces to enable reuse. In these hardware-centric frameworks, hardware execution is explicitly modeled and communication between modules is through ports and connections. Explicit port-based communication negatively impacts the resulting simulation speed. Furthermore, the complexity of such hardware-centric models is large. In the SystemC based PowerPC behavioral model [15], more than 200 wires or buses are used to connect 20 modules. Complexity in the specification results in reduced productivity.

Specific attempts have been made to address the complexity issue in hardware-centric approaches. UPFAST [16] abstracts the ports and connections away to improve productivity as well as efficiency for synthesized simulators. However, since all pipeline hazards need to be resolved explicitly by the user, for superscalar processors with complex control, the modeling productivity is still not ideal.

LISA [18] is a pipeline description language. It uses the L-chart formalism to model the operation flow in the pipeline and simplifies the handling of structure hazards. Data hazards and structure hazards induced by resources other than the pipeline stages still need to be explicitly resolved by users in the C language. Therefore, the productivity is not significantly different from that of the UPFAST system. The flexibility of LISA is also limited by the L-chart formalism. No LISA based model for out-of-order architectures has been reported.

BUILDABONG [19] is an architecture and compiler design environment based on the abstract state machine (ASM) [1] formalism. Since it models microprocessors at the register transfer level, the modeling productivity is similar to that of MIMOLA.

SimpleScalar [2] is a tool-set with significant usage in the computer architecture research community. Its micro-architecture simulators have very good performance. However, the model has only limited flexibility through parameterization. When retargeting SimpleScalar to a new microprocessor, programmers have to sequentialize the concurrency of hardware in ad-hoc ways that can be slow and error-prone.

3 Operation State Machine Model

Microprocessor specifications commonly partition their descriptions into two fundamental layers: the operation layer including the semantics and the timing of the operations, and the hardware layer including various hardware units. Most of the existing modeling frameworks focus on one layer or the other. Although a few do take a balanced view of both, they have limited flexibility since the layers are modeled in ad-hoc ways. In order to distinguish the two layers clearly and to formalize the complex interactions between the two, we propose the new OSM model as a flexible and formal model for the operation layer and the interface between the two layers.

In the OSM model, machine operations are modeled as state machines whose states represent the execution steps of the operations. A director [13] coordinates the state transitions of the state machines. Each state transition may require one or more tokens, representing specific structure or data resources, for execution. The hardware layer is represented by token managers, which control the use of tokens by the operations. The state machines communicate with the hardware layer through token transactions defined by a formal language. The organization of the OSM model is illustrated in Figure 1.

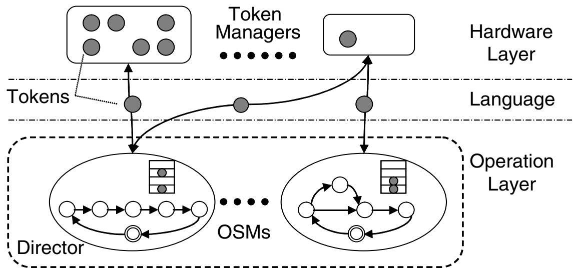

Figure 1. The OSM model

3.1 Operation State Machines

Figure 2 shows an example state machine. The vertices define the states and the edges define the possible transitions among the states. Each edge of the state machine is associated with a guard condition defined in the language. A state transition along an edge can happen only if the corresponding condition is satisfied. The multiple outgoing edges from a state have static priorities, and when more than one outgoing edges are simultaneously satisfied, execution proceeds along the edge with the highest priority. This models execution with multiple paths commonly seen in superscalar processors. Each state machine contains a token buffer for allocated resources. It also has an initial state I in which the token buffer is empty.

OSMs do not directly communicate with each other. The only means by which they interact with the environment is through token transactions.

3.2 Tokens and Token Managers

Microprocessor operations require structure and data resources for their fetching, issuing, execution and completion. In the OSM model, we model the resources as tokens. A token manager manages one or more closely related tokens. It can grant a token to, or reclaim a token from an OSM upon request. Token managers may check the identity of the requesting OSMs when making decisions.

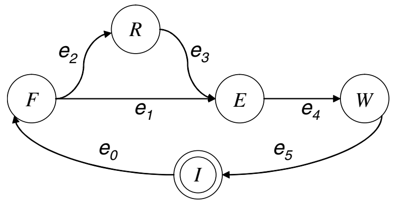

Figure 2. An operation state machine example

3.3 Language

To formalize the token transactions between the state machines and the token managers, we define the language as the four basic primitive transactions listed below.

Allocate An OSM may request a token from a manager by presenting a token identifier. The manager interprets the identifier and maps it to a token. If the token is available to the OSM, the primitive succeeds and the manager will grant the ownership of the token to the OSM. On success, the OSM can either commit the transaction by accepting the token or abandon the transaction. This primitive is used to model the transactions of exclusive resources. Most structure resources in a microprocessor are exclusive.

Inquire An OSM may inquire about the resource unit represented by a token without the intention to obtain the token. The primitive succeeds if the token is available to the OSM. The primitive is used for non-exclusive resource transactions. One example of non-exclusive transactions is reading the state of a machine register.

Release An OSM may request to return a token to its manager if the token is no longer used, which is the opposite behavior to allocation. If the manager rejects the request, the OSM must retain the token. Otherwise, the primitive succeeds and the OSM can either commit the transaction by releasing the token or abandon the request and keep the token.

Discard An OSM can discard a token. This transaction requires no permission from the token manager and always succeeds. Discard can be used when the OSM is reset.

In Section 3.1, we mentioned the condition associated with each edge of the OSM. It is defined as the conjunction of a set of primitives. *A condition is satisfied only if all its primitives succeed simultaneously*. If a condition is satisfied, the OSM can transition to the next state along the edge and commit all transactions of the condition simultaneously. If all primitives do not succeed, the condition is not satisfied and all transaction requests are abandoned. Disjunction is not supported by the language. It can be realized through parallel edges between two states.

3.4 Director

An OSM model may contain multiple state machines and token managers. At each control step, the state machines voluntarily send token transaction requests to the token managers and change their states if possible. The director coordinates the OSMs and ensures that the behavior of the model is deterministic.

To avoid the possible non-determinism when multiple OSMs are competing for the same resource, the director ranks each OSM at the beginning of each control step. The ranking may be based on the status and the identity of the operations represented by the OSMs. An OSM with a higher rank has higher token transaction priority than an OSM with a lower rank. The scheduling rules for individual OSMs are listed below.

- State transition occurs at most once for each OSM at each control step.
- State transition occurs as soon as an outgoing edge has satisfied condition.
- State transition along a higher priority edge is preferred.

Conforming to these rules, we chose a simple sequential scheduling algorithm shown in Figure 3, which has low implementation overhead and guarantees deterministic OSM behavior. The OSMList is first sorted according to the ranks. The OSM with the highest rank is scheduled first so that its token transaction requests can be served first. The EdgeList contains the outgoing edges of the current state of the OSM. It is sorted by the static edge priorities from the highest to the lowest. When an OSM changes its state and commits its primitives, it is removed from the list so that it will not be scheduled for state transition again in the current control step. During its state transition, an OSM may free resources useful to its dependent OSMs that have higher ranks and have been blocked on the resources. To allow these OSMs to obtain the resources, we restart the outer-loop from the remaining OSM with the highest rank. When the OSMList becomes empty or when no more OSM can change its state, the director stops.

In general, scheduling deadlock may occur in the model if cyclic resource dependency involving two or more OSMs exists. Deadlocks are considered pathological situations and the director will abort in such cases. In OSM based microprocessor models, such cyclic dependency implies a cyclic pipeline, which occurs only under faulty situations. Therefore, this property of the director does not affect the usability of the model.

```
Director::control_step()
{
    updateOSMList();
    OSM = OSMList.head; // head.next is the first
    while ((OSM=OSM.next)!=OSM.tail) {
        EdgeList = OSM.currentOutEdges();
        foreach edge in EdgeList {
            result = OSM.requestTransactions(edge);
            if (result == satisfied) {
                OSM.commitTransactions(edge);
                OSM.updateState(edge);
                OSMList.remove(OSM);
                OSM = OSMList.head;
                break;
            }
        }
    }
}
```

Figure 3. The scheduling algorithm

4 Modeling Microprocessors

In the OSM-based modeling scheme, we model a microprocessor in two layers: the operation layer where operations are modeled as OSMs, and the hardware layer where hardware modules interact with each other under the DE MOC. To communicate with the operation layer, each hardware module directly interacting with the operations should implement a token manager interface (TMI). A TMI contains four methods corresponding to the four primitives of the language. The methods implement the resource management policies of the hardware module and will be activated upon incoming token transaction requests. TMIs do not communicate with each other directly.

Since operations flow in a microprocessor synchronously, the control steps of the OSM model are synchronized with the clock edges of the hardware layer. Depending on the implementation, the interval between two control steps may correspond to either a clock cycle or a phase. In the simulation kernel, the OSM MOC is embedded inside the DE scheduler, as shown in Figure 4.

During the interval between two control steps, the hardware modules communicate with one another and exchange information with their TMIs. At the end of each phase or clock, an OSM control step is activated. Since no event should be introduced by the control step directly, from the viewpoint of the DE domain, the control step finishes in zero time.

```
nextEdge = 0;
eventQueue.insert(new clock_event(nextEdge));
while (!eventQueue.empty())
{
    event  = eventQueue.pop();
    if (event->timeStamp >= nextEdge) {
        director.control_step();
        nextEdge += regularInterval;
        eventQueue.insert(new clock_event(nextEdge));
        break;
    }
    event->run();
    delete event;
}
```

Figure 4. The simulation kernel

We illustrate this modeling scheme with the example of a simple 5-stage pipelined processor shown in Figure 5. Operation flow in such a pipeline can be modeled by the L-chart used in LISA [18]. In comparison, we will show that OSM can cover not only the operation flow, but also pipeline control behaviors under the same model.

We model each operation as an OSM shown in Figure 6. Since multiple operations can flow in the pipeline simultaneously, multiple OSMs exist in the processor model. The states of the OSMs correspond to the status of the operations in the pipeline and the edges correspond to the possible paths of the operation flow. In this simple example, an operation can go through the F, D, E, B and W states which correspond respectively to the pipeline stages of fetch, decode, execution, buffer and write-back. The initial state I corresponds to the case when the OSM is unused.

We then provide TMIs for the 5 pipeline stages. Each TMI controls one token indicating the occupancy of the corresponding stage. The register file contains a TMI m_r, which manages a set of value tokens corresponding to the registers, and several register-update tokens whose usage will be described below. Since the memory subsystem does not communicate with the OSMs directly, it is modeled purely in the hardware layer.

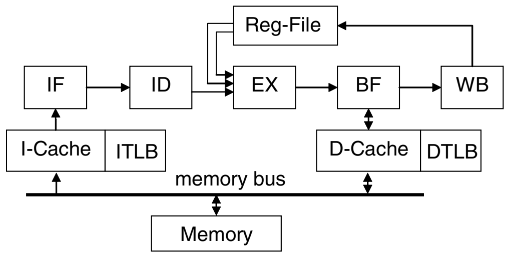

Figure 5. A 5-stage pipelined RISC processor

At the beginning of a clock cycle, the OSMs in state I post token allocation requests to the fetch manager m_f, trying to enter state F. If the fetch stage is available, m_f will grant the token to the requesting OSM with the highest rank, say α. α will then advance to state F, while its competitors will remain in state I. Holding the fetch token allows α to access the instruction word when it is fetched from the memory. α can then decode the instruction and initialize all its allocation and inquiry identifiers.

In the next cycle, α will send a release request to m_f and an allocation request to the decode manager m_d, trying to enter the decode stage. If both requests are satisfied, α will accept the decode token, advance to state D and release the fetch token, allowing another operation to enter the fetch stage in the same control step.

In the following cycle, α will send a release request to m_d and an allocation request to m_e, asking for the permission to enter the execution stage. Meanwhile, it will send two inquiries to m_r about the value tokens corresponding to its two source registers. It will also send an register-update token allocation request to m_r, asking for the permission to update its destination register. If all requests are satisfied, α will advance to state E.

In state E, α will compute the result based on the source register values. In the next two cycles, α will perform a series of token transactions with m_e, the buffer manager m_b and the write back manager m_w and go through states B and W. In the cycle after its arrival in W, α will release the write-back token to m_w and the register-update token to m_r with the updated computation result, and then return to state I. From there, it will send an allocation request to m_f again, waiting to start the life cycle of another operation.

So far we have described how to model the basic operation flow in a pipeline. Below, we explain how common control behaviors can be modeled.

Structure hazard As has been described, a pipeline stage contains a token manager interface controlling one occupancy token. Since the token can be allocated to only one operation at a time, at most one operation can occupy the pipeline stage at a time. Structure hazards are therefore resolved.

Data hazard The register file manager m_r serves to resolve data hazards. In the above example, α retains a register-update token from state E to state W. During this period, the following OSMs depending on the register should fail on inquiring about the value token corresponding to the same register and will stall at D. Data dependency is thus preserved. If the processor supports bypassing, we can create another manager working as the bypassing logic. OSMs can inquire either m_r or the bypassing manager for source operand availability.

Variable latency Variable latency occurs very often in processors. For instance, the latency of cache access can vary depending on whether the access hits or misses. We utilize the release transaction to model the variable latency. Suppose that an instruction cache miss occurs when OSM α is in state F. When α tries to proceed to state D, the fetch manager m_f can

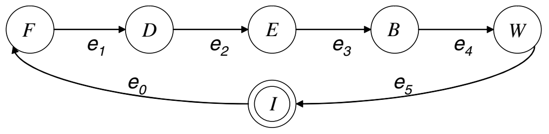

Figure 6. OSM of a pipelined processor

turn down its token release request until the cache access is finished. α will have to stall until the request is satisfied.

Control hazard To accurately model control hazard, we create a reset manager m_{reset}. We also augment the OSM in Figure 6 by adding reset edges from state F and D to state I. Each reset edge is assigned a higher static priority than normal edges and contains an inquiry request to m_{reset} and one or more discard primitives. m_{reset} will reject inquiry requests from normal OSMs so that their behaviors are not changed.

Now suppose a branch mis-prediction occurs and several OSMs speculatively leave state I. After the mis-predicted branch operation leaves state E where the branch condition is resolved, the execution stage will notify the fetch stage to alter the program counter. It will also notify m_{reset} so that m_{reset} will accept inquiry requests from the speculative OSMs. At the next control step, the speculative OSMs will execute along their reset edges, discard their tokens and return to I. In other words, the speculative operations are killed.

In summary, the OSM model can easily model common pipeline behaviors in the form of state transitions and token transactions. In an OSM-based microprocessor model, the operation flow paths, execution timing and resource consumptions are captured in the OSMs in the operation layer; while structure and data resources and their management policies are captured in the TMIs in the hardware layer. The clean separation of the two layers helps orthogonalize design considerations and significantly improves modeling productivity. It also enables object-oriented modeling in both layers.

Compared with a microprocessor model purely in the hardware domain, a model utilizing the OSM can be much simpler and more efficient. An OSM can collect arbitration results from various TMIs and can decide the transition of state by itself according to the conjunction of the transaction request responses and the static edge priorities. The effect of this is two-fold: the operation forwarding paths between pipeline stages in the hardware layer are replaced by the state transition edges in the OSM, and a large portion of the pipeline forwarding and steering control semantics is now encoded in the conditions and priorities of the OSM. As a result, the OSM model greatly simplifies the modules and their interconnections in the hardware layer. In the above example, modules such as the register file, the decode stage and the write back stage need no interconnection with others and contain no more code than their TMIs. Such reduced complexity of the hardware modules improves both the modeling productivity and the efficiency of the resulting simulator. Furthermore, TMIs of the same nature are very much alike and code reuse can be exploited to a great extent. The specification of OSMs is purely declarative and can therefore be automated through the use of description languages.

5 Case Studies

To demonstrate the advantages of the OSM model, we conducted case studies with two popular microprocessors, the StrongARM [5] core and the PowerPC 750 [14] core.

We first implemented an OSM library as the foundation of both simulators. In the implementation, the director ranks the OSMs according to their ages, i.e. the order in which they last leave state I. Since no senior operation will depend on junior operation for resources in both processors, the director does not need to restart the outer-loop in Figure 3 when an OSM transitions its state.

We based both models on existing ISSs, which are capable of simulating user-level ELF binaries. We utilized cycle-driven simulation for the hardware layer. The development of both micro-architecture simulators, including implementing the OSM library, studying the processor specifications, coding and validation, was completed in two man-months – a testament to the increased productivity of this model.

5.1 StrongARM

The StrongARM [5] is a five stage pipelined implementation of the Advanced RISC Machine (ARM) architecture. It is integrated in Intel's SA-1100 chip and is widely used in modern hand-held devices. The structure of the StrongARM is similar to the pipeline in Figure 5, but it includes forwarding paths and a multiplier. We implemented TMIs for the pipeline stage modules, the combined register file and forwarding paths module, and the multiplier module. The caches, the TLBs and the bus interface unit do not interact directly with operations and do not need any TMI.

The resulting simulator runs at the average speed of 650k cycles/sec on a P-III 1.1GHz desktop. In comparison, the ARM simulator of the SimpleScalar tool-set runs at 550k cycles/sec on the same machine. To ensure the accuracy of the model, we validated the simulator against an iPAQ-3650 PDA containing a SA-1100. We used 40 small kernel loops to diagnose timing mismatches between the model and the real processor. We then compared the performance metrics between the two by running the largest applications from the MediaBench [12] benchmarks. The iPAQ run time was collected by the Linux utility *time*. The results shown in Table 1 validate the accuracy of the model. We attribute the remaining differences to the resolution of the *time* utility and the interpretation of system calls in the ISS. Since all details of the memory subsystem were not available, the memory modules may have also contributed to the differences.

5.2 PowerPC 750

PowerPC 750 [14] is a dual-issue out-of-order superscalar processor. It has a 6-entry fetch queue, 6 function units with 6 independent reservation stations, 5 register files with renaming buffers, and a 6-entry completion queue. To faithfully model all these units, we created 19 TMI-enabled modules. The memory subsystem, the branch history table and the branch target instruction cache of PowerPC 750 are implemented purely in the hardware layer.

PowerPC 750 utilizes the reservation stations to increase its issue rate. When an instruction is dispatched from the fetch queue, it will check if all source operands and the function unit are available. If this is the case, it will enter directly into the unit. Otherwise, it will enter the reservation station of the unit. Such typical superscalar behavior cannot be modeled by L-chart, but it can be easily modeled by an OSM as shown in Figure 2.

We validated our PowerPC 750 model against the SystemC based model [15]. We tested a benchmark mix from MediaBench and SPECint 2000 and found that the differences in timing are within 3% in all cases. The remaining differences are mainly due to subtle mismatches in interpreting the micro-architecture specifications between the two models. The average speed of the OSM

benchmark	iPAQ(sec)	Simulator(sec)	difference
gsm/dec	0.59	0.572	3.05%
gsm/enc	1.69	1.647	2.54%
g721/dec	2.23	2.205	1.12%
g721/enc	2.31	2.293	0.74%
mpeg2/dec	14.85	14.55	2.02%
mpeg2/enc	32.85	32.38	1.43%

Table 1. StrongARM model comparison

parts	SA-1100	PPC-750
Modules with TMI	279	930
Modules without TMI	184	481
Decoding and OSM init.	2,130	2,963
Miscellaneous	439	630
Total	3,032	5,004

Table 2. Source code line numbers

model is 250k cycles/sec on a P-III 1.1GHz desktop, 4 times that of the SystemC model.

Finally, we show the source code line counts for both simulators in Table 2 as a measure of productivity. In comparison, the micro-architecture simulator portion of the SimpleScalar ARM target contains 4,633 lines of C code and the SystemC based PowerPC model contains about 16,000 lines of C++ code. This does not include the instruction semantics simulation portion, comments and blank lines in all cases. About 60% of the source code in Table 2 is dedicated to instruction decoding and OSM initialization, which can be automatically synthesized through the use of an architecture description language. Most hardware modules and their TMIs were reused across the two targets.

6 Discussions

The case studies demonstrate the flexibility of OSM in modeling scalar and superscalar architectures. Since Very Long Instruction Word (VLIW) architectures have simpler pipeline control, they can be easily modeled by OSM as well. The OSM model can also be used for the modeling of multi-threaded (MT) architectures. When modeling MT with OSM, each OSM carries a tag indicating the thread that it belongs to. The tags are used as part of the identifiers for token transactions and may contribute to the ranking of the OSMs.

The OSM model is highly declarative. The state machines in the model can be expressed in the ASM [1] formalism. Thus it is possible to extract model properties for formal verification purposes. Operation properties such as the operand latencies and reservation tables can also be extracted and used by a retargetable compiler during operation scheduling.

7 Conclusions and Future Work

In this paper, we propose the OSM model as a flexible formalism for micro-architecture modeling. We demonstrate that the OSM model is efficient, expressive, declarative and productive. It is suitable for use as the foundation of a retargetable simulator generation framework for a wide range of architectures including scalar, superscalar, VLIW and MT architectures.

The next step in our research is to devise an architecture description language based on the OSM model and to implement a retargetable microprocessor modeling framework combining the model and a general purpose hardware modeling environment.

8 Acknowledgments

This work is supported by the Gigascale Research Center sponsored by DARPA/MARCO. We thank the Liberty group led by Prof. David August for discussions, Prof. Margaret Martonosi for providing experimental equipment, and Prof. Stephen Edwards, Keith Vallerio and Subbu Rajagopalan for their generous help in improving the paper.

References

[1] E. Börger. The Origins and the Development of the ASM Method for High Level System Design and Analysis. *Journal of Universal Computer Science*, 8(1):2–74, 2002.

[2] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Department of Computer Science, University of Wisconsin-Madison, June 1997.

[3] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[4] 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.

[5] Digital Equipment Corporation, Maynard, MA. *Digital Semiconductor SA-110 Microprocessor Technical Reference Manual*, 1996.

[6] S. A. Edwards. *The specification and execution of heterogeneous synchronous reactive systems*. PhD thesis, University of California at Berkeley, Berkeley, CA, 1998.

[7] 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.

[8] 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.

[9] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Boston, MA, 2002.

[10] 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.

[11] 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.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, Dec 1997.

[13] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 1, July 2002.

[14] Motorola Inc. *MPC750 RISC Microprocessor User's Manual*, 1997.

[15] G. Mouchard. `http://www.microlib.org`, 2002.

[16] 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.

[17] S. Pees, A. Hoffmann, and H. Meyr. Retargetable compiled simulation of embedded processors using a machine description language. *ACM Transactions on Design Automation of Electronic Systems*, 5(4):815–845, Oct 2002.

[18] 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.

[19] J. Teich, R. Weper, D. Fischer, and S.Trinkert. A joined architecture/compiler environment for ASIPs. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, San Jose, CA, Nov 2000.

[20] M. Vachharajani, N. Vachharajani, D. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *International Symposium on Microarchitecture*, Nov 2002.

[21] G. Zimmerman. The MIMOLA design system: A computer-aided processor design method. In *Proceedings of Design Automation Conference*, pages 53–58, June 1979.