High-Level Architecture Modelling Assisting the Processor Platform Development, Debugging and Simulation

Martin Zabel, Thomas B. Preußer, Rainer G. Spallek

Department of Computer Science, Technische Universität Dresden, Germany

{zabel,preusser,rgs}@ite.inf.tu-dresden.de

Abstract

The processor simulator PROSIM introduces an alternative debugging and simulation technique beyond VHDL / Verilog simulation and an on-chip logic analyzer. The simulated architecture model of our embedded bytecode processor SHAP – described in a high-level architecture description language – provides critical insight into the running system for debugging, profiling as well as for feature exploration and evaluation.

1 Introduction

The development of generic architecture-level software toolkits sparkled in the early 1990s. They typically aim at the automatic generation of a compiler, an assembler and a simulator to aid the design space exploration (DSE) as well as the development of software for the chosen architecture. As they allow the cost-effective evaluation of a wide range of platforms without the need for a designated toolkit for each architecture, they are an integral part of the application-specific processor development process.

Generic toolkits must be instantiated for a specific architecture before employment. Their retargation is achieved by an architecture description language (ADL), which can take a structural (Mimola), behavioral (nML, ISDL) or mixed (EXPRESSION, LISA) approach to the description of the target architecture. Either approach has its specific advantages: Structural descriptions favor the synthesis support over simulation speed whereas purely behavioral descriptions sacrifice synthesizability for increased simulation speed. Mixed approaches usually attempt to reduce the verbosity of structural descriptions without constraining its wide applicability.

While our own software toolkit was the first to support the dynamic reconfiguration of the instruction set [10], this paper explores its capabilities as architectural debugging tool for a more conventional, but still complex, pipelined architecture. In particular, it describes the approach to modelling

and simulating the SHAP Microarchitecture [12] in the course of its development. This will be preceded by a concise overview over other and our approaches to architectural modelling and significant design choices in the simulator implementation.

2 Related Work

Prominent examples for behavioral ADLs are nML [2] and ISDL [3]. Both permit the concise description of instruction sets. Structural constraints as must be communicated to a scheduler are, however, difficult to capture. While ISDL allows the specification of boolean constraints to be met by a schedule, the nML extension Sim-nML introduces abstract resources to specify conflicts.

Mimola [1] offers a structural approach that is too low a level of abstraction for a software toolkit. Although structural descriptions are usually very detailed and precise, they lack brevity and intuitivity.

A feasible compromise between structural and behavioral descriptions is established by mixed approaches as by EXPRESSION [4] or LISA [13, 5]. Both are capable of modelling VLIW as well as pipelined architectures. EXPRESSION, however, lacks the support for a hierarchical instruction set description that is often a great aid for eliminating redundancy. LISA, on the other hand, features the support of arbitrary C code within behavioral description blocks. While this certainly increases flexibility, it, however, also constrains toolkit applications other than the simulator from automatically deriving the described behavior as it is hidden inside a black box.

3 ADL Design

Similar to EXPRESSION and LISA, we chose a mixed behavioral and structural approach to our ADL. Whereas memory hierarchy and pipeline are described structurally, the instruction set is modelled on a behavioral level. In matters of syntax, the designed ADL leans rather towards LISA using a C-like notation.

The architecture description is divided into three main sections describing the memory hierarchy, the pipeline structure and the instruction set in this order. Optionally, these sections may be preceded by another one declaring references to external libraries.

The description of the memory hierarchy is straightforward as it enumerates the memory components and parametrizes their attributes in a fashion much like EXPRESSION. These attributes are not limited to the physical layout of the memory components but also include functional parameters as the arbitration model of a bus or the delay imposed on accesses.

The attribute `Alias` as can be found in the specification of a register file is especially valuable for the readability of the descriptions of the instruction set and possibly the pipeline. By assigning names to individual registers, their special purpose like e.g. program counter, stack pointer and link register is emphasized.

The description of the pipeline defines the available pipeline stages and that part of their behavior that is independent from the executed instruction context. Typically, the instruction fetch and decode as well as the default flow of instruction contexts through the pipeline are defined here. The structural composition of a pipeline is, however, not necessarily static. In contrast to, e.g., LISA, instructions may override their routing even depending on the data they process. Additionally, an instruction context may be split and routed to multiple pipeline stages, which allows the modelling of multi-issue VLIW architectures as illustrated in Fig. 3.

Structural hazards occurring in the execution of an instruction context are detected automatically and hinder its flow to the next pipeline stage. Typical examples for such hazards are pipeline stages

```
Bus Mem {   // Memory Bus
  // Smallest addressable unit: 1 Byte
  Unit: 8;

  // Accesses, Default: 1 Byte
  Access [byte] {
    byte: 1;
    half: 2;
    word: 4;
  }

  Arbitration: Slotted(0, 1, 0);
  // Single outstanding read,
  //   no buffering

  // Bus Devices
  Ram Mem {           // RAM
    Base: 0x000000;   // Base Address: 0
    Size: 0x800000;   // Size: 8 MiByte
    Endian: Big;      // Big Endian
    Delay:  Const(0, 0, 0); // Fast SRAM
  }
}

RegFile R { // Register File
  Size:  16;  // 16 Registers of
  Width: 32;  // 32 Bit

  rPorts: 2;  // 2 Read Ports
  wPorts: 1;  // 1 Write Port
  xPorts: 0;  // 0 any Ports

  // Alias "R15" to "PC"
  Alias { PC: 15; }
}
```

Figure 1: Sample Memory Hierarchy and Register File

```
ISA {
 Decoder: Mask(32);

 //- Register-Register Instructions
 ALURR "<0:14><6><src1:4><src2:4><dst:4>",
       "$$\tR$src1, R$src2, R$dst" {
   ID {
     req  s1 = R[src1]; // Fetch operands
     req  s2 = R[src2];
   }
   WB { R[dst] = res; } // Write back
 }

 ADD : ALURR "000000" {
   EX { res = s1 + s2; }
 }
 AND : ALURR "000001" {
   EX { res = s1 & s2; }
 }
 SRL : ALURR "000010" {
   EX { res = s1 >> s2(0,3); }
 }
 ...
}
```

Figure 2: Instruction Set Hierarchy

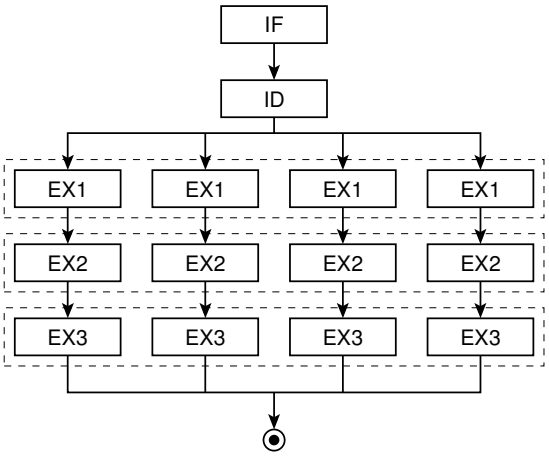
occupied for multiple cycles by the preceding instruction, the exhaustion of the ports available at a register file or an access conflict on a memory bus.

The instruction set is modelled in a hierarchical fashion as is done in nML or LISA. This hierarchy is, however, described in terms of an inheritance tree rather than a formal grammar. An instruction may define a specific behavior for each of the declared pipeline stages. Derived instructions implicitly also inherit the behavior of their super instruction. This is a powerful means to reduce the redundancy of descriptions of hierarchical ISAs as exemplified in Fig. 2.

As may be observed in Fig. 2, there is more to an instruction than just its behavior. The first string following its declaration is forwarded to the decoder. The one presently used implements a hierarchical mask-and-match algorithm. It is also capable of extracting bit fields from the instruction word into variables of the instruction context, which are thus directly accessible to the behavioral description.

The mandatory decoder string may optionally be followed by a disassembler string, which is likewise forwarded to a disassembler. Although this disassembler is plugged into the simulator, it only serves the user interface and is not used by the simulator itself. The choice of the disassembler, if used at all, does not influence the described behavioral semantics in any way.

Although the behavioral description of the instruction set assumes a pipelined architecture, the description domain can be naturally extended to architectures or descriptions without pipelines by simply letting the pipeline degenerate into one with a single stage.




```
// Assume byte addressing with
// 1 dword = 8 bytes = 64 bits

Pipe {
  IF { // fetch VLIW
    req  ir = Mem[PC].dword;
    => ID;
  }{   // increment PC
    PC = PC + 8;
  }

  ID {
    => EX1; // forward all to EX1

    // split VLIW into 4 contexts
    { decode(ir( 0, 16)); |
      decode(ir(16, 16)); |
      decode(ir(32, 16)); |
      decode(ir(48, 16)); }
  }

  // Stages with 4 parallel contexts
  EX1 [4] { => EX2; }
  EX2 [4] { => EX3; }
  EX3 [4] {}
}
```

Figure 3: Sample 4-Issue-VLIW-Architecture

Besides the dynamic routing of instruction contexts through the pipeline, also a dynamic instruction set is supported. This feature allows multiple instructions to share a binary encoding. They are only disambiguated at run time by the currently active decoding context. Such contexts are modelled as sibbling branches of the instruction hierarchy, which may be switched by appropriate instructions. Thus, architectures with a (partially) reconfigurable instruction set can be modelled.

An important feature of the designed ADL is that the behavioral descriptions can only be composed from a fixed set of operational statements. The behavioral semantics can, thus, not be blurred by vague code in some programming language or even references to external functions. Nevertheless, the inclusion of external libraries is supported. These can, however, only be used to provide custom implementations for structurally isolated models as the arbitration of a bus or the decoding of instructions.

4 Simulation

The architectural description is interpreted by a simulator implemented in C++. In addition to the architectural description, a boot image containing an executable assembled binary must be provided. An ELF frontend enabling the integration with standard compiler toolchains is available.

The choice of an interpreted simulation is a tradeoff, which reduces design cycle times at the expense of simulation speed. It is justified in the context of architectural development where errors in the architectural design rather than within long-running applications need to be identified. Nonetheless, we have taken several measures to minimize this penalty through the design of the simulation engine. Thus, the architectural description is fully parsed, and the components of the architecture are instantiated from parametrized compiled classes before simulation. Behavioral blocks are translated into an executable parse tree with all symbolic references resolved. The simulation engine itself is freed from any expensive string handling. The names and identifiers used in the architectural description

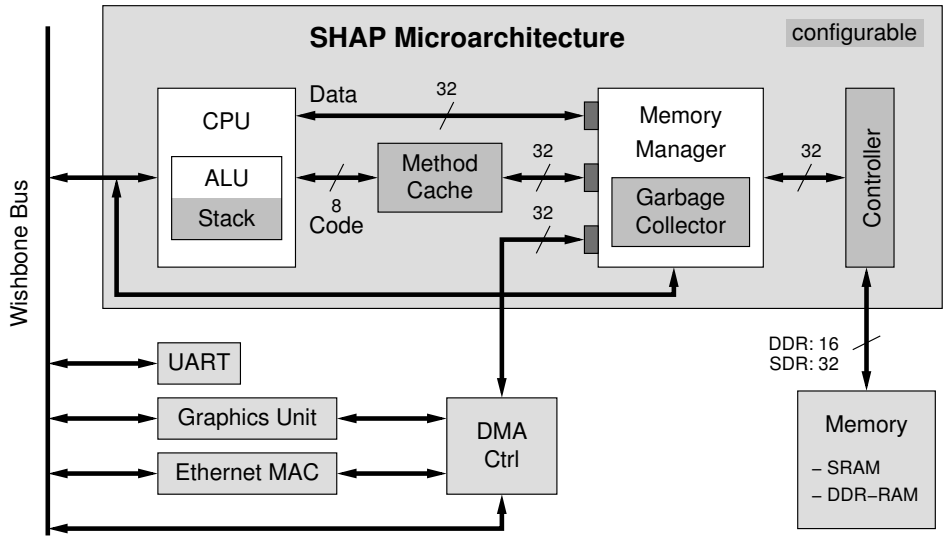


Figure 4: Overview of the SHAP Microarchitecture

are merely maintained for the purpose of user communication and are used by the graphical user interface [8]. The latter is allowed to inspect the architectural state, to register listeners for architectural events, to watch data expressions, and to set state- and event-based breakpoints. In consequence, this enables the cycle-accurate debugging of the architecture.

Special care is taken to permit the fast operation with data words of arbitrary sizes so as not to impose undue constraints on the simulatable architectures. For this purpose, we have designed a `BitVector` class, which directly implements arithmetic and logic operations as well as slicing just as supported by the behavioral description. The internal representation of `BitVector`s takes two different shapes depending on their sizes. It either directly contains the vector's value when it fits into two words of the simulating host or it keeps a reference to a heap-allocated data area otherwise. This ensures generality while also maintaining a fast processing of short vectors – usually of a length of up to 64 bits. For longer vectors the relative costs of the extra memory indirection are less severe. Additionally, expensive data block duplication for long vectors has been minimized through a copy-on-write strategy backed by reference counting.

We have verified and evaluated our ADL and simulator with models of the MIPS-II and ARM7v4 processor cores. As a more complex application, TADL models have been used for simulation, debugging and profiling of the coarse-grain reconfigurable array ARRIVE [6] as well as SHAP.

5 Simulating and Debugging SHAP

5.1 Overview of SHAP

SHAP is an implementation of the Java Virtual Machine (JVM) Specification [7]. To support real-time applications, significant parts of the JVM are implemented in hardware such as the dynamic stack and thread management, the concurrent CPU-independent memory management including garbage collection (GC), and the preemptive scheduling [12]. Furthermore, the SHAP JVM supports commonly used object-oriented features, such as automatic memory management, structured exception handling and multiple inheritance through interfaces. Communication with the outer world is established by I/O devices connected through the well-known Wishbone bus [9]. An overview of the SHAP Microarchitecture is given by Fig. 4.

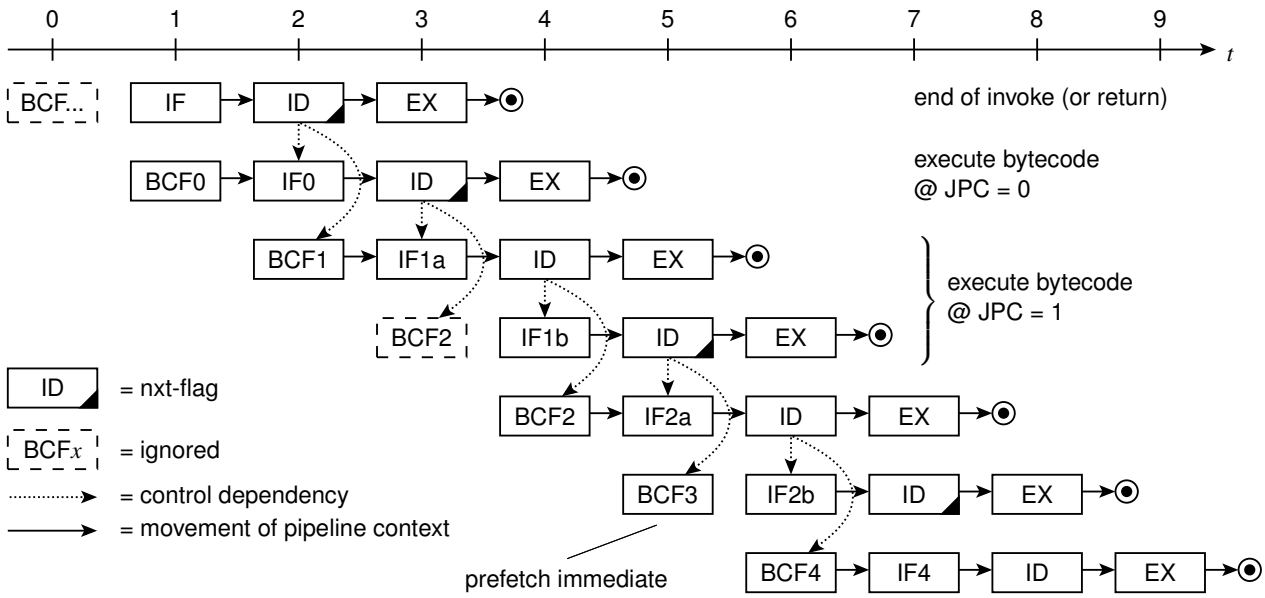


Figure 5: The real 4-stage SHAP pipeline in action.

The Java bytecode is executed natively by the SHAP Microarchitecture using a microcode approach. Complex bytecodes are split into simpler instructions during execution, so that the hardware consumes less chip-space and is easier to maintain. The microcode itself is written in assembler.

Another important part of SHAP is the SHAP Linker [11] which transform the Java classfiles into the layout used by the microarchitecture. Many optimizations are applied to ensure fast method calls and field accesses in constant time. Nonetheless, the final SHAP file is not a static memory layout. Instead, each class has its own separate section to support dynamic class loading. Furthermore, calls to native methods are replaced by special bytecodes, which trigger the appropriate firmware action.

5.2 The Real Pipeline of SHAP

The Java bytecodes are executed by a 4-stage pipeline consisting of the first stage bytecode-fetch (`BCF`) and a typical 3-stage pipeline for the microcode: instruction fetch (`IF`), instruction decode (`ID`) and execute (`EX`). Even simple bytecodes are mapped to microcode. The corresponding routine might be as short as one microcode instruction. Due to the pipeline, no additional delay is imposed by this method.

The end of a microcode routine, is encoded in the microcode instruction by an additional bit, called the *nxt-flag*. A table approach, storing the end addresses, is not applicable because the routine may be left through different paths, e.g., normal return or throw exception. Instead, the decoded nxt-flag (`ID` stage) directly controls the concurrently executing `IF` and `BCF` stages of the following instructions, as depicted in Fig. 5.

At the beginning, the example shows the start of a method after an invoke bytecode. The same principle applies to return-bytecodes and the first invoke after system boot. At time $t = 2$ the last instruction of the invoke-routine is decoded which has the nxt-flag set. At the same time, control signals are calculated a) to fetch the first microcode instruction (`IF0`) of the first bytecode, and b) to prefetch the following bytecode 1 (`BCF1`). For the latter, the Java program counter (JPC) is pre-incremented to 1. This dependence of control is depicted by dotted arrows.

The first bytecode 0 is already fetched at $t = 1$ (`BCF0`). This is accomplished by setting the JPC early enough, and prefetching all the time. Needlessly fetched bytecodes are simply ignored by the

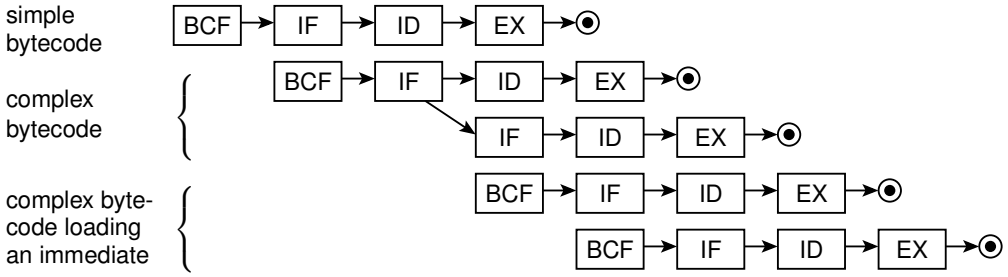


Figure 6: Bytecode execution, if modelling SHAP with 4 pipeline stages.

pipeline. Independent prefetching is required, because the bytecode read from the method cache may change after the JPC has been set, especially during cache-filling on return.

As next, the example also shows execution of bytecodes with more than one microcode. At time $t = 4$, the decoded instruction 1a has no nxt-flag set, denoting, that instruction 1b must be fetched to continue execution of bytecode 1. As the JPC has already be incremented to 2 by instruction 0 with nxt-flag ($t = 3$), the bytecode must be fetched again at the same address ($t = 4$) to fill the pipeline accordingly. The prefetch from the cycle before is discarded.

Whenever a bytecode immediate must be fetched, the same prefetch principle applies as for the nxt-flag. At time $t = 5$, the bytecode 3 is prefetched (`BCF3`) as usual. But now, at $t = 6$ the bytecode is loaded as immediate (and not discarded) by instruction 2a during decoding. The JPC is pre-incremented to prefetch bytecode 4 (`BCF4`) which will be executed next.

To generalize the example:

1. If the nxt-flag is set, then a) pre-increment JPC and prefetch the bytecode following the next one, and b) fetch the first instruction of the next bytecode.

2. Otherwise, increment the microcode program counter (PC) to continue the microcode routine, and

   a) if a bytecode immediate must be fetched, then pre-increment the JPC and prefetch the bytecode following the next one,

   b) otherwise, prefetch from the current JPC again.

5.3 Modelling SHAP with TADL

Processors and their pipelines can be modelled with TADL at different detail levels. To allow comparisons with the hardware implementation, the goal is to keep the timing of the 4 stages also in the TADL model.

But within TADL, control of other (parallel executed) pipeline stages (`BCF` and `IF`) from a stage (`ID`) is only limited to pipeline *stalls* due to resource conflicts. Here, as described in the previous section, the `ID` stage must *change* the control flow in the other stages. To resolve this issue, several approaches are possible.

The simplest solution is to pack the stages `BCF`, `IF`, and `ID` into one stage in TADL. Depending on the current instruction, instruction fetch and bytecode prefetch can be controlled accordingly. Memory read requests are still used to keep the timing correct and to enable triggering during simulation. The combined stage must be moved to itself, to resolve the memory requests in the following cycle.

Another approach is to also model 4 stages to enhance the readability during simulation. To control the other stages, the instruction is now looked ahead in parallel to the fetch (using a requested memory read as usual) in the previous cycle by a direct read to the memory. If the looked-ahead instruction

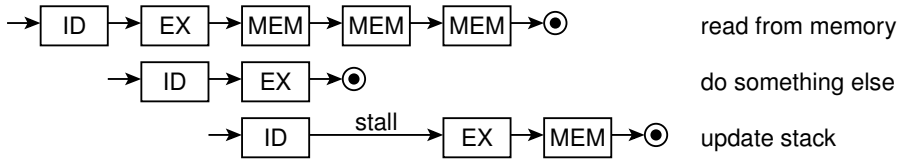


Figure 7: Timeline of memory command's concurrent execution in the SHAP pipeline.

has the nxt-flag set or includes loading of a bytecode immediate, then the instruction context is moved from IF to ID, only. Otherwise, the context is moved to both IF and ID, as depicted in Fig. 6. In the first case, the BCF stage is executed, because it can be moved to IF, and the JPC is incremented. Furthermore, the looked-ahead instruction must be stored in a special register, so that in the next cycle, the IF stage can evaluate the instruction currently decoded in the ID stage. (As no context is moved from IF to IF, no information can be transferred this way.) In the latter case, BCF is stalled. The next bytecode fetch takes place, when the end of the routine has been reached.

Irrespective of this issue, additional pipeline stages are modelled for the concurrently running sub-systems such as memory management (MEM stage), method caching, Wishbone master, and timer, as well as IO devices such as UART and 7-segment display. Whereas, the timer and IO devices communicate with the SHAP core only through registers (timer IRQ, wishbone state registers), the other sub-systems are accessed by moving an instruction context from EX to one of the additional pipeline stages.

Fig. 7 demonstrates this by the example of a memory read. The first instruction reads a value from the heap memory, but this requires several clock cycles which is modelled by traversing the MEM stage three times. The read data is placed in a dedicated transfer register at the end of the second MEM cycle. The additional MEM cycle is required to delay the instruction copying the read value onto the stack (third instruction in this example). Thus, the context of the latter instruction must also be moved from EX to MEM to check if the transfer register is valid. As intended, the EX phase of this instruction is stalled until the context can be moved to MEM.

Due to the additional pipeline stage, commands not accessing the memory subsystem can be placed in between instead of stalling the complete pipeline. The result is a shortened overall execution time of the microcode routine. The inserted commands may, of course, modify the stack. The same principle applies to the Wishbone and the method-cache subsystem.

Last but not least, the EX stage can be traversed several times, to complete complex thread and stack operations, like entering method frames, creating and killing threads. All pipeline stages before this one are stalled accordingly.

5.4 Simulation and Debugging of Java Applications

The TADL model can run the same Java applications as the hardware implementation. While multiple-thread support and method caching are completely modelled, the memory subsystem is reduced to plain object allocation without support of the self-contained automatic garbage collection found in the original SHAP architecture. As to avoid memory exhaustion, nonetheless, the memory space provided in the model is enlarged to 64 MiByte as compared to the 1 MiByte typically sufficient for the real hardware SHAP. The additional emulation of the configuration interface of the original hardware garbage collector by appropriate stub implementations made explicit model-specific changes to the SHAP microcode or Java runtime implementation unnecessary.

The main purpose of the TADL model is to debug the SHAP microcode while executing real-world Java applications. Especially, complex microcode routines such as method invoke/return, type check-

ing, exception throwing, array copying and object cloning are a point of interest. During debugging, the user has full access to the internal state, the stack as well as the heap memory. All values can be changed at run time to correct mistakes without restarting the simulation from beginning. To help debugging, all stack and heap / object accesses are checked in the TADL model, by storing additional information, which is not required for normal operation. Not to confuse with type and array-access checking provided by the JVM and implemented by the microcode itself. Whenever, an illegal state is encountered, the simulator stops and the user can examine the situation. After a hot-fix, the simulation can continue. The TADL description will check the state again.

The TADL model allows stepping at both microcode and Java bytecode level. At the microcode level, single-stepping is supported, which may be combined with data and code breakpoints to step over loops, e.g. during array copying. Method calls are not available in the microcode. At the bytecode level, the conventional features are available, such as step one instruction, step over method invokes, run until method finishes. Of course, arbitrary breakpoints can be defined to stop execution on data, including internal state, and bus accesses.

The TADL model is highly configurable so that the Java program execution can be simulated under arbitrary system configurations. It also supports the disabling of additional features, such as memory checks, to speed up the simulation. Optionally, performance counters profile the memory subsystem, and give numbers about the frequency of the various memory commands. Due to the simple embedded description, the profiling can be enhanced easily in the future. This profiling or evaluation of the microarchitecture provides the main reason for the cycle-accurate SHAP model, so that retrieved knowledge can be mapped onto the real hardware implementation.

As an option, the method cache can be replaced by a direct memory access. But, in contrast to the other configuration options, the execution time of the Java program is shortened, and thus, the simulation time reduced. This mode is often useful during debug sessions when timing-accuracy is not of interest.

The processor simulator can also record a trace of the memory accesses, which is then examined by a cache simulator. The gained results will be used to further improve the SHAP microarchitecture by implementing object caches.

To execute the Java application inside the simulator, the generated SHAP file is transformed by scripts into a memory initialization block for the UART to be used as the incoming data. As a result, the SHAP microcode boots and loads the Java classes as it would in hardware. Additional memory initialization holding the microcode are script-built from the microcode assembler output.

All normal outputs via the UART, or debugging information via the 7-segment display are printed in the messages window of the processor simulator.

The processor simulator requires 194 s to run our test application checking the JVM implementation with an execution time of 2.44 million clock cycles. Object access checks were enabled in the TADL model, otherwise the runtime shortens to 186 s. Simulation times were measured on a 2.67 GHz Intel Core 2 Duo while only one core was utilized by the simulators.

6 Summary

After giving a brief survey on architecture modelling and simulation, the features of our processor simulator are introduced and compared with other modelling languages. Highlights are the flexible pipeline description and the hierarchical instruction set specification. As an application, the SHAP Microarchitecture is introduced. A focus is laid on the SHAP pipeline's hardware implementation with direct influence on the parallel executed pipeline stages. Two possible solutions for modelling the 4-stage pipeline in TADL as well as the concurrently running sub-systems, like memory man-

agement, are explained. Based on the TADL model, typical use cases are explored in the field of microcode debugging and architecture profiling / evaluation. The former is assisted by (optionally deactivatable) TADL code checking the state of the modelled SHAP architecture during simulation of the Java application.

References

[1] S. Bashford, U.Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenhauer. *The MIMOLA Language, Version 4.1*. Technische Universität Dortmund, Department of Computer Science 12, 1994.

[2] M. Freericks. The nML Machine Description Formalism. Technical Report 1991/15, Technische Universität Berlin, 1991.

[3] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *in Proceedings of Design Automation Conference (DAC'97)*, pages 299–302. ACM Press, 1997.

[4] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the European Conference on Design, Automation and Test (DATE)*, 1999.

[5] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.

[6] S. Köhler, J. Schirok, J. Braunes, and R. G. Spallek. Efficiency of dynamic reconfigurable datapath extensions – a case study. In *Reconfigurable Computing: Architectures, Tools and Applications, 4th International Workshop ARC 2008*, volume 4943 of *Lecture Notes in Computer Science*, pages 300–305. Springer-Verlag, Mar. 2008.

[7] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification*. Addison-Wesley Professional, 2 edition, Apr. 1999.

[8] D. Neumann. Entwurf und Implementierung einer generischen Benutzerschnittstelle für einen Prozeßsimulator. Technical report, Technische Universität Dresden, 2004.

[9] OPENCORES.ORG. *Specification for the: WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, rev. b.3 edition, 7 2002.

[10] T. B. Preußer, S. Köhler, and R. G. Spallek. RECAST – design space exploration for dynamic and reconfigurable embedded computing. In *The 2004 International Conference on Embedded Systems and Applications*, 2004.

[11] T. B. Preußer and R. G. Spallek. Java-programmed bootloading in spite of load-time code patching on a minimal embedded bytecode processor. In H. R. Arabnia and Y. Mun, editors, *The 2008 International Conference on Embedded Systems and Applications*, pages 260 –264. CSREA Press, July 2008.

[12] M. Zabel, T. B. Preußer, P. Reichel, and R. G. Spallek. Secure, real-time and multi-threaded general-purpose embedded java microarchitecture. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 59–62. IEEE, Aug. 2007.

[13] V. Zivojnovic, S. Pees, and H. Meyr. LISA Machine Description Language and Generic Machine Model for HW/SW Co-Design. In W. Burleson, K. Konstantinides, and T. Meng, editors, *Proceedings of the Workshop on VLSI Signal Processing*, 1996.