A methodology for validation of microprocessors using symbolic simulation

Prabhat Mishra*

Department of Computer and Information Science and Engineering,
University of Florida, Gainesville FL 32611, USA
E-mail: prabhat@cise.ufl.edu
*Corresponding author

Nikil Dutt

Center for Embedded Computer Systems, Donald Bren School of Information and Computer Sciences, University of California, Irvine CA 92697, USA
E-mail: dutt@uci.edu

Narayanan Krishnamurthy

Qualcomm, San Diego, CA 92121, USA
E-mail: narikrish@gmail.com

Magdy Abadir

Freescale Semiconductor, Austin TX 78729, USA
E-mail: M.Abadir@freescale.com

Abstract: Functional validation is one of the most complex and expensive tasks in the current processor design methodology. A significant bottleneck in the validation of processors is the lack of a golden reference model. Thus, many existing approaches employ a bottom-up methodology by using a combination of simulation techniques and formal methods. We present a top-down validation approach using a language-based specification. The specification is used to generate the necessary reference models for processor validation using symbolic simulation. We applied our methodology for property checking as well as equivalence checking of microprocessors.

Keywords: processor validation; symbolic simulation; microprocessor; hardware synthesis; architecture description language.

Reference to this paper should be made as follows: Mishra, P., Dutt, N., Krishnamurthy, N. and Abadir, M. (2005) 'A methodology for validation of microprocessors using symbolic simulation', *Int. J. Embedded Systems*, Vol. 1, Nos. 1/2, pp.14–22.

Biographical notes: Prabhat Mishra is an Assistant Professor in the Department of Computer and Information Science and Engineering at the University of Florida. His research interests include system level modeling, architecture synthesis, design space exploration, and functional validation of embedded systems. He has a BE from Jadavpur University, India; an MTech from the Indian Institute of Technology, Kharagpur; and PhD from University of California, Irvine - all in Computer Science and Engineering. He is a member of the ACM, IEEE, ACM SIGDA, and ASEE.

Nikil Dutt is a Professor in the Donald Bren School of Information and Computer Sciences at the University of California, Irvine. His research interests include embedded-systems design automation, computer architecture, optimizing compilers, system specification techniques, and distributed embedded systems. He has a BE in mechanical engineering from the Birla Institute of Technology and Science, India; an MS in computer science from Penn State University; and a PhD in from the University of Illinois at Urbana-Champaign. He is a senior member of the IEEE, serves on the advisory boards of ACM SIGBED and ACM SIGDA, and is vice chair of IFIP WG 10.5.

Narayanan Krishnamurthy is a Staff Engineer in the ASIC Design Automation group at Qualcomm. Prior to Qualcomm, he was a Principal Staff Engineer in the Global Strategy and Future Technologies group at Freescale Semiconductor. His research interests include hardware-software design methodologies for microprocessors and SOC platforms, formal methods and verification techniques, and CAD for VLSI systems. He holds a BTech in Instrumentation from the Indian Institute of Technology, India, and an MS and PhD in Electrical and Computer Engineering from the University of Texas at Austin.

Magdy Abadir is the Manager of the Global Strategy, Tools, and Methodology Group at Freescale Semiconductor, and is an adjunct faculty member of the Computer Engineering Department at the University of Texas at Austin. His research interests include microprocessor test and verification, test economics, EDA tools, and DFT. Abadir has a BS in computer science from the University of Alexandria, Egypt; an MS in computer science from the University of Saskatchewan, Canada; and a PhD in electrical engineering from the University of Southern California. He is a senior member of the IEEE.

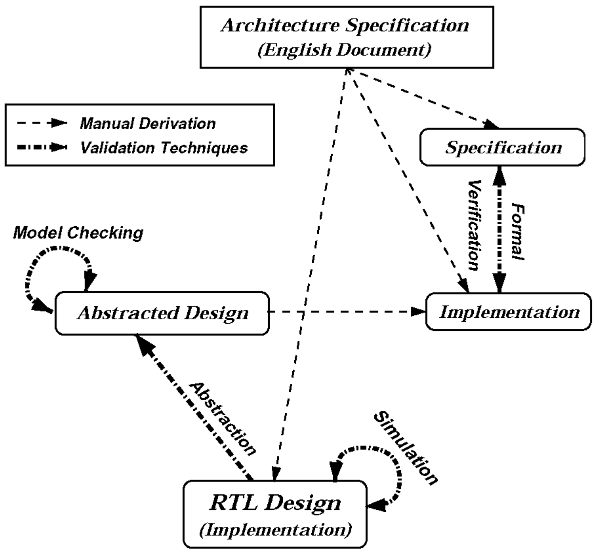
1 Introduction

Shrinking time-to-market coupled with short product lifetimes create a critical need to drastically reduce microprocessor design cycle time. Since verification and design analysis are major components of this cycle time, any effort that improves verification effectiveness and design quality is crucial for meeting customer deadlines and requirements. Design validation techniques can be broadly categorised into simulation-based approaches and formal techniques. Due to the complexity of modern designs, validation using only traditional scalar simulation cannot be exhaustive. Formal techniques do an exhaustive analysis of the design but cannot be applied on the complete design because of the state space explosion. *Symbolic simulation* has proven to be an efficient technique, bridging the gap between traditional simulation and full-fledged formal verification.

Figure 1 shows a traditional architecture validation flow. In current validation methodology, the architect prepares an informal specification of the microprocessor in the form of an English document. The logic designer implements the modules in the register-transfer level (RTL). The *RTL design* is validated using a combination of simulation techniques and formal methods. Simulation is the most widely used form of microprocessor validation: millions of cycles are spent during simulation using random (or pseudo-random) test cases. Model checking is applied on the high-level description of the design abstracted from the RTL implementation. Formal verification is performed by describing the system using a formal language. The specification for the formal verification is derived from the architecture description. The implementation for the formal verification can be derived either from the architecture specification or from the abstracted design. The existing techniques employ a bottom-up approach to validation, where the functionality of an existing processor is, in essence, reverse-engineered from its RTL implementation. Our validation technique is complementary to these bottom-up approaches. We leverage the system architects' knowledge about the behaviour of the processor through Architecture Description Language (ADL) constructs, thereby allowing a powerful *top-down approach* to microprocessor validation.

The contribution of this paper is a top-down validation methodology for microprocessors using automatic reference model generation. We specify the processor using EXPRESSION ADL (Halambi et al., 1999). The reference model of the processor is generated from the ADL specification. A symbolic simulator is used to verify the equivalence between the generated reference model and the implementation (RTL design). We applied our methodology in two validation scenarios: verification of a memory management unit of a microprocessor that is compliant with the PowerPC instruction-set, and verification of a RISC DLX processor.

Figure 1 Traditional bottom-up validation flow

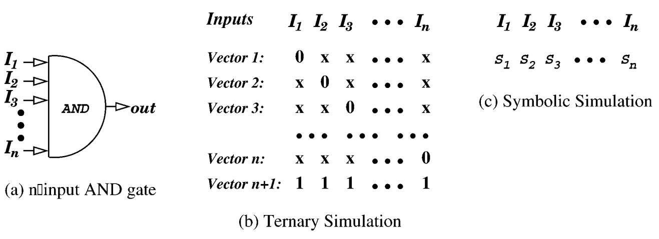


The rest of the paper is organised as follows. Section 2 briefly describes the symbolic simulation technique. Section 3 presents related work addressing validation of microprocessors. Section 4 presents our top-down validation methodology. Section 5 describes our reference model generation technique followed by a case study in Section 6. Finally, Section 7 concludes the paper.

2 Symbolic simulation

Symbolic simulation combines traditional simulation with formal symbolic manipulation (Bryant, 1990). Each symbolic value represents a signal value for different operating conditions, parameterised in terms of a set of symbolic Boolean variables. By this encoding, a single symbolic simulation run can cover many conditions that would require multiple runs of a traditional simulator. Figure 2(a) shows a simple n-input *and* gate. Exhaustive simulation of the *and* gate requires 2^n binary test vectors. However, the ternary simulation (Bryant and Seger, 1990) (uses 0, 1 and x) requires (n + 1) test vectors for the *and* gate. Figure 2(b) shows the vectors: n vectors with one input set to ‘0’ and the remaining inputs set to ‘*x*’, and one vector with all inputs set to ‘1’. Finally, symbolic simulation (Bryant, 1990) requires only one vector using *n* symbols ($s_1, s_2, ..., s_n$), as shown in Figure 2(c).

Figure 2 Test vectors for validation of an *and* gate



Inputs	I_1	I_2	I_3	•••	I_n
Vector 1:	0	x	x	•••	x
Vector 2:	x	0	x	•••	x
Vector 3:	x	x	0	•••	x
	•••	•••	•••	•••	•••
Vector n:	x	x	x	•••	0
Vector n+1:	1	1	1	•••	1

I_1	I_2	I_3	•••	I_n
s_1	s_2	s_3	•••	s_n

(a) n‐input AND gate

(b) Ternary Simulation

(c) Symbolic Simulation

Researchers at IBM first introduced symbolic simulation to reason out properties of circuits described at the register-transfer level (Carter et al., 1979). With the advent of binary decision diagrams (BDD), the technique became much more practical (Bryant, 1986). Providing a canonical representation for Boolean functions, BDDs enabled the implementation of an efficient event-driven logic simulator that operated over a symbolic domain. By encoding a model’s finite domain using a Boolean encoding, it is possible to symbolically simulate the model using BDDs. Bryant’s formal state transition model for a ternary system (Bryant and Seger, 1990) and Seger’s work on symbolic trajectory evaluation renewed further interest in symbolic execution (Seger and Bryant, 1995).

Symbolic trajectory evaluation (STE) is a modified form of symbolic simulation that operates over the quaternary logic domain 0, 1, *X* and *T* (Seger and Bryant, 1995). A state of the circuit is defined as the set of all node values at a particular time instant. The value domain is partially ordered and forms a complete lattice, $X \sqsubseteq 0$ indicates that *X* has less information than 0, or *X* is weaker than 0. The information content of 0 and 1 are not comparable. If $r \sqsubseteq q$ and $r \sqsubseteq t$, we can think of *r* as representing both *q* and *t*. Any property that holds for a state such as *r* will also hold for all states above it in the lattice, for example *q* and *t*.

STE differs from symbolic simulation in that it provides a mathematically rigorous method for establishing that properties (assertions) of the form antecedent (A) $\Rightarrow$ consequent (C) hold for a given simulation model of a circuit. For the test vector shown in Figure 2(c), the antecedent is: (I_1 is s_1, I_2 is s_2, …, I_n is s_n) from time 0 to 1, and the consequent is: *out* is $s_1\&s_2\& \ldots \&s_n$ from time 1 to 2.

Circuit state holders are initialised with symbolic values specified by the antecedent. The model is then simulated, typically for one or two clock cycles, while driving the inputs with symbolic values during simulation. The resulting values, appearing on selected internal nodes and primary outputs, are compared with the expected values expressed in the consequent. In general, the values could be functions over a finite set of variables. A trajectory is a sequence of states such that each state has at least as much information as the next-state function applied to the previous state. Intuitively, a trajectory is a state sequence constrained by the system’s next-state function. A successful simulation of assertion $A \Rightarrow C$ establishes that any sequence of assignments of values to circuit nodes that is both consistent with the circuit behaviour and consistent with antecedent *A* is also consistent with consequent *C*.

Symbolic trajectory evaluation is used to verify whether an implementation satisfies its specification. Necessary assertions are extracted from the specification. If the implementation is correct, these assertions should hold during symbolic simulation of the *RTL design*. An assertion ($A \Rightarrow C$) holds if the weakest antecedent trajectory that the implementation goes through during simulation (using *A*) is at least as strong as the weakest sequence satisfying the consequent *C*. Informally, the outputs produced during simulation (using *A*) should be at least as strong as the expected outputs (given in *C*).

3 Related work

Several approaches for formal or semi-formal verification of processors have been developed in the past. Theorem proving techniques, for example, have been successfully adapted to verify processors (Cyrluk, 1993; Sawada and Hunt, 1997; Srivas and Bickford, 1990). However, these approaches require a great deal of user intervention, especially for verifying control-intensive designs. Burch and Dill (1994) presented a technique for formally verifying processor control circuitry. Their technique verifies the correctness of the implementation model of a pipelined processor against its ISA model based on quantifier-free logic of equality with uninterpreted functions. The technique has been extended to handle more complex pipelined architectures by several researchers (Skakkebaek et al., 1998; Velev and Bryant, 2000; Velev, 2000). The approach of Velev and Bryant (2000) focuses on efficiently checking the commutative condition for complex microarchitectures by reducing the problem to checking equivalence of two terms in a logic with equality, and uninterpreted function symbols.

Hosabettu (2000) proposed an approach to decompose and incrementally build the proof of correctness of pipelined microprocessors by constructing the abstraction function using completion functions. Huggins and Campenhout verified the ARM2 pipelined processor using Abstract State Machine (Huggins and Campenhout, 1998).

In 1997, Levitt and Olukotun presented a verification technique called unpipelining, which repeatedly merges the last two pipe stages into a single stage, resulting in a sequential version of the processor. A framework for microprocessor correctness statements about safety that is independent of implementation representation and verification approach is presented by Aagaard et al. (2001). Ho et al. (1998) extracted controlled token nets from a logic design to perform efficient model checking. Jacobi (2002) used a methodology to verify out-of-order pipelines by combining model checking for the verification of the pipeline control and theorem proving for the verification of the pipeline functionality. Compositional model checking is used to verify a processor microarchitecture containing most of the features of a modern microprocessor (Jhala and McMillan, 2001). There has been a lot of work in the area of module level validation such as verification of floating-point unit (Ho et al., 1996), and protocol validation such as verification of cache coherence protocol (Pong and Dubois, 1997).

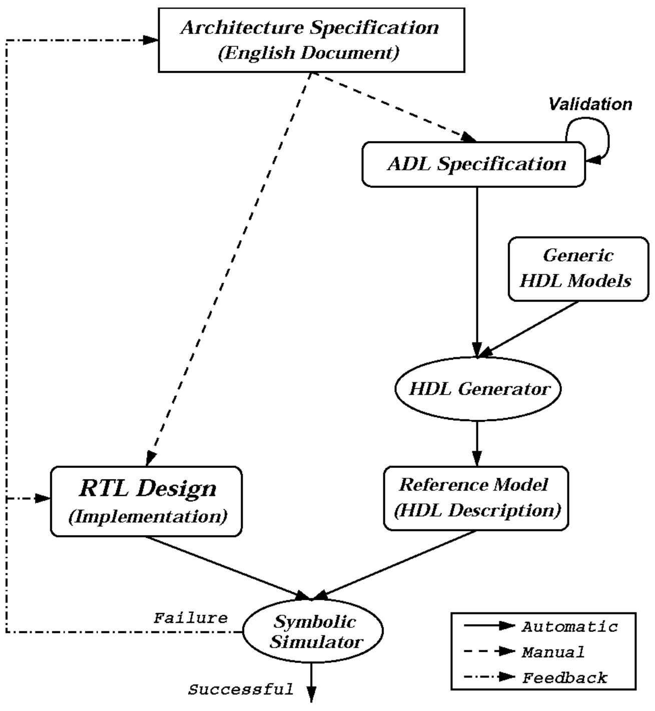
Traditionally, validation of a microprocessor has been performed by applying a combination of random and directed test programs, using simulation techniques. Many techniques have been proposed for generation of directed test programs. Aharon et al. (1995) have proposed a test program generation methodology for functional verification of PowerPC processors in IBM. Shen et al. (1999) have used the processor to generate tests at run-time by self-modifying code, and performed signature comparison with the one obtained from emulation. Ur and Yadin (1999) presented a method for generation of assembler test programs that systematically probe the micro-architecture of a PowerPC processor. Iwashita et al. (1994) used a FSM-based processor modelling to automatically generate test programs. Campenhout et al. (1999) have proposed a test generation algorithm that integrates high-level treatment of the datapath with low-level treatment of the controller. Mishra et al. (2004) have proposed a graph-based functional test program generation technique for pipelined processors.

Symbolic simulation has proven to be an efficient technique, bridging the gap between traditional simulation and full-fledged formal verification. Versys2 (Krishnamurthy et al., 2001) serves as the mainstream custom-memory verification tool for checking Register Transfer Language (RTL) designs against schematics at Motorola's Somerset Design Center. Beatty (1993) verified a switch-level non-pipelined processor description by using Binary Decision Diagrams (BDDs) and symbolic simulation. Bhagwati and Devadas (1994) verified a pipelined implementation of the DLX processor architecture using BDDs and symbolic simulation.

4 Top-down validation methodology

Figure 3 shows our top-down validation methodology. Logic designers implement the architecture at register-transfer level (*RTL design*). The structure and behaviour of the processor are captured using an architecture description language (ADL). The ADL specification is validated to ensure that it specifies a well-formed architecture (Mishra et al., 2002, 2003a). The validated ADL specification can be used for top-down validation of microprocessors. The specification can be used for generating a software toolkit (including compiler and simulator) to perform exploration of processor-memory architectures (Mishra et al., 2001b, 2001c). The specification can also be used to generate test programs for functional validation of pipelined processors (Mishra et al., 2001a; Mishra and Dutt, 2004). In this paper, we present a specification-driven validation methodology using symbolic simulation.

Figure 3 Top-down validation methodology



We use the EXPRESSION ADL (Halambi et al., 1999) in our framework. Our methodology is independent of the ADL used. We can use any ADL that captures both structure and behaviour of the processor. The reference model (HDL description) is automatically generated from the ADL specification. The generated reference model is used as specification by the symbolic simulator.

We use Versys2 symbolic simulator (Krishnamurthy et al., 2001) to establish equivalence between the generated reference model and the actual implementation (*RTL Design*). Versys2 uses symbolic trajectory evaluation to perform equivalence checking. It is necessary to manually specify the state mappings between the reference model and the implementation. This involves mapping of both latches and bit cells by specifying their names. The assertions are automatically generated from the reference model (Wang et al., 1998). Versys2 symbolically simulates the RTL design by using the generated assertions to ensure that the implementation satisfies the specification. A counter-example is generated if an assertion fails in the *RTL design*. The feedback is used to modify the RTL design. In case of an ambiguity in the original description that leads to the mismatch, the architecture specification needs to be updated.

Consider the validation scenario for an adder. The *RTL design* implements an n-input adder. Our framework generates the HDL description of the specification, $\text{output} = \sum_{i=1}^{n} \text{input}_i$, that needs to be satisfied by the implementation. The RTL design should satisfy this specification irrespective of the adder implementation, such as ripple-carry adder or carry look-ahead adder.

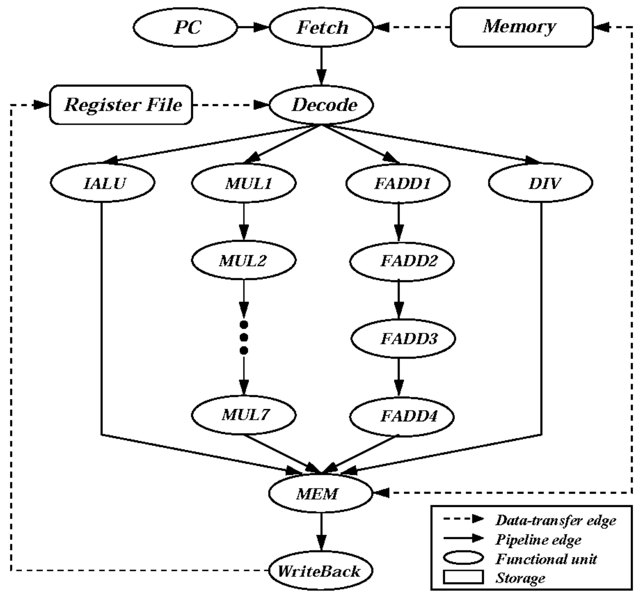
5 Reference model generation

We use the functional abstraction technique to generate the reference model (HDL description) from the ADL specification. The functional abstraction technique was first introduced by Mishra et al. (2001b) for generating simulation models for a wide variety of architectures. In this paper, we have applied the functional abstraction technique to automatically generate HDL models from the ADL specification. First, we briefly describe the EXPRESSION ADL followed by a brief description of the functional abstraction technique. Finally, we describe the generation of HDL models.

5.1 The ADL specification

The EXPRESSION ADL (Halambi et al., 1999) contains information regarding the structure, behaviour and mapping (between structure and behaviour) of the processor. The structure contains the description of each component and the connectivity as a netlist. There are two types of components: *units* (e.g., ALUs) and *storages* (e.g., register files). Each component has a list of attributes. For example, a functional unit will have information regarding latches, ports, connections, supported opcodes, execution timing and capacity. The ADL captures two types of edges in the netlist: pipeline edges and data transfer edges. The pipeline edges specify instruction transfer between units via pipeline latches, whereas data transfer edges specify data transfer between components, typically between units and storages or between two storages. For example, the oval (unit) and rectangular (storage) boxes represent components, and the solid (pipeline) and dotted (data-transfer) lines represent edges in Figure 4.

Figure 4 Structure of the DLX architecture



The behaviour is organised into operation groups, with each group containing a set of operations having some common characteristics. Each operation is described in terms of its opcode, operands, behaviour and instruction format. Each operand is classified either as source or as destination. Furthermore, each operand is associated with a type that describes the type and size of the data it contains. The instruction format describes the fields of the operation in binary and assembly. For example, Figure 5 shows the description of an ADD operation.

Figure 5 Behaviour of an ADD operation

```
( OPCODE add
  (OPERANDS (SRC1 reg) (src2 reg/immediate) (DEST reg))
  (BEHAVIOR DEST = SRC1 + SRC2)
  (FORMAT 010110 dest(25-21) src1(20-16) src2(15-0))
)
```

5.2 The functional abstraction

The functional abstraction technique was first introduced by Mishra et al. (2001b) for generating simulation models from the ADL specification. The notion of functional abstraction comes from a simple observation: different architectures may use the same functional unit (e.g., fetch) with different parameters; the same functionality (e.g., operand read) may be used in different functional units, or may have new architectural features. The first difference can be eliminated by defining generic functions with appropriate parameters. The second difference can be eliminated by defining generic sub-functions, which can be used by different architectures at different stages in the pipeline. The last one is difficult to alleviate since it is new, unless this new functionality can be composed of existing sub-functions (e.g., *multiply-accumulate* operation by combining *multiply* and *add* operations). The necessary generic functions, sub-functions and computational environment needed to capture a wide variety of processor and memory features were defined.

The structure of each functional unit is captured using parameterised functions. For example, the fetch unit functionality contains several parameters, such as number of operations read per cycle, number of operations written per cycle, reservation station size, branch prediction scheme, number of read ports, number of write ports, etc. Figure 6 shows a specific example of a fetch unit described using sub-functions. Each sub-function is defined using appropriate parameters. For example, *ReadInstMemory* reads *n* operations from instruction cache using current PC address (returned by *ReadPC*) and writes them to the reservation station. The notion of generic sub-function allows the flexibility of specifying the system in finer detail. It also allows reuse of the components.

The behaviour of a generic processor is captured through the definition of opcodes. Each opcode is defined as a function with a generic set of parameters, which performs the intended functionality. The parameter list includes source and destination operands, necessary control and data type information. For example, some common sub-functions are ADD, SUB, MUL, SHIFT, etc. The opcode functions may use one or more sub-functions.

For example, the MAC (multiply and accumulate) uses two sub-functions (ADD and MUL) as shown in Figure 7.

Figure 6 A fetch unit example

```
FetchUnit(# of read/cycle, buffer size, ...)
{
   address = ReadPC();
   instructions = ReadInstMemory(address, n);
   WriteToReservationStation(instructions, n);
   outInst = ReadFromReservationStation(m);
   WriteLatch(decode_latch, outInst);

   pred = QueryPredictor(address);
   if pred {
      nextPC = QueryBTB(address);
      SetPC(nextPC);
   } else
      IncrementPC(x);
}
```

Figure 7 Modelling of MAC operation

```
ADD (src1, src2) {                MUL (src1, src2) {
  return (src1 + src2);             return (src1 * src2);
}                                 }

    MAC (src1, src2, src3) {
       return ( ADD( MUL(src1, src2), src3) );
    }
```

Similarly, generic functions and sub-functions for memory modules, controller, interrupts, exceptions, DMA and coprocessor were defined. The detailed description of generic abstractions for all of the microarchitectural components can be found in Mishra et al. (2001b).

5.3 Generation of HDL description

We have implemented all the generic functions and sub-functions using HDL. Our framework generates HDL description from the ADL specification of the processor by composing functional abstraction primitives. In this section, we briefly describe how to generate three major components of the processor: instruction decoder, datapath and controller, using the generic HDL models. The detailed description is available in Mishra et al. (2003b).

A generic instruction decoder uses information regarding individual instruction format and opcode mapping for each functional unit to decode a given instruction. The instruction format information is available in operations section of the EXPRESSION ADL. The decoder extracts information regarding opcode, operands, etc., from input instruction using the instruction format. The mapping section of the EXPRESSION ADL captures the information regarding the mapping of opcodes to the functional units. The decoder uses this information to perform/initiate necessary functions (e.g., operand read), and decides where to send the instruction.

The implementation of datapath consists of two parts. First, it composes each component in the structure. Second, it instantiates components (e.g., fetch, decode, ALU, LdSt, writeback, branch, caches, register files, memories, etc.) and establish connectivity using appropriate number of latches, ports and connections using the structural information available in the ADL. To compose each component in the structure, we use the information available in the ADL regarding the functionality of the component and its parameters. For example, to compose an execution unit, it is necessary to instantiate all the opcode functionalities (e.g., ADD and SUB, for an ALU) supported by that execution unit. Also, if the execution unit is supposed to read the operands, appropriate number of operand read functionalities need to be instantiated, unless the same read functionality can be shared using multiplexors. Similarly, if this execution unit is supposed to write the data back to register file, the functionality for writing the result needs to be instantiated. The actual implementation of an execute unit might contain many more functionalities, e.g., read latch, write latch and insert/delete/modify reservation station (if applicable).

The controller is implemented using a combination of distributed (local) as well as centralised control mechanisms. A local controller is maintained at each functional unit in the pipeline to generate necessary control signals based on the input instruction. For example, the local controller in an execute unit will activate the add operation if the opcode is add, or it will set the busy bit in case of a multi-cycle operation. The centralised controller maintains the information regarding each functional unit, such as busy, stalled, etc. using generic controller function with appropriate parameters. It stalls or flushes the pipeline based on the list of instructions in the pipeline and hazard details.

6 Experiments

An important aspect of our methodology is the ability to perform both model (property) checking and equivalence checking depending on the generated reference model. To verify whether the implementation (RTL design) satisfies certain properties, our framework generates behaviours for the intended properties instead of generating the complete reference design. On the other hand, if the generated reference model contains the complete description of the design, our framework performs equivalence checking between the implementation and the generated reference model.

The major advantage of property checking is reduction of the verification complexity. However, it raises an important question: how to choose the set of properties. It can be done in two different ways. First, the designers can decide the properties that are important to be verified for the design based on their design knowledge and past experiences. Second, a set of behaviours can be chosen and their effectiveness can be evaluated. For example, to verify a memory controller in a microprocessor, it is necessary to generate properties to validate each output of the controller. To measure the effectiveness of these properties, a set of coverage measures can be used during property checking

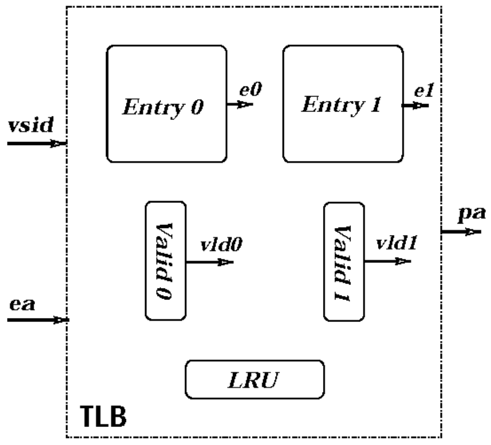
(Chockler et al., 2001). The discussion on the completeness of a set of properties is beyond the scope of this paper.

6.1 Property checking of a memory management unit

The PowerPC Memory Management Unit (MMU) supports demand-paged virtual memory. It consists of blocks such as *Segment Registers, Translation Lookaside Buffers (TLBs)* and *Block Address Translation (BAT)* arrays. Each of these memory blocks is composed of sub-blocks. For example, the TLB has three sub-blocks viz., *entry* (data information), *LRU* (least recently used information) and *valid* (information regarding validity of the data), as shown in Figure 8. Each of these sub-blocks is implemented as SRAM. The typical operations in SRAM are read and write. Therefore, a natural property to verify is to check, read and write for each SRAM cell. The generated reference model contains the following Verilog code segment to verify the read and write properties for an SRAM cell.

```
//Write Property
always @ (wrClk or wrEn or dIn or wrAddr)
begin
  if (wrClk & wrEn) ram[wrAddr] <= dIn;
end
//Read Property
assign out = (rdClk & rdEn) ? ram[rdAddr] : 32'b0;
```

Figure 8 TLB block diagram



We modified Versys2 configuration file to give the node mapping between the reference model and the implementation. For example, the wrClk of the reference model is mapped to sramWrClk of the implementation. An interesting feature of this validation approach is that the same property (without any modification) is applied to all the memory blocks in the MMU. We modified the Versys2 configuration file to provide node mapping between the reference model (property) and the implementation.

To verify whether the *RTL design* correctly implements the TLB miss detection, our framework generated the following Verilog code segment. The information needed to build this property is directly available from the specification of the MMU.

```
assign inp =({1'b1,vsid[0:23],ea[4:9],ea[10:13]});
assign out0=({vld0,e0[0:23],e0[24:29],e0[54:57]});
assign out1=({vld1,e1[0:23],e1[24:29],e1[54:57]});
assign hit0=(inp == out0);
assign hit1=(inp == out1);
assign miss=~(hit0 | hit1);
```

This property verifies miss detection for a two-way set-associative TLB. It would be a simple extension to generate this property for a *n*-way set-associative TLB. Here *vsid* (virtual segment id) and *ea* (effective address) are inputs, and *pa* (physical address) is output of the TLB block. The *e* and *vld* variables are outputs from the *entry* and *valid* blocks, respectively, as shown in Figure 8.

Similarly, we generated and validated the property for the BAT array miss detection. There were several mismatches found during property checking. The architecture specification document does not provide the value for the else condition (default value of a signal, for example) in some of the cases. As a result, the description of the property does not have the default value for a signal, whereas the signal has a definite value in its implementation under all possible conditions. Symbolic simulation produced mismatches in those cases. Consider the following read implementation of a SRAM cell. This implementation assigns *32'b*0 to signal *out* when condition (*rdClk* & *rdEn*) is *false.* However, the architecture document does not specify the value in the default case. As a result, the generated property does not have the value that caused the mismatch.

```
assign out = (rdClk & rdEn) ? ram[rdAddr] : 32'b0;
```

The architecture document can be updated to add the values in all cases. It is also possible to impose certain constraints in Versys2 (Krishnamurthy et al., 2001) to avoid the detection of such false negatives. For example, we can set the condition (*rdClk* & *rdEn*) as *true* in the Versys2 configuration file to avoid the detection of the mismatch mentioned above.

6.2 Equivalence checking of the DLX architecture

In a case study, we successfully applied the proposed methodology to validate the DLX (Hennessy and Patterson, 1990) processor. We have chosen DLX processor since it has been well studied in academia, and there are HDL implementations available that can be used in our validation framework.

The Versys2 symbolic simulator accepts Verilog descriptions only. We could not find any publicly available Verilog implementation of the DLX processor. We obtained a VHDL implementation of the synthesisable 32-bit RISC DLX from *eda.org* (http://www.eda.org/rassp/vhdl/models/processor.html). We converted it to Verilog description using Synopsys Design Compiler (http://www.synopsys.com). We used this Verilog description as the implementation.

The structure and behaviour of the DLX architecture is captured using EXPRESSION ADL. Our framework generated the Verilog description from the ADL specification using the method described in Section 5. The generated Verilog description is used as the reference model (specification) for the validation.

Versys2 generated assertions from the reference model and applied them to the implementation using symbolic trajectory evaluation. The equivalence checking process took 149.5 seconds on a 296 MHz Sun Ultra-250 with 1024 M RAM. The node names (inputs, outputs and sequential elements) of the specification and the implementation are mapped in the Versys2 configuration file. We have encountered many mismatches during verification. One of them was a mismatch in the output data bus at clock cycle 2500. The analysis of the assertion (antecedent $\Rightarrow$ consequent) that caused the failure and the signals of both the reference model and the implementation revealed that the problem was in the overflow bit of the adder. The ripple-carry adder implementation of the DLX (http://www.eda.org/rassp/vhdl/models/processor.html) had incorrect computation of the overflow bit.

Design analysis in our framework is very fast once we figure out the module that is causing the problem. For example, in this particular case, once we know that the adder is causing the problem, we can verify the adder implementation of the DLX by generating an adder specification (HDL description) from our framework and applying symbolic simulation. The framework took 5.6 seconds to produce the error showing differences in the overflow bit.

7 Conclusion

Verification is one of the most complex and expensive tasks in the current microprocessor design flow. A significant bottleneck in the validation of such systems is the lack of a golden reference model. Thus, many existing approaches employ bottom-up validation methodology by using a combination of simulation techniques and formal methods.

We presented a top-down validation approach using symbolic simulation. The reference model (HDL description) is automatically generated from the ADL specification of the processor architecture. An important aspect of our methodology is the ability to perform both model (property) checking and equivalence checking, depending on the generated reference model. Our framework generates behaviours of the intended properties to enable model checking, and generates the complete description of the processor to enable equivalence checking. To verify the implementation, Versys2 (Krishnamurthy et al., 2001) generates assertions from the reference model and applies them to the implementation using symbolic trajectory evaluation. We applied our methodology in two validation scenarios: property checking of a memory management unit of a microprocessor that is compliant with the PowerPC instruction-set, and verification of a DLX processor. Our future work will focus on improving this methodology for verifying real-world microprocessors.

Acknowledgement

This work was partially supported by NSF grants, CCR-0203813 and CCR-0205712. We would like to acknowledge the members of the ACES laboratory for their inputs.

References

Aagaard, M., Cook, B., Day, N. and Jones, R. (2001) 'A framework for microprocessor correctness statements', in Margaria, T. and Melham, T. (Eds.): *Proc. of Correct Hardware Design and Verification Methods (CHARME), Vol. 2144 of LNCS*, Springer-Verlag, pp.433–448.

Aharon, A., Goodman, D., Levinger, M., Lichtenstein, Y., Malka, Y., Metzger, C., Molcho, M. and Shurek, G. (1995) 'Test program generation for functional verification of PowerPC processors in IBM', *Proc. of Design Automation Conference (DAC)*, pp.279–285.

Beatty, D.L. (1993) *A Methodology for Formal Hardware Verification with Application to Microprocessors*, PhD Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA.

Bhagwati, V. and Devadas, S. (1994) 'Automatic verification of pipelined microprocessors', *Proc. of Design Automation Conference (DAC)*, pp.603–608.

Bryant, R. (1986) 'Graph-based algorithms for Boolean function manipulation', *IEEE Trans. Computers*, Vol. C-35, No. 8, August, pp.677–691.

Bryant, R. (1990) 'Symbolic simulation – techniques and applications', *Proc. of Design Automation Conference (DAC)*, pp.517–521.

Bryant, R. and Seger, C. (1990) 'Formal verification of digital circuits using symbolic ternary system models', *Proc. of Computer Aided Verification (CAV)*, pp.121–146.

Burch, J. and Dill, D. (1994) 'Automatic verification of pipelined microprocessor control', in Dill, D. (Ed.): *Proc. of Computer Aided Verification (CAV), Vol. 818 of LNCS*, Springer-Verlag, pp.68–80.

Campenhout, D., Mudge, T. and Hayes, J. (1999) 'High-level test generation for design verification of pipelined microprocessors', *Proc. of Design Automation Conference (DAC)*, pp.185–188.

Carter, W., Joyner, W. and Brand, D. (1979) 'Symbolic simulation for correct machine design', *Proc. of Design Automation Conference (DAC)*, pp.280–286.

Chockler, H., Kupferman, O., Kurshan, R. and Vardi, M. (2001) 'A practical approach to coverage in model checking', *Proc. of Computer Aided Verification (CAV), Vol. 2102 of LNCS*, Springer-Verlag, pp.66–78.

Cyrluk, D. (1993) *Microprocessor Verification in PVS: A Methodology and Simple Example*, Technical Report, SRI-CSL-93-12.

Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N. and Nicolau, A. (1999) 'EXPRESSION: a language for architecture exploration through compiler/simulator retargetability', *Proc. of Design Automation and Test in Europe (DATE)*, pp.485–490.

Hennessy, J. and Patterson, D. (1990) *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc, San Mateo, CA.

Ho, P., Hoskote, Y., Kam, T., Khaira, M., O'Leary, J., Zhao, X., Chen, Y. and Clarke, E. (1996) 'Verification of a complete floating-point unit using word-level model checking', in Srivas, M. and Camilleri, A. (Eds.): *Proc. of Formal Methods in Computer-Aided Design (FMCAD), Vol. 1166 of LNCS*, Springer-Verlag, pp.19–33.

Ho, P., Isles, A. and Kam, T. (1998) 'Formal verification of pipeline control using controlled token nets and abstract interpretation', *Proc. of International Conference on Computer-Aided Design (ICCAD)*, pp.529–536.

Hosabettu, R.M. (2000) *Systematic Verification of Pipelined Microprocessors*, PhD Thesis, Department of Computer Science, University of Utah.

Huggins, J. and Campenhout, D. (1998) 'Specification and verification of pipelining in the ARM2 RISC microprocessor', *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, Vol. 3, No. 4, October, pp.563–580.

Iwashita, H., Kowatari, S., Nakata, T. and Hirose, F. (1994) 'Automatic test pattern generation for pipelined processors', *Proc. of International Conference on Computer-Aided Design (ICCAD)*, pp.580–583.

Jacobi, C. (2002) 'Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving', in Brinksma, E. and Larsen, K. (Eds.): *Proc. of Computer Aided Verification (CAV), Vol. 2404 of LNCS*, Springer-Verlag, pp.309–323.

Jhala, R. and McMillan, K.L. (2001) 'Microarchitecture verification by compositional model checking', in Berry, G. *et al*. (Eds.): *Proc. of Computer Aided Verification (CAV), Vol. 2102 of LNCS*, Springer-Verlag, pp.396–410.

Krishnamurthy, N., Abadir, M., Martin, A. and Abraham, J. (2001) 'Design and development paradigm for industrial formal verification tools', *IEEE Design and Test of Computers*, Vol. 18, No. 4, July–August, pp.26–35.

Levitt, J. and Olukotun, K. (1997) 'Verifying correct pipeline implementation for microprocessors', *Proc. of International Conference on Computer-Aided Design (ICCAD)*, pp.162–169.

Mishra, P. and Dutt, N. (2004) 'Graph-based functional test program generation for pipelined processors', *Proc. of Design Automation and Test in Europe (DATE)*, pp.182–187.

Mishra, P., Dutt, N. and Nicolau, A. (2001a) 'Automatic validation of pipeline specifications', *Proc. of High Level Design Validation and Test (HLDVT)*, pp.9–13.

Mishra, P., Dutt, N. and Nicolau, A. (2001b) 'Functional abstraction driven design space exploration of heterogeneous programmable architectures', *Proc. of International Symposium on System Synthesis (ISSS)*, pp.256–261.

Mishra, P., Tomiyama, H., Halambi, A., Grun, P., Dutt, N. and Nicolau, A. (2002) 'Automatic modeling and validation of pipeline specifications driven by an architecture description language', *Proc. of Asia South Pacific Design Automation Conference (ASPDAC)/International Conference on VLSI Design*, pp.458–463.

Mishra, P., Dutt, N. and Tomiyama, H. (2003a) 'Towards automatic validation of dynamic behaviour in pipelined processor specifications', *To appear, Kluwer Design Automation for Embedded Systems*, Vol. 8, Nos. 2–3, June–September, pp.249–265.

Mishra, P., Kejariwal, A. and Dutt, N. (2003b) 'Rapid exploration of pipelined processors through automatic generation of synthesizable RTL models', *Proc. of Rapid System Prototyping (RSP)*, pp.226–232.

Mishra, P., Grun, P., Dutt, N. and Nicolau, A. (2001c) 'Processor-memory co-exploration driven by an architectural description language', *Proc. of International Conference on VLSI Design*, pp.70–75.

Pong, F. and Dubois, M. (1997) 'Verification techniques for cache coherence protocols', *ACM Computing Surveys*, Vol. 29, No. 1, pp.82–126.

Sawada, J. and Hunt, J.W.A. (1997) 'Trace table based approach for pipelined microprocessor verification', in Grumberg, O. (Ed.): *Proc. of Computer Aided Verification (CAV), Vol. 1254 of LNCS*, Springer-Verlag, pp.364–375.

Seger, C. and Bryant, R. (1995) 'Formal verification by symbolic evaluation of partially-ordered trajectories', *Formal Methods in System Design*, Vol. 6, No. 2, pp.147–189.

Shen, J., Abraham, J., Baker, D., Hurson, T., Kinkade, M., Gervasio, G., Chu, C. and Hu, G. (1999) 'Functional verification of the equator MAP 1000 microprocessor', *Proc. of Design Automation Conference (DAC)*, pp.169–174.

Skakkebaek, J., Jones, R. and Dill, D. (1998) 'Formal verification of out-of-order execution using incremental flushing', in Hu, A. and Vardi, M. (Eds.): *Proc. of Computer Aided Verification (CAV), Vol. 1427 of LNCS*, Springer-Verlag, pp.98–109.

Srivas, M. and Bickford, M. (1990) 'Formal verification of a pipelined microprocessor', *IEEE Software*, Vol. 7, No. 5, pp.52–64.

Synopsys Design Compiler, http://www.synopsys.com.

Synthesizable DLX: Generic 32-bit RISC Processor, http://www.eda.org/rassp/vhdl/models/processor.html.

Ur, S. and Yadin, Y. (1999) 'Micro architecture coverage directed generation of test programs', *Proc. of Design Automation Conference (DAC)*, pp.175–180.

Velev, M. and Bryant, R. (2000) 'Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction', *Proc. of Design Automation Conference (DAC)*, pp.112–117.

Velev, M.N. (2000) 'Formal verification of VLIW microprocessors with speculative execution', in Emerson, E. and Sistla, A. (Eds.): *Proc. of Computer Aided Verification (CAV), Vol. 1855 of LNCS*, Springer, pp.296–311.

Wang, L., Abadir, M. and Krishnamurthy, N. (1998) 'Automatic generation of assertions for formal verification of PowerPC microprocessor arrays using symbolic trajectory evaluation', *Proc. of Design Automation Conference (DAC)*, pp.534–537.