To appear in the proceedings of the *36th Design Automation Conference (DAC'99)*, June 1999.

Exploiting Positive Equality and Partial Non-Consistency in the Formal Verification of Pipelined Microprocessors[1]

Miroslav N. Velev[*]
mvelev@ece.cmu.edu
http://www.ece.cmu.edu/~mvelev

Randal E. Bryant[‡, *]
randy.bryant@cs.cmu.edu
http://www.cs.cmu.edu/~bryant

[*]Department of Electrical and Computer Engineering
[‡]School of Computer Science
Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

Abstract

We study the applicability of the logic of Positive Equality with Uninterpreted Functions (PEUF) [2][3] to the verification of pipelined microprocessors with very large Instruction Set Architectures (ISAs). Abstraction of memory arrays and functional units is employed, while the control logic of the processors is kept intact from the original gate-level designs. PEUF is an extension of the logic of Equality with Uninterpreted Functions, introduced by Burch and Dill [4], that allows us to use distinct constants for the data operands and instruction addresses needed in the symbolic expression for the correctness criterion. We present several techniques that make PEUF scale very efficiently for the verification of pipelined microprocessors with large ISAs. These techniques are based on allowing a limited form of non-consistency in the uninterpreted functions, representing initial memory state and ALU behaviors. Our tool required less than 30 seconds of CPU time and 5 MB of memory to verify a 5-stage MIPS-like pipelined processor that implements 191 instructions of various classes. The verification was done by correspondence checking - a formal method, where a pipelined microprocessor is compared against a non-pipelined specification.

1. Introduction

The logic of Positive Equality with Uninterpreted Functions (PEUF) [2][3] was proposed as an extension of the logic of Equality with Uninterpreted Functions (EUF), introduced by Burch and Dill [4]. Uninterpreted functions allow the abstract modeling of functional units and memories by replacing their actual implementations when formally verifying microprocessors. That leads to a considerable reduction of the computational complexity when verifying pipelined microprocessors.

1. This research was supported in part by the SRC under contract 98-DC-068.

By imposing some restrictions in the syntax of EUF, PEUF allows the use of a distinct constant for each data operand or instruction address used in the symbolic expression for the correctness criterion. By a distinct constant we mean a term which is not equal to any other term in the same domain. The result is a significantly increased computational efficiency of PEUF, compared to EUF.

The focus of this paper is how to make PEUF scale easily for verification of realistic pipelined microprocessors with large ISAs. We propose the use of initial state which is non-consistent across instructions, but consistent for the same instruction. We also propose an efficient way of generating distinct constants. The result is a very high computational efficiency for PEUF, and invariance of the verification CPU time and memory with the number of instructions implemented in the processor.

In modeling of microprocessors, we use abstraction of memory arrays and functional units. We achieve the abstraction by means of the Efficient Memory Model (EMM) [13][14] and its capability to dynamically introduce new initial state (as required by a simulation sequence) which is consistent with previously introduced initial state. Observing that every combinational block of logic can be implemented as a read-only memory with the logic block inputs serving as memory addresses, we abstract functional units at the bit level by replacing them with read-only EMMs. The definition of the EMM automatically enforces consistency of the output values for the present input pattern with output values returned for previous input patterns.

When using the EMM to replace memories and functional units, we assume that their actual implementations have been verified separately. For example, the formal method of symbolic trajectory evaluation has been combined with symmetry reductions to enable the verification of very large memory arrays at the transistor level [11]. An efficient representation of word-level functions has enabled the verification of complex functional units like floating-point multipliers [6].

We also use an efficient encoding technique [15], targeted to EUF and PEUF, for representing term-variables by means of BDD variables [1]. This technique allows such term-variables to be used while symbolically simulating the processor's control logic, kept intact from the actual gate-level design. Thus, we avoid the need to create a separate abstract model, as is done in previous methods based on uninterpreted functions [4][7][8]. Our previous results [15] showed that the encoding technique cannot break the circular dependencies between functional units,

which result from the feedback loops of the forwarding logic, and hence cannot scale for verification of realistic pipelined microprocessors.

When verifying pipelined microprocessors, we use the formal method of *correspondence checking* - comparison to a non-pipelined specification, as pioneered by Burch and Dill [4][5].

2 Logic of Positive Equality with Uninterpreted Functions

Burch and Dill illustrated that EUF fits very efficiently the problem of representing and verifying a pipelined microprocessor [4] by comparing it against a non-pipelined specification. Particularly, functional units and memories can be abstracted as uninterpreted functions or predicates that take as inputs data operands, represented abstractly as terms. The forwarding logic can be described as nested *ITE* operators that select one out of several terms, based on a formula, produced by the control logic.

We can observe that there are 3 classes of terms needed in the verification of pipelined processors by correspondence checking: instruction addresses, register identifiers, and data. Of these only the register identifiers are compared for equality by the gate-level control logic, in order to form control decisions, e.g., for forwarding or stalling. This is based on the assumption that equality comparisons of data terms are made only by means of uninterpreted predicates, e.g., the decision to take a conditional branch when the terms are equal. Hence, the interpreted equality predicate "=" would never be applied on such terms. The same is true for instruction address terms. This allows us to impose some restrictions to EUF in order to gain computational efficiency.

2.1 Syntax

$$
\begin{aligned}
\textit{formula} &::= \textit{propositional-variable} \mid \textbf{true} \mid \textbf{false} \\
&\mid \neg \textit{formula} \mid (\textit{formula} \wedge \textit{formula}) \\
&\mid (\textit{term} = \textit{term}) \\
&\mid \textit{predicate-symbol}(\textit{p-term}, \ldots, \textit{p-term}) \\
\textit{p-formula} &::= \textit{formula} \\
&\mid (\textit{p-formula} \vee \textit{p-formula}) \\
&\mid (\textit{p-formula} \wedge \textit{p-formula}) \\
&\mid (\textit{p-term} = \textit{p-term}) \\
\textit{term} &::= \textit{term-variable} \\
&\mid \textit{ITE}(\textit{formula}, \textit{term}, \textit{term}) \\
&\mid \textit{function-symbol}(\textit{p-term}, \ldots, \textit{p-term}) \\
\textit{p-term} &::= \textit{term} \\
&\mid \textit{p-term-variable} \\
&\mid \textit{ITE}(\textit{formula}, \textit{p-term}, \textit{p-term}) \\
&\mid \textit{p-function-symbol}(\textit{p-term}, \ldots, \textit{p-term})
\end{aligned}
$$

Fig. 1: Syntax rules for the logic of Positive Equality with Uninterpreted Functions

The logic of Positive Equality with Uninterpreted Functions (PEUF) extends EUF [4] by adding an additional class of terms called "p-terms" (for "positive terms") with its own set of variables and function symbols such that these terms are used in a highly restricted fashion. In particular, we allow equality tests to be performed among p-terms, but we only allow the results of these tests to be used in monotonically positive Boolean formulas. These formulas cannot be used to control *ITE* operators. The benefit of keeping this restricted class of terms is that they can be handled in a simpler and more efficient way by the validity checker. Fig. 1 shows the syntax of PEUF.

P-terms include general terms, in addition to a special class of variables called "p-term-variables," the *ITE* operator applied to p-terms, and the application of a special class of function symbols known as "p-function" symbols. Two p-terms may be compared for equality, but the result is a restricted form of formula called a p-formula. A p-formula may only contain the monotonically positive Boolean connectives $\wedge$ and $\vee$. It cannot contain any negations and it cannot be used as the control for an *ITE* operator outside of uninterpreted functions and predicates.

2.2 Deciding P-Formulas

We say that an interpretation I is maximally diverse with respect to a set of p-terms T, when for any two terms s and t in T, $I(s) = I(t)$ only when either 1) s and t are identical, or 2) s and t are both applications of the same p-function-symbol f on lists of argument terms with equal interpretations, i.e., $s = f(s_1, \ldots, s_k)$ and $t = f(t_1, \ldots, t_k)$, where $I(s_i) = I(t_i)$, $1 \le i \le k$. Also, it is assumed that a p-term-variable or a p-function-symbol application is not equal to any term-variable or function-symbol application.

For a p-formula F, an interpretation I is maximally diverse when I is maximally diverse with respect to the set of terms $\{t \text{ in } F \mid t \text{ is a p-term-variable or a p-function application result}\}$.

Theorem. *A p-formula is valid iff it is true for a maximally diverse interpretation.*

The intuitive explanation of the theorem is that a maximally diverse interpretation of a p-formula creates the worst case scenario for the p-term equality comparisons in it. If the p-formula is true under a maximally diverse interpretation, then the validity of the formula will be preserved for any other interpretation, due to the monotonicity of the Boolean connectives $\wedge$ and $\vee$, which are allowed in p-formulas. The complete proof is presented in [3]. The result allows us to use a distinct constant for each p-term-variable and p-function-symbol application, i.e., for each instruction address or data operand, when computing the correctness criterion. By a distinct constant we mean a term which is not equal to any other term in the same domain.

3 Encoding Term Values with Symbolic Bit Vectors

We will consider two kinds of term values - constants, which represent p-term-variables and p-function-symbol application results, and variables, which represent term-variables and function-symbol application results. We introduce a separate domain for constants and a separate domain for variables, such

that terms are compared for equality with only terms from the same domain. Constants have a fixed interpretation and are encoded with distinct bit vectors with a Boolean constant (i.e., either **true** or **false**) in each position. Variables have an interpretation that may map them to any value in the domain. Our technique to dynamically generate bit vectors that encode term variables from the same domain can be summarized as follows (see [14] for details). When generating the n^{th} vector, it could potentially have n possible values - to be equal to any of the previous n-1 vectors, or to be distinct from all of them. Therefore, we use $\lceil \log(n) \rceil$ new Boolean variables in the low order bits of the nth vector and the binary constant 0 in the remaining bit positions. If the vectors have a width of k bits, as determined by the circuit, then the number of variables generated for a new vector saturates at k.

4 Abstracting Memories and Functional Units

We will use the types address expression, **AExpr,** and data expression, **DExpr**, for denoting the kind of information that can be applied at the inputs or produced by the outputs of an abstract memory. Let m_0 : **AExpr** $\rightarrow$ **DExpr**, defined as a mapping from address expressions to data expressions, be the initial state of such a memory. Then, $m_0(a)$, where a is an address expression, will return the initial data of the memory at address a. The write operation for an abstract memory will be defined as $Write(m_i,\ a_1,\ d_1) \rightarrow m_{i+1}$ [10], i.e., taking as arguments the present state m_i of a memory, and address expression a_1 designating the location which is updated to contain data expression d_1, and producing the subsequent memory state m_{i+1}, such that:

$$m_{i+1}(a_2) \rightarrow ITE(a_1 = a_2,\ d_1,\ m_i(a_2)). \qquad (1)$$

Based on the observation that any functional block can be represented as a read-only-memory (ROM), with the block's inputs serving as memory addresses, we will represent abstract functional units as abstract ROMs. According to the semantics of an abstract memory, an abstract ROM will always satisfy the property $a_1 = a_2 \Rightarrow f(a_1) = f(a_2)$, where $f()$ denotes the output function of the ROM-modeled abstract functional unit.

Motivated by application to actual circuits, we will represent address and data expressions by vectors of Boolean expressions having width n and w, respectively, for a memory with $N = 2^n$ locations, each holding a word of w bits. The type **BExpr** will denote Boolean expressions.

Address comparison is implemented as:

$$A1 = A2 \ \doteq\ \neg \bigvee_{i=1}^{n} A1_i \oplus A2_i\,, \qquad (2)$$

while address selection $A1 \leftarrow ITE(b,\ A2,\ A3)$ is implemented by selecting the corresponding bits:

$$A1_i \leftarrow ITE(b,\ A2_i,\ A3_i)\,, \quad i = 1, \ldots, n\,. \qquad (3)$$

The definition of data operations is similar, but over vectors of width w.

We use the Efficient Memory Model (EMM), a behavioral memory model for symbolic simulation, in order to represent register files, memories, and latches in the circuits that we examine. During symbolic simulation, the sequence of writes to each EMM is represented as a list, *write_list*, that contains entries of the form $\langle c, a, d \rangle$, where c is a Boolean expression denoting the set of contexts (conditions) for which the entry is defined, a is an address expression denoting a memory location, and d is a data expression denoting the contents of this location. The context information is included for modeling memory systems where the *Write* operations may be performed conditionally, depending on the value of a control signal, i.e., a write port enable signal. Initially *write_list* is empty for each EMM. Given an update $Write(m_i, \langle c_1, a_1, d_1 \rangle)$ of the current memory state m_i, the subsequent memory state m_{i+1} is defined as:

$$m_{i+1}(a_2) \rightarrow ITE((a_1 = a_2) \wedge c_1,\ d_1,\ m_i(a_2)). \qquad (4)$$

5 Exploiting Non-Consistency of a Memory's Initial State

In this paper we relax the constraint for consistency of all the initial state that is introduced on-the-fly. First, we use the value of the Sequential Program Counter (pointing to the instruction that follows sequentially the presently executed instruction, so that it will be equal to PC + 4 in many architectures), already available in the Execution Stage for computing the Target Program Counter of jump and branch instructions, as an additional input to the ALU. The effect is to make function *InitState*() of the EMM, that models the ALU, be non-consistent across instructions but consistent for the same instruction (as identified by the instruction address), thus turning the ALU into a different uninterpreted function for each instruction executed. The ALU in the specification non-pipelined processor is defined identically, with the Sequential Program Counter serving as an input.

This idea is based on the observation that we need to preserve the consistency between the implementation and the specification simulation sequences, while the consistency within the same simulation sequence is not important when evaluating the correctness criterion. It should be pointed out that non-consistency is a conservative approximation. If the processor is correct when its functional units' outputs are non-consistent across instructions (from the same simulation sequence), it will also be correct when the constraint for consistency is imposed. The same idea can be applied to the initial state of all functional units and memory arrays.

Second, when modeling a Register File, which has one write-port and two read-ports (one for each of two source register identifiers), we represent it with two register files. Each of them provides the data for one of the source registers, while both get updated in the way that the original register file is. In this way, the initial states for the source registers read will not be consistent across the two register files. Note that this representation of a Register File is more conservative than the original one - if the pipelined processor is correct without the consistency constrained for the initial states of the two source registers for each instruction, it will be correct when we impose that constraint as well.

6 Experimental Results

In previous work, we attempted to verify a pipelined MIPS processor with a memory stage [15]. The control logic was kept intact at the gate-level. Term-variables, encoded with BDD variables as opposed to constants, were used for the data operands

and instruction addresses. However, for optimal BDD sizes, the BDD variables that encode the address term-variables of an EMM should precede in the variable ordering those BDD variables that encode the EMM's data term-variables. The problem was that the Data Memory gets addressed with data term-variables, produced by the ALU in the Execution stage. At the same time the Data Memory produces term-variables which are fed back into the ALU, by means of the forwarding logic and the Register File. As inputs to the ALU, these term-variables will be compared for equality against former input term-variables in order to select an output term-variable from among both the former output term-variables and a newly generated one. Hence, there is a circular dependency between the term-variables produced by the ALU and the Data Memory, due to the feedback loops in the data path. The result is an exponential complexity of the BDDs, when computing the correctness criterion, and insufficient memory, given a limit of 256 MB. Isles *et al.* [8], who were trying to verify an abstractly-defined design of a pipelined DLX processor with a memory stage, have also run out of memory, given a limit of 1 GB.

In this paper, we examine three versions of a 5-stage pipelined MIPS processor [12] with a 32-bit data path - see Table 1. Each is compared to its non-pipelined specification processor. The functional units, register file, and pipeline latches are replaced by EMMs. The control logic is described at the gate level. The instruction-decoding PLA is defined to produce a unique pattern of ALU control bits for each instruction, while the other control bits are determined according to the class of the instruction.

Processor	ISA count	ISA Composition			
		register-register instructions	register-immediate instructions	loads/ stores	branches/ jumps
MIPS-15	15	6	4	2	3
MIPS-42	42	16	8	10	8
MIPS-191	191	129	26	2	34

Table 1: Processors used in the experiments

MIPS-15 and MIPS-42 are based on the original MIPS instruction encodings [9] for the implemented instructions, while MIPS-191 is based on the same instruction formats, but has different operation-code and function-code encodings, in order to use all possible instruction-encoding patterns and to define a very large ISA. This processor has the same functional units and the same pipeline structure as the previous two, except that it has more control signals going from the instruction-decoding PLA into the ALU in order to identify a different computation to be performed for each instruction.

When symbolically simulating the circuits and computing the correctness criterion, we used p-terms (represented as constants) for the data operands, immediate values, and instruction addresses. The register identifiers were represented as term-variables, and were encoded with BDD variables, as explained in Sect. 3. For a way to encode the initial state of pipeline latches, the reader is referred to [15].

We also exploited the idea of a non-consistent initial state (Sect. 5) by using a split Register File and a non-consistent ALU behavior, based on the Sequential Program Counter. The latter was achieved by letting the user specify nets that will be used as "imaginary" additional address inputs when generating the initial state of an EMM. We call such imaginary address inputs a *tag*.

We compared two encoding schemes for the distinct constants that are used for p-terms - consecutive constants, i.e., bit-vectors encoding consecutive binary numbers, and 1-hot constants which have a 1 in a single bit position and 0s in the other bit positions - see Tables 2 and 3, respectively. The experiments were performed on an IBM RS/6000 43P-140 with a 233MHz PowerPC 604e microprocessor, having 256 MB of physical memory, and running AIX 4.1.5.

Processor	Register File	Sequential PC used as tag for	CPU Time [s]	Memory [MB]	Max. BDD Nodes
MIPS-15	Unified	---	96	13.6	524,366
		ALU	37	7.6	262,181
	Split	---	89	7.6	262,226
		ALU	34	4.6	131,147
MIPS-42	Unified	---	325	25.6	1,048,666
		ALU	50	7,6	262,216
	Split	---	265	25.6	1,048,647
		ALU	37	7.6	262,211
MIPS-191	Unified	---	>881	>256	---
		ALU	49	7.9	262,199
	Split	---	>878	>256	---
		ALU	42	7.9	262,224

Table 2: Results when using consecutive constants for the p-terms

Processor	Register File	Sequential PC used as tag for	CPU Time [s]	Memory [MB]	Max. BDD Nodes
MIPS-15	Unified	---	50	7.6	262,235
		ALU	24	4.6	131,079
	Split	---	46	4.6	131,165
		ALU	21	3.1	65,589
MIPS-42	Unified	---	177	25.4	1,048,651
		ALU	36	4.6	131,141
	Split	---	145	13.6	524,363
		ALU	27	4.6	131,165
MIPS-191	Unified	---	>935	>256	---
		ALU	32	4.9	131,137
	Split	---	>1,710	>256	---
		ALU	27	4.9	131,147

Table 3: Results when using 1-hot constants for the p-terms

Tagging the ALU with the Sequential PC made the difference between impossible and possible when verifying the MIPS-like processor with 191 instructions, as shown in Table 4. The

use of constants for the values of instruction addresses (i.e., the values of the Sequential Program Counter) made the selection of the distinct uninterpreted function, to be performed by the ALU for each instruction, very efficient. As a result, the ALU was prevented from accumulating a single complex output term that is consistent across all instructions, which broke the effect of the forwarding logic feedback loops.

Using 1-hot constants, as opposed to consecutive constants, reduced the CPU time and memory with 1/3 and the maximum BDD node count by half. The reason is that the 1-hot constants lead to spreading the Boolean expressions, which result from deep nesting of *ITE* operators, across all bit positions of a bit-vector, thus avoiding the building of a single big BDD.

Splitting the Register File resulted in only a negligible CPU time reduction, when the above two ideas were employed. Additionally tagging the initial state of the Register File and the Data Memory with the Sequential PC did not lead to a significant performnce improvement. However, processing the ALU tag (i.e., the Sequential PC) first, immediately followed by the other input p-terms, when performing the address comparisons for computing the EMM's initial state, resulted in reducing the CPU time by half but did not change the memory consumption.

When generating the initial state for the pipeline latches, we obtained the best performance from the following BDD variable order: 1) Execution/Memory, 2) Memory/Write-Back, 3) Instruction-Decode/Execution, 4) Instruction-Fetch/Instruction-Decode, 5) Instruction Memory, where a "/" separates the two pipeline stages divided by the pipeline latch. The reason is that the Execution/Memory latch contains symbolic information for taken branches or jumps that will result in symbolic conditions for squashing the instructions in the preceding stages, i.e., this information will affect many instructions in flight. Hence, the BDD variables used for encoding the initial state of that pipeline latch have to precede the BDD variables that encode the initial state of the other latches in order to get smaller BDD sizes when computing the expression for the correctness criterion. Then, the Memory/Write-Back latch will affect the expressions for the data operands of all the subsequent instructions by means of the forwarding logic, so that this latch is ranked second. Similar reasoning explains the order of the other pipeline latches.

7 Conclusions

We showed that the logic of Positive Equality with Uninterpreted Functions scales very efficiently for verification of pipelined microprocessors with very large ISAs. Critical to that was the idea of functional units' behavior that is non-consistent across instructions, but consistent for the same instruction, based on the value of the Sequential PC. This idea made the difference between impossible and possible when verifying a 5-stage MIPS-like pipelined processor that implements 191 instructions and resulted in verification time and memory which are invariant with the size of the implemented ISA, when extending it from 42 to 191 instructions.

References

[1] R.E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *ACM Computing Serveys*, Vol. 24, No. 3 (September 1992), pp. 293-318.

[2] R.E. Bryant, S. German, and M.N. Velev, "Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions,"[2] *Computer-Aided Verification*, LNCS, Springer-Verlag, June 1999.

[3] R.E. Bryant, S. German, and M.N. Velev, "Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic,"[2] Technical Report CMU-CS-99-115, Carnegie Mellon University, 1999.

[4] J.R. Burch, and D.L. Dill, "Automated Verification of Pipelined Microprocessor Control," *CAV'94*, D.L. Dill, ed., LNCS 818, Springer-Verlag, June 1994, pp. 68-80.

[5] J.R. Burch, "Techniques for Verifying Superscalar Microprocessors," *33rd Design Automation Conference (DAC'96)*, June 1996, pp. 552-557.

[6] Y.-A. Chen, "Arithmetic Circuit Verification Based on Word-Level Decision Diagrams," Ph.D. thesis, School of Computer Science, Carnegie Mellon University, May 1998.

[7] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, "BDD Based Procedures for a Theory of Equality with Uninterpreted Functions," *CAV'98*, Springer-Verlag, June 1998.

[8] A.J. Isles, R. Hojati, and R.K. Brayton, "Computing Reachable Control States of Systems Modeled with Uninterpreted Functions and Infinite Memory," *CAV'98*, Springer-Verlag, June 1998.

[9] G. Kane, and J. Heinrich, *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[10] G. Nelson, and D.C. Oppen, "Simplification by Cooperating Decision Procedures," *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 2, October 1979, pp. 245-257.

[11] M. Pandey, "Formal Verification of Memory Arrays," Ph.D. thesis, School of Computer Science, Carnegie Mellon University, May 1997.

[12] D.A. Patterson, and J.L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2nd edition, Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[13] M.N. Velev, R.E. Bryant, and A. Jain, "Efficient Modeling of Memory Arrays in Symbolic Simulation,"[2] *CAV'97*, O. Grumberg, *ed.*, LNCS 1254, Springer-Verlag, June 1997, pp. 388-399.

[14] M.N. Velev, and R.E. Bryant, "Efficient Modeling of Memory Arrays in Symbolic Ternary Simulation,"[2] *TACAS'98*, B. Steffen, *ed.*, LNCS 1384, Springer-Verlag, March-April 1998, pp. 136-150.

[15] M.N. Velev, and R.E. Bryant, "Bit-Level Abstraction in the Verification of Pipelined Microprocessors by Correspondence Checking,"[2] *FMCAD'98*, G. Gopalakrishnan and P. Windley, *eds.*, LNCS 1522, Springer-Verlag, November 1998, pp. 18-35.

2. Available from: `http://www.ece.cmu.edu/~mvelev`