Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction[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 extend the Burch and Dill flushing technique [6] for formal verification of microprocessors to be applicable to designs where the functional units and memories have multicycle and possibly arbitrary latency. We also show ways to incorporate exceptions and branch prediction by exploiting the properties of the logic of Positive Equality with Uninterpreted Functions [4][5]. We study the modeling of the above features in different versions of dual-issue superscalar processors.

1 Introduction

In order for formal methods to scale for verification of modern microprocessors, they need to be applicable easily and with a high degree of automation to designs with multicycle functional units, multicycle memories, exceptions, and branch prediction. Burch and Dill's verification methodology has the potential to be highly automatic, but has previously been applied only to designs with single-cycle functional units and memories that produce their results instantaneously [5][6][7][21]. The approach is very elegant in using *flushing* of the processor—feeding it with bubbles until all instructions in flight complete their execution—in order to compute an abstraction function mapping Implementation states to a Specification state. (The difference between a bubble and a nop is that a bubble does not modify any user-visible state, while a nop increments the PC.) The correctness criterion is a commutative diagram, stating that an application of the transition function of the Implementation followed by flushing should produce the same user-visible state as first flushing the Implementation and then using the resultant user-visible state to apply the transition function of the Specification between 0 and k times, where k is the issue-width of the Implementation. As observed and exploited by Burch in his controlled flushing [7], we can change the logic during flushing, since the only purpose of that logic is to compute an abstraction function. Having an improper abstraction function will not compromise the verification and can only result in a false negative.

The same correctness criterion has been adopted by the theorem-proving community and applied to an out-of-order design with exceptions and interrupts by Sawada and Hunt [17] and to an out-of-order design with only arithmetic instructions by Hosabettu, Srivas and Gopalakrishnan [13]. However, the former approach requires the user to manually build an intermediate abstraction of the processor and to define a large number of lemmas (Sawada defined nearly 4,000), necessary for the correctness proof. The latter approach requires the user to manually define a set of completion functions, one per unfinished instruction in flight, describing how that instruction will be completed, given that all the instructions that it has data dependencies on have already been completed. Furthermore, the user has to manually define a way to compose these completion functions in order to form the abstraction function for the processor. Both of these theorem-proving methods require months of manual work for complex designs, i.e., they are not automatic.

Not only has the Burch and Dill flushing technique not been applied to processors with multicycle functional units, but Hosabettu, Srivas and Gopalakrishnan [12] have claimed that it has the drawback of being hard to use for pipelines with indeterminate latency, particularly where an ALU computation might have a data-dependent duration or a memory hierarchy of multiple levels might have a non-deterministic delay.

In this work we extend Burch and Dill's flushing-based methodology to be applicable to microprocessors with functional units and memories of multicycle and possibly arbitrary latency. We also model exceptions and branch prediction. Our most complex designs have 10 abstract instruction types. They have two completely functional pipelines, each consisting of five stages, for a total of up to 10 instructions in flight. Therefore, exhaustive binary simulation must consider 10^{10} instruction sequences of 10 instructions each. Furthermore, accounting for possible data dependencies, raised exceptions, correctness/incorrectness of the branch predictions, and multicycle computations will make that number significantly higher. Even directed simulation will probably find it very hard, if at all possible, to generate all interesting instruction sequences for such a design. However, we were able to formally verify it in less than 44 minutes of CPU time.

2 Background

The key to our success is a very efficient decision procedure [21] for the logic of Equality with Uninterpreted Functions and Memories (EUFM) [6], which exploits the properties of Positive Equality [4][5] and the e_{ij} encoding [10] to generate a propositional formula, which is then evaluated with BDDs [3] or SAT-checkers. We found BDDs to be unmatched by SAT-checkers and SVC [18] (a decision procedure for the logic of EUFM that does not exploit Positive Equality) when verifying correct designs. However, SAT-checkers outperform BDDs on buggy processors.

The syntax of EUFM [6] includes terms and formulas. A term can be an Uninterpreted Function (UF) applied on a list of argument terms, a domain variable, or an *ITE* operator selecting between two argument terms based on a controlling formula, such that *ITE*(*formula*, *term*1, *term*2) will evaluate to *term*1 when *formula* = **true** and to *term*2 when *formula* = **false**. A formula can be an Uninterpreted Predicate (UP) applied on a list of argu-

1. This research was supported in part by the SRC under contract 99-DC-068.

DAC 2000, Los Angeles, California

ment terms, a propositional variable, an *ITE* operator selecting between two argument formulas based on a controlling formula, or an equation (equality comparison) of two terms. Formulas can be negated and connected by Boolean connectives. We will refer to both terms and formulas as expressions.

UFs and UPs are used to abstract away the implementation details of functional units by replacing them with "black boxes" that satisfy no particular properties other than that of *functional consistency*. Namely, that the same combinations of values to the inputs of the UF (or UP) produce the same output value. Then, it no longer matters whether the original functional unit is an adder or a multiplier, etc., as long as the same UF (or UP) is used to replace it in both the Implementation and the Specification. Note that in this way we will prove a more general problem—that the processor is correct for any implementation of its functional units. However, that more general problem is much easier to prove (see [20] for the scaling of the correctness proof for a processor with an actual bit-level implementation of its ALU).

The syntax for terms can be extended to model memories by means of the functions *read* and *write*, where *read* takes 2 argument terms serving as memory and address, while *write* takes 3 argument terms serving as memory, address, and data. Both functions return a term. Also, they satisfy the forwarding property of the memory semantics, *read*(*write*(*mem*, *waddr*, *wdata*), *raddr*) is equivalent to *ITE*((*raddr* = *waddr*), *wdata*, *read*(*mem*, *raddr*)), in addition to the property of functional consistency. Versions of *read* and *write* that extend the syntax for formulas can be defined similarly, such that the former returns a formula, while the latter takes a formula as its third argument.

Three possible ways to impose the property of functional consistency of UFs and UPs are Ackermann constraints [1], nested *ITE*s [4][5][20], and "pushing-to-the-leaves" [21]. The Ackermann scheme replaces each UF (UP) application in the EUFM formula F with a new domain variable (propositional variable) and then adds external consistency constraints. For example, the UF application $f(a_1, b_1)$ will be replaced by a new domain variable c_1, another application of the same UF, $f(a_2, b_2)$, will be replaced by a new domain variable c_2. Then, the resultant EUFM formula F' will be extended as $[(a_1 = a_2) \wedge (b_1 = b_2) \Rightarrow (c_1 = c_2)] \Rightarrow F'$. In the nested *ITE*s scheme, the first application of the UF above will still be replaced by a new domain variable c_1. However, the second one will be replaced by $ITE((a_2 = a_1) \wedge (b_2 = b_1), c_1, c_2)$, where c_2 is a new domain variable. A third one, $f(a_3, b_3)$, will be replaced by $ITE((a_3 = a_1) \wedge (b_3 = b_1), c_1, ITE((a_3 = a_2) \wedge (b_3 = b_2), c_2, c_3))$, where c_3 is a new domain variable, and so on. Similarly for UPs. Note that the nested *ITE*s approach keeps the consistency information located in the internal structure of the formula. In the pushing-to-the-leaves scheme, a UF application is pushed towards the leaves of its argument nested *ITE* expressions until all arguments become domain variables. Then, each UF application on a unique list of argument domain variables is replaced with a new domain variable. For example, $f(ITE(d, a_1, a_2), b)$ will be transformed into $ITE(d, f(a_1, b), f(a_2, b))$ and then into $ITE(d, c_1, c_2)$, where c_1 and c_2 are new domain variables replacing $f(a_1, b)$ and $f(a_2, b)$, respectively. Although this scheme has the potential to result in a term blow up, it has the advantage that it does not create equations between argument terms, as the previous two schemes do. Note also that the pushing-to-the-leaves scheme results in a conservative approximation to functional consistency in that it satisfies the Ackermann functional consistency constraints only for syntactically identical terms.

Positive Equality allows the identification of two types of terms in the structure of an EUFM formula—those which appear only in positive equations and are called *p-terms*, and those which can appear in both positive and negative equations and are called *g-terms* (for general terms). A negative equation is one which appears under an odd number of negations or as part of the controlling formula for an *ITE* operator. The computational efficiency from exploiting Positive Equality is due to a theorem which states that the truth of an EUFM formula under a maximally diverse interpretation of the p-terms implies the truth of the formula under any interpretation. The classification of p-terms vs. g-terms is done before UFs and UPs are eliminated by nested *ITE*s, such that if an UF is classified as a p-term (g-term), the new domain variables generated for its elimination are also considered to be p-terms (g-terms). After the UFs and the UPs are eliminated, a maximally diverse interpretation is one where the equality comparison of two syntactically identical (i.e., exactly the same) domain variables evaluates to **true**, that of a p-term domain variable with a syntactically distinct domain variable evaluates to **false**, and that of a g-term domain variable with a syntactically distinct g-term domain variable could evaluate to either **true** or **false** and can be encoded with a dedicated Boolean variable—an e_{ij} variable [10]. An alternative encoding has been proposed by Pnueli *et al.* [15].

In order to fully exploit the benefits of Positive Equality, the designer of an abstract processor model has to use a set of suitable abstractions and conservative approximations. For example, an equality comparison of two data operands, as used to determine the condition to take a branch-on-equal instruction, must be abstracted with an UP in both the Implementation and the Specification, so that the data operand terms will not appear in negated equations but only as arguments to UPs and UFs and will be classified as p-terms. Similarly, a Finite State Machine (FSM) model of a memory, which is a conservative approximation of an actual memory, has to be employed to model the Data Memory, so that its addresses, which are produced by the ALU and also serve as data operands, can be classified as p-terms. The result is that data values produced by the Register File, the ALU, and the Data Memory, as well as the PC values, can be classified as p-terms. Only the register identifiers, whose equations control forwarding and stalling conditions that are negated, are classified as g-terms. Finally, and this is done automatically [21], the conservative approximation of pushing-to-the-leaves has to be used for eliminating reads from the initial state of memories addressed by g-terms, such as the Register File, in order to reduce the number of distinct equations between g-terms, i.e., to reduce the number of e_{ij} Boolean variables.

3 Modeling Multicycle Functional Units

We replace multicycle functional units with "place holders," which are implemented with the constructs of EUFM and exhibit enough of the timing characteristics of the original functional units, such that the correctness of the abstract processor with place holders will imply the correctness of the actual Implementation processor with the original functional units. For example, we can model the timing behavior of a functional unit with a fixed latency of n cycles by a chain of n-1 latches situated between the two pipeline latches that limit the stage of the functional unit, and a single UF abstracting the functionality of the unit. The chain of latches will be used to delay the signal that controls the updating of user-visible state with the result produced by the functional unit (see [22] for details). However, it will be cumbersome to use such a model in processors that have many multicycle instructions, each of a different fixed latency. Most importantly, this model is not applicable for functional units where the latency depends on the values of the input operands or on arbitrary environment factors, e.g., a memory system with cache-coherence mechanisms [11], where a data value might be locked in order to be modified by another processor. It is such functional units and memory systems that other researchers [12] have found to make hard the application of the Burch and Dill method to real processors.

We resolve this problem by using a technique that we call *accelerated flushing*. Namely, during the one cycle of regular

symbolic simulation of the Implementation, we model the indeterminate outcome of possibly completing the computation of a multicycle functional unit by a new Boolean variable. Then, during flushing we force the functional unit to complete its computations on every clock cycle. That is, if the original computation was not finished during the single cycle of regular symbolic simulation of the Implementation, it will be definitely completed on the first cycle of flushing and a new computation will be completed on each subsequent cycle of flushing. Such a signal, controlling the completion of multicycle computations, can be generated with the circuit on Fig. 1.

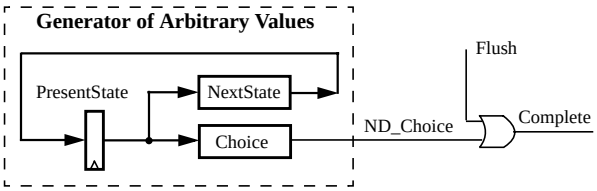


Figure 1. A generator of new Boolean variables, extended with an OR-gate in order to produce signal *Complete* that controls the computation completion for a given multicycle functional unit, according to the accelerated flushing technique.

Observe that the present state of the Finite State Machine (FSM) above and the output of UF *NextState* are not used in equality comparisons in the circuit, so that they will be classified as p-terms when the EUFM formula is translated to a propositional formula. Then, UF *NextState* will map each input p-term domain variable to a new p-term domain variable for the corresponding output value. Therefore, the state of the FSM will be updated with a new p-term domain variable on every cycle. Each of these p-term domain variables will be mapped to a new Boolean variable by UP *Choice*, so that the FSM will generate a sequence of new Boolean variables. However, only the first one will be passed on to signal *Complete* during the one cycle of regular simulation when input *Flush* is set to **false**. During flushing, input *Flush* will be **true**, and so will be signal *Complete*. Similarly, we can use an UF that depends on the present state of the FSM on Fig. 1 in order to generate a sequence of new domain variables. We call such FSMs *generators of arbitrary values*.

When designing place holders for multicycle functional units of arbitrary latency, we assume that the computation semantics can be expressed with a combinational functional unit abstracted with UF *ALU*—see Fig. 2. The FSM on Fig. 2.b is used to abstract the timing of the functional unit. Conceptually, a multicycle computation can be in one of 3 abstract states—in its first cycle, in flight (i.e., has already been executed for at least one cycle), or has completed but is stalled by the next pipeline stages (state *DoneButStalled*). We latch the output of UF *ALU* when the computation is in its first cycle, assuming the data inputs have the correct values only during that cycle.

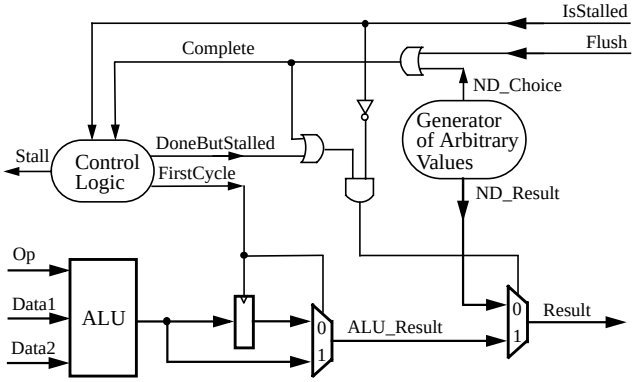
The 3 abstract states on Fig. 2.b can be encoded with 2 state bits that should have arbitrary initial Boolean values, so that the control FSM could be in any state initially. Signal *Stall* is used to stall previous pipeline stages when the computation has not completed (both *Complete* and *DoneButStalled* are **false**) or the result cannot be accepted (*IsStalled* is **true**). Signal *Complete* is generated according to Fig. 1. The control logic of the place holder generates the signals *FirstCycle* and *DoneButStalled* which indicate that the control FSM is in the corresponding state of its state transition diagram on Fig. 2.b.

In order to model computations that are guaranteed to complete on the clock cycle when they start, the OR gate driving signal *Complete* can be extended with extra inputs that account for such conditions. When the next pipeline stages do not have a mechanism to stall the multicycle functional unit, the control logic of the place holder can be simplified by setting *IsStalled* to **false** and removing state *DoneButStalled* from the state transition diagram on Fig. 2.b. Furthermore, a raised squash signal that affects the multicycle functional unit should set signal *Complete* to **false** and cause the state transition diagram on Fig. 2.b to transition to state *FirstCycle*.

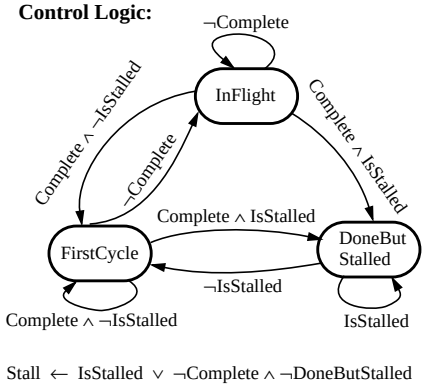
The correct result *ALU_Result* will be passed to the output *Result* of the place holder only when the stages ahead are ready to accept the result (*IsStalled* is **false**) and the multicycle computation has completed in either the present cycle (*Complete* is **true**) or previously (*DoneButStalled* is **true**). Otherwise, the output of the place holder will get a new domain variable produced by the generator of arbitrary values at its output *ND_Result*. The effect is analogous to using Xs is symbolic ternary simulation in order to express ambiguity [19]. Therefore, the place holder on Fig. 2 is a conservative approximation of a multicycle functional unit of arbitrary latency.

The same functional unit is modeled in the Specification processor by UF *ALU* only, but not the extra logic required to implement the place holder, as the Specification defines the semantics of the instructions regardless of their timing. Note that by making the op-code term *Op* one of the inputs to UF *ALU*, we model a potentially different computation for each instruction. UF *ALU* has to be replaced by a memory model, e.g., the FSM-based one from [4][5][21], when implementing a place holder for a memory system of arbitrary latency.

The place holder on Fig. 2 is based on the following assumptions about the original functional unit that it abstracts: 1) the functional unit will not deadlock and will eventually complete



(a)



(b)

Figure 2. (a) Implementation of a place holder for a functional unit of arbitrary latency; (b) State transition diagram for the control logic of a place holder that can be stalled, as determined by signal *IsStalled*.

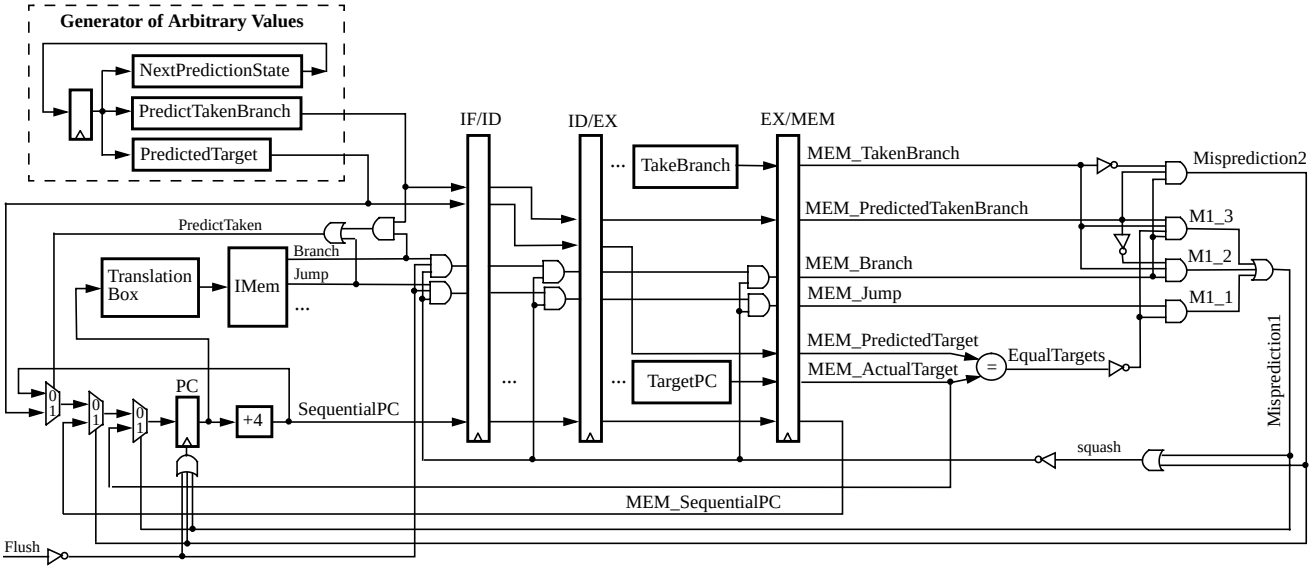


Figure 3. Branch Prediction in a single-issue 5-stage pipelined DLX. The logic not directly related with updating the PC is omitted.

every multicycle computation; 2) the functional unit has a mechnism to store its input values if they are guaranteed to be available only on the first cycle of a multicycle computation; 3) the functional unit has a mechanism to store the result of its computation, until the stages ahead are ready to accept that result (this property should hold only for functional units that can be stalled); and, 4) the functional unit will discard an on-going computation and will be ready to begin a new one on the next clock cycle if a controlling squash signal is raised on the present clock cycle (this property should hold only for functional units that can be affected by a squash signal). The original functional unit has to be formally verified for satisfying the above properties, e.g., by model checking [8].

4 Modeling Exceptions

For every functional unit (abstracted with either an UF or an UP) that can generate an exception we introduce an UP that depends on the same inputs as the functional unit and produces a Boolean signal indicating whether an exception was raised. If the functional unit can be a source of several kinds of exceptions, we can introduce an UF that again depends on the same inputs and produces a term that indicates the type of the raised exception. Processors with exceptions have user-visible state elements that contain exception status and recovery information. Our architecture has 3 exception status registers—indication whether an exception was raised by each of the Instruction Memory, the ALU, and the Data Memory—as well as an ExceptionPC latch, containing the PC of the excepting instruction.

Exceptions result in squashing of subsequent instructions in flight and branching to an exception handler, whose address depends on the exception type in our architecture. The exception-handlers are implemented in software and are assumed to be correct. A return-from-exception instruction has the effect of jumping to the ExceptionPC and clearing the exception status registers. Exceptions are modeled in both the Implementation and the Specification processors, since the instruction semantics depends on exceptions and the exception status registers are user-visible state elements.

5 Modeling Branch Prediction

We use a generator of arbitrary values in order to abstract the Branch Predictor in the implementation processor—see Fig. 3. Every clock cycle, this generator produces: 1) a new Boolean variable at the output of UP *PredictTakenBranch*, serving as a prediction for the taken/not-taken direction of a newly fetched branch (jumps are always taken in our architecture); and, 2) a new domain variable at the output of UF *PredictedTarget*, serving as a prediction for the target of a branch or a jump. What is verified is that if the Implementation updates speculatively the PC according to a prediction made in the Fetch stage of the pipeline and the prediction is incorrect as determined when the actual direction and target become available after the Execution stage, then the processor has a mechanism to correct the misprediction. The Specification does not include a Branch Predictor, which is not part of the user-visible state and is irrelevant for defining the correct instruction semantics. Note that if an Implementation processor is verified with completely arbitrary predictions for the direction of a branch and for the target of a branch or a jump, then that processor will be correct for any actual implementation of the Branch Predictor.

Correcting branch mispredictions requires a negated equality comparison of the actual and predicted targets, so that they will be classified as g-terms. These terms will update the PC and will address the Instruction Memory (IMem). That will result in dependencies of the newly fetched instructions on e_{ij} variables, encoding equality comparisons of such g-term domain variables, when the nested *ITE*s scheme is used to enforce consistent initial state of memories addressed by g-terms. These dependencies will affect the entire final formula and will increase the complexity of the evaluation. Alternatively, we could use the pushing-to-the-leaves strategy in order to enforce a consistent initial state for the IMem. That would avoid the e_{ij} variables, but will result in a term blow up, especially for wide processors.

Our solution is to introduce a “translation box” for the address terms of the IMem (see Fig. 3), i.e., an UF that will translate its input terms to output p-terms. Indeed, the output terms of that UF are not be used in any equations, so that they will be classified as p-terms. Note that such a translation box has to be used in both the Implementation and the Specification. An UF serving as a translation box is a conservative approximation—if a processor is verified with such UFs, the processor will be correct for any implementation of these UFs, including the identity function that connects the input to the output. However, the translation boxes help us to better exploit the computational efficiency of Positive Equality in modeling Branch Prediction. Note that incorporating Branch Prediction does not require any changes to

Processor	Final *Vp*	Final *Vg*				BDD Variables			Max. BDD Nodes	Memory [MB]	CPU Time [s]
		Src Regs	Dest Regs	Other	Total	e_{ij}	Other	Total			
1×DLX-C	52	7	6	0	13	27	36	63	2,127	5.7	0.24
1×DLX-C-BP	49	7	6	10	23	49	41	90	4,004	5.9	0.50
1×DLX-C-MC	55	9	6	0	15	36	47	83	4,650	5.9	0.86
1×DLX-C-EX	69	7	6	0	13	27	64	91	7,482	6.5	1.78
1×DLX-C-MC-EX	72	9	6	0	15	36	77	113	20,624	7.4	6
1×DLX-C-MC-EX-BP	62	9	6	10	25	62	81	143	22,270	8.3	6
2×DLX-CA	87	13	12	0	25	116	46	162	24,227	11	6
2×DLX-CA-BP	83	13	12	15	40	170	68	238	48,076	15	20
2×DLX-CA-MC	92	17	12	0	29	146	76	222	83,106	15	46
2×DLX-CA-EX	112	15	12	0	27	139	89	228	345,786	17	410
2×DLX-CA-MC-EX	120	17	12	0	29	146	125	271	546,502	20	814
2×DLX-CA-MC-EX-BP	102	17	12	15	44	211	131	342	779,495	23	979
2×DLX-CC	100	13	12	0	25	116	57	173	51,826	16	20
2×DLX-CC-BP	96	13	12	20	45	209	82	291	113,330	16	81
2×DLX-CC-MC	115	17	12	0	29	146	94	240	182,257	17	164
2×DLX-CC-EX	131	13	12	0	25	122	103	225	430,613	19	581
2×DLX-CC-MC-EX	137	17	12	0	29	146	150	296	1,394,618	37	3,221
2×DLX-CC-MC-EX-BP	121	17	12	20	49	260	158	418	986,740	31	2,635

Table 1. Statistics for the number of domain variables and the resources needed for the BDD evaluation of the final propositional formula. The memory and CPU time are reported for the sequence of symbolic simulation, translation of the EUFM formula to a propositional one, and the evaluation of the latter by BDDs. V_p designates the set of p-term domain variables, while V_g the set of g-term domain variables. "Src Regs" stands for source registers, while "Dest Regs" for destination registers. The category "Other" of the final V_g consists of the predicted and actual targets of branch and jump instructions.

the Specification processor other than a translation box for the address terms of the Instruction Memory, i.e., the implementation details of Branch Prediction remain invisible to the Specification. An alternative way to incorporate branch prediction in an abstract processor is studied in [22].

6 Experimental Results

We started with three base abstract processor models: **1×DLX-C**, a single-issue pipelined DLX [11] with one complete pipeline, that can execute the 7 abstract instruction types—register-register, register-immediate, load, store, branch, jump, and nop; **2×DLX-CA,** a dual-issue superscalar DLX with one complete pipeline and another capable of executing only arithmetic (register-register and register-immediate) instructions, such that between 0 and 2 instructions can be fetched per cycle—this design is comparable to Burch's [7] and is inspired by the Intel Pentium processor; **2×DLX-CC**, a dual-issue superscalar DLX with two complete pipelines, i.e., it has no structural hazards but 4 load interlocks, so that again between 0 and 2 instructions can be fetched per cycle.

These models were extended with versions that implement: 1) branch prediction, designated with "**-BP**"; 2) multicycle functional units, marked with "**-MC**," where the Instruction Memory, the ALU in the Execution stage, and the Data Memory were modeled as having an arbitrary latency, such that two new abstract instruction types were introduced—multicycle register-register and multicycle register-immediate (all other computations that use the ALU were modeled to complete in the clock cycle when they start)—such that the 2 new instructions could be executed only by the complete pipeline in 2×DLX-CA-MC; and, 3) exceptions, "**-EX**," where the Instruction Memory, the ALUs, and the Data Memory could generate exceptions and the new instruction return-from-exception (executed only by the complete pipeline in 2×DLX-CA-EX) was implemented to clear the 3 Exception Status bits and jump to the Exception-PC. Then, we created hybrid versions, "**-MC-EX**" and "**-MC-EX-BP**," which combine several of the above features. Since the second pipeline in 2×DLX-CA could execute only arithmetic instructions, it had a vacuous Memory stage, so that "-EX" versions of that processor could have Data Memory exceptions generated only by the first pipeline. Similarly, "-MC" versions of the same processor could have stalling of the pipeline stages before the Memory stage only due to Data Memory accesses of arbitrary latency of the first pipeline. In "-BP" versions of the dual-issue processors, branch predictions were made for the two newly fetched instructions; in "-EX" versions of these models, either of the two new instruction fetches could generate an Instruction Memory exception; and, in "-MC" versions, either of the two instruction fetches could be invalid, due to an unfinished Instruction Memory access of arbitrary latency. All processors were modeled in the style described in [21].

The results are presented in Table 1. The experiments were performed on a 336 MHz Sun4 with 1.2 GB of memory. The Colorado University BDD package [9] and the sifting dynamic BDD variable reordering heuristic [16] were used to evaluate the final propositional formula. Burch's controlled flushing [7] was employed for all of the designs. As the table shows, our verification times range from less than a second for the single-issue case,

up to a little less than 54 minutes for one of the most complex dual-issue superscalar designs. The memory requirement varies between 5.7 and 37 MB. The number of propositional variables ranges from 63 to 418, with between 27 and 260 comprising the e_{ij} variables encoding the equality comparisons of g-term domain variables. The number of the p-term domain variables is between 2 and 5 times greater than that of the g-term domain variables.

Analyzing the results from the benchmarks where a base model is extended with a single feature, we can see that adding exceptions leads to the greatest increase in complexity. This can be explained with several characteristics of these designs. First, their user-visible state contains 4 extra latches—an Exception-PC, and 3 Exception Status bits—so that extra equality comparisons for the final states of these latches are added to the EUFM formula for the correctness criterion. Second, these designs require more Boolean variables as either part of the initial state of their pipeline latches or as outputs of the UPs, indicating that a certain type of exception has been raised for a particular instruction in flight. Indeed, if we compare the category of "Other BDD Variables" in Table 1 for the "-EX" models vs. their corresponding base model, and for the "-MC-EX" models vs. their corresponding "-MC" model, we will see that the number of such Boolean variables increases significantly, approaching the double of the original number. Third, the models with exceptions exhibit the greatest increase in the branching behavior of the program execution. Namely, each raised exception results in squashing of all subsequent instructions in flight, jumping to the address of the corresponding exception-handler, and conditional modification of all user-visible state elements. Hence, the term blow up in the correctness criterion formula.

Modeling multicycle functional units—an Instruction Memory, an ALU, and a Data memory of arbitrary latency, as explained in Sect. 3—results in a slight increase in the number of "Other BDD Variables." This can be attributed to the Boolean variables used as outputs of UPs that produce the non-deterministic choice for completing ALU computations or memory accesses of arbitrary latency. The slightly more e_{ij} BDD variables are due to the increased ambiguity of the instruction flow in the processor.

Incorporating Branch Prediction results in the least increase in evaluation complexity, compared to "-MC" and "-EX" extensions. Between 10 and 20 extra g-term domain variables are created (relative to the model that was extended), serving as predicted and actual targets for branch/jump instructions. This decreases the number of p-term domain variables, as the actual targets are no longer classified as p-terms. Potentially, each of the extra g-term domain variables can be compared for equality against all other extra g-term domain variables (unless simplifications take effect), when the equality comparisons for the final state of the PC are formed as part of the correctness criterion. That explains the significant increase in the number of e_{ij} Boolean variables. However, most of these e_{ij} variables do not affect the instruction flow but only the final equality comparisons, so that their effect is relatively limited and our BDD variable ordering heuristic [21], combined with sifting [16], worked very well.

Although 2×DLX-CC-MC-EX requires 122 fewer BDD variables than 2×DLX-CC-MC-EX-BP, the verification of the former takes more CPU time because of variations in the performance of the sifting heuristic when used on different Boolean formulas. However, neither benchmark can be formally verified without the sifting heuristic—the experiments ran out of memory after more than 24 hours of CPU time.

Additional details of this research and techniques for accelerating the verification are presented in [22].

7 Conclusions

We were able to formally verify a dual-issue superscalar DLX processor with two complete pipelines, where the Instruction Memory, the ALUs, and the Data Memory could each have an arbitrary latency and possibly generate an exception, as well as with branch prediction of the two newly fetched instructions, in less than 44 minutes of CPU time. We believe that the success of our approach in the extremely efficient formal verification of a single-issue pipelined DLX with multicycle functional units, exceptions, and branch prediction (1×DLX-C-MC-EX-BP)—requiring 6 seconds of CPU time and 8.3 MB of memory—will enable the formal verification of real pipelined processors with the same features, e.g., the ARM [2] and the M•CORE [14].

References

[1] W. Ackermann, *Solvable Cases of the Decision Problem*, North-Holland, Amsterdam, 1954.

[2] *ARM Technical Reference Manuals & Data Sheets*, `http://www.arm.com`.

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

[4] R.E. Bryant, S. German, and M.N. Velev, "Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions,"[2] *Computer-Aided Verification (CAV'99)*, N. Halbwachs and D. Peled, eds., LNCS 1633, Springer-Verlag, June 1999, pp. 470-482.

[5] 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.

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

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

[8] E.M. Clarke, Jr., O. Grumberg, and D.A. Peled, *Model Checking*, MIT Press, Cambridge, MA, 2000.

[9] CUDD-2.3.0, `http://vlsi.colorado.edu/~fabio`.

[10] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, "BDD Based Procedures for a Theory of Equality with Uninterpreted Functions," *Computer-Aided Verification (CAV'98)*, A.J. Hu and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 244-255.

[11] J.L. Hennessy, and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd edition, Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[12] R. Hosabettu, M. Srivas, and G. Gopalakrishnan, "Decomposing the Proof of Correctness of Pipelined Microprocessors," *Computer-Aided Verification (CAV'98)*, A.J. Hu and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 122-134.

[13] R. Hosabettu, M. Srivas, and G. Gopalakrishnan, "Proof of Correctness of a Processor with Reorder Buffer Using the Completion Functions Approach," *Computer-Aided Verification (CAV'99)*, N. Halbwachs and D. Peled, eds., LNCS 1633, Springer-Verlag, June 1999, pp. 45-59.

[14] *M•CORE: microRISC Engine Programmer's Manual*, `http://www.motorola.com/SPS/MCORE`.

[15] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel, "Deciding Equality Formulas by Small-Domain Instantiations," *Computer-Aided Verification (CAV'99)*, LNCS, Springer-Verlag, June 1999, pp. 455-469.

[16] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *International Conference on Computer-Aided Design (ICCAD'93)*, November 1993, pp. 42-47.

[17] J. Sawada, and W.A. Hunt, Jr., "Processor Verification with Precise Exceptions and Speculative Execution," *Computer-Aided Verification (CAV'98)*, A.J. Hu and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 135-146.

[18] Stanford Validity Checker, `http://sprout.stanford.edu`.

[19] M.N. Velev, and R.E. Bryant, "Incorporating Timing Constraints in the Efficient Memory Model for Symbolic Ternary Simulation,"[2] *International Conference on Computer Design (ICCD '98)*, October 1998, pp. 400-406.

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

[21] M.N. Velev, and R.E. Bryant, "Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic,"[2] *Correct Hardware Design and Verification Methods (CHARME '99)*, L. Pierre and T. Kropf, eds., LNCS 1703, Springer-Verlag, September 1999, pp. 37-53.

[22] M.N. Velev, and R.E. Bryant, "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction,"[2] Technical Report CMU-CS-00-116, Carnegie Mellon University, 2000.

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