Automatic Formal Verification of Liveness for Pipelined Processors with Multicycle Functional Units

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

Abstract. Presented is a highly automatic approach for proving bounded liveness of pipelined processors with multicycle functional units, without the need for the user to set up an inductive argument. Multicycle functional units are abstracted with a placeholder that is suitable for proving both safety and liveness. Abstracting the branch targets and directions with arbitrary terms and formulas, respectively, that are associated with each instruction, made the branch targets and directions independent of the data operands. The observation that the term variables abstracting branch targets of newly fetched instructions can be considered to be in the same equivalence class, allowed the use of a dedicated fresh term variable for all such branch targets and the abstraction of the Instruction Memory with a generator of arbitrary values. To further improve the scaling, the multicycle ALU was abstracted with a placeholder without feedback loops. Also, the equality comparison between the terms written to the PC and the dedicated fresh term variable for branch targets of new instructions was implemented as part of the circuit, thus avoiding the need to apply the abstraction function along the specification side of the commutative diagram for liveness. This approach resulted in 4 orders of magnitude speedup for a 5-stage pipelined DLX processor with a 32-cycle ALU, compared to a previous method for indirect proof of bounded liveness, and scaled for a 5-stage pipelined DLX with a 2048-cycle ALU.

1 Introduction

Previous work on microprocessor formal verification has almost exclusively addressed the proof of *safety*—that if a processor does something during a step, it will do it correctly—as also observed in [2], while ignoring the proof of *liveness—that a processor will complete a new instruction after a finite number of steps*. Several authors used theorem proving to check liveness [15][16][17][19][23][28][32][34], but invested extensive manual work. This paper is the first to prove liveness for pipelined processors with multicycle functional units in an automatic way.

Functional units in recent state-of-the-art processors usually have latencies of up to 20–30 cycles, and rarely up to 200 cycles, but it is expected that the memory latencies in next generation high-performance designs will reach 1,000 cycles [13]. Thus, the need to develop automatic techniques to prove the liveness of pipelined processors where the functional units can have latencies of up to thousands of cycles.

In the current paper, the implementation and specification are described in the high-level hardware description language HDL [46], based on the logic of Equality with Uninterpreted Functions and Memories (EUFM) [7]. In EUFM, word-level values are abstracted with terms (see Sect. 4) whose only relevant property is that of equality with other terms. Restrictions on the style for describing high-level processors [35][36] reduced the number of terms that appear in both positive and negated equality comparisons—and are so called *g-terms* (for general terms)—and increased the number of terms that appear only in positive polarity—and are so called *p-terms* (for positive terms). The property of Positive Equality [35][36] allowed us to treat syntactically dif-

ferent p-terms as not equal when evaluating the validity of an EUFM formula, thus achieving significant simplifications and orders of magnitude speedup. (See [5] for a correctness proof.)

The formal verification is done with an automatic tool flow, consisting of: 1) the term-level symbolic simulator TLSim [46], used to symbolically simulate the implementation and specification, and produce an EUFM correctness formula; 2) the decision procedure EVC [46] that exploits Positive Equality and other optimizations to translate the EUFM correctness formula to an equivalent Boolean formula, which has to be a tautology in order for the implementation to be correct; and 3) an efficient SAT-solver. This tool flow was used at Motorola [18] to formally verify a model of the M•CORE processor, and detected bugs.

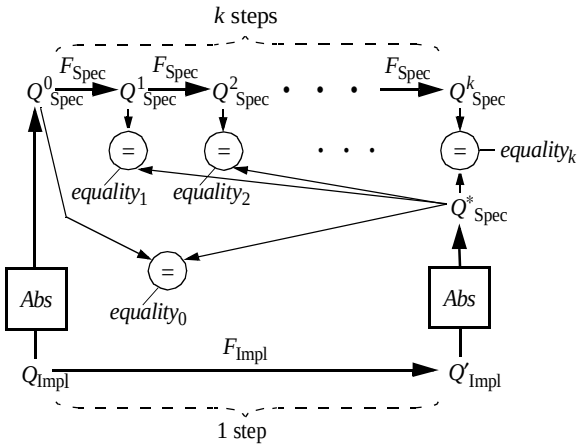
The rest of the paper is organized as follows. Sect. 2 defines safety and liveness. Sect. 3 discusses related work. Sect. 4 summarizes the logic of EUFM, the property of Positive Equality, and efficient translations from EUFM to CNF. Sect. 5 presents a previous indirect method for proving liveness of pipelined processors by exploiting Positive Equality. Sect. 6 explains the application of that indirect method to proving the liveness of pipelined DLX processors having ALUs with latencies of up to 2048 cycles. Sect. 7 describes an abstraction for multicycle ALUs that is applicable to proving both safety and liveness of pipelined processors. The next three sections present optimizations that speed up the automatic formal proof of liveness for pipelined processors with multicycle functional units: Sect. 8 describes techniques for abstracting the branch targets and directions of instructions; Sect. 9 makes the observation that the branch targets of newly fetched instructions can be considered to be in the same equivalence class, and so can be replaced with the same fresh term variable; and Sect. 10 shows an approach to avoid the abstraction function along the specification side of the commutative correctness diagram for liveness. Sect. 11 presents experimental results, and Sect. 12 concludes the paper.

2 Definition of Safety and Liveness

The formal verification is done by correspondence checking—comparison of a pipelined implementation against a non-pipelined specification. The abstraction function, *Abs*, maps an implementation state to an equivalent specification state, and is computed by *flushing* [7]—feeding the implementation pipeline with bubbles (combinations of control signals that do not modify architectural state) until all partially executed instructions are completed. The safety property (see Fig. 1) is expressed as a formula in the logic of EUFM, and checks that one step of the implementation corresponds to between 0 and k steps of the specification, where k is the issue width of the implementation. F_{Impl} is the transition function of the implementation, and F_{Spec} is the transition function of the specification. We will refer to the sequence of first applying *Abs* and then F_{Spec} as the *specification side* of the diagram in Fig. 1, and to the sequence of first applying F_{Impl} and then *Abs* as the *implementation side*.

The safety property is the inductive step of a proof by induction, since the initial implementation state, Q_{Impl}, is arbitrary. If the implementation is correct for all transi-

tions that can be made for one step from an arbitrary initial state, then the implementation will be correct for one step from the next implementation state, Q'_{Impl}, since that state will be a special case of an arbitrary state as used for the initial state, and so on for any number of steps. For some processors, e.g., where the control logic is optimized by using unreachable states as don't-care conditions, we may have to impose *invariant constraints* for the initial state in order to exclude unreachable states. Then, we need to prove that those constraints are satisfied in the implementation state after one step, Q'_{Impl}, so that the correctness will hold by induction for that state, and so on for all subsequent states. (See [1][2] for a discussion of correctness criteria.)



Safety property:

$equality_0 \vee equality_1 \vee \ldots \vee equality_k = true$

Fig. 1. The safety correctness property for an implementation processor with issue width k: one step of the implementation should correspond to between 0 and k steps of the specification, when the implementation starts from an arbitrary initial state Q_{Impl} that is possibly restricted by a set of invariant constraints.

To illustrate the safety property in Fig. 1, let the implementation and specification have three architectural state elements—Program Counter (PC), Register File, and Data Memory. Let PC^i_{Spec}, $RegFile^i_{\text{Spec}}$, and $DMem^i_{\text{Spec}}$ be the state of the PC, Register File, and Data Memory, respectively, in specification state Q^i_{Spec} (i = 0, ..., k) along the specification side of the diagram. Let PC^*_{Spec}, $RegFile^*_{\text{Spec}}$, and $DMem^*_{\text{Spec}}$ be the state of the PC, Register File, and Data Memory in specification state Q^*_{Spec}, reached after the implementation side of the diagram. Then, each disjunct $equality_i$ (i = 0, ..., k) is defined as:

$$equality_i \leftarrow pc_i \wedge rf_i \wedge dm_i,$$

where

$$pc_i \leftarrow (PC^i_{\text{Spec}} = PC^*_{\text{Spec}}),$$
$$rf_i \leftarrow (RegFile^i_{\text{Spec}} = RegFile^*_{\text{Spec}}),$$
$$dm_i \leftarrow (DMem^i_{\text{Spec}} = DMem^*_{\text{Spec}}).$$

That is, $equality_i$ is the conjunction of pair-wise equality comparisons for all architectural state elements, thus ensuring that they are updated in synchrony by the same

number of instructions. In processors with more architectural state elements, an equality comparison is conjuncted for each additional state element. Hence, for this implementation processor, the safety property is:

$$pc_0 \wedge rf_0 \wedge dm_0 \ \vee \ pc_1 \wedge rf_1 \wedge dm_1 \ \vee ... \vee \ pc_k \wedge rf_k \wedge dm_k \ = \ true.$$

We can prove liveness by a modified version of the safety correctness criterion—by symbolically simulating the implementation for a finite number of steps, *n*, and proving that:

$$equality_1 \ \vee \ equality_2 \ \vee \ \ldots \vee \ equality_{n \times k} \ = \ true \qquad (1)$$

where *k* is the issue width of the implementation. The formula proves that *n* steps of the implementation match between 1 and $n \times k$ steps of the specification, when the implementation starts from an arbitrary initial state that may be restricted by invariant constraints. Note that (1) *guarantees that the implementation has made at least one step*, while the safety correctness criterion allows the implementation to stay in its initial state when formula $equality_0$ (checking whether the implementation matches the initial state of the specification) is *true*. The correctness formula is generated automatically in the same way as the formula for safety, except that the implementation and the specification are symbolically simulated for many steps, and formula $equality_0$ is not included. As in the formula for safety, every formula $equality_i$ is the conjunction of equations, each comparing corresponding states of the same architectural state element. That is, formula (1) consists of top-level positive equations that are conjuncted and disjuncted but not negated, allowing us to exploit Positive Equality when proving liveness. The minimum number of steps, *n*, to symbolically simulate the implementation, can be determined experimentally, by trial and error, or identified by the user after analyzing the processor (see Sect. 6).

The contribution of this paper is a highly automatic method to prove bounded liveness of pipelined processors with multicycle functional units. The proposed method enables the liveness check for a 5-stage pipelined DLX processor [13] with a 2048-cycle ALU, while producing 4 orders of magnitude speedup for a pipelined DLX with a 32-cycle ALU compared to a previous method for indirect proof of bounded liveness [42] (see Sect. 5).

3 Related Work

Safety and liveness were first defined by Lamport [20]. Most of the previous research on formal verification of processors has addressed only safety, as also observed in [2]. The most popular theorem-proving approach for proving microprocessor liveness is to prove that for each pipeline stage that can get stalled, if the stalling condition is *true* then the instruction initially in that stage will stay there, and if the stalling condition is *false* then the instruction will advance to the next stage. It is additionally proved that if the stalling condition is *true*, then it will eventually become *false*, given the implementation of the control logic and fairness assumptions about arbiters. Liveness was proved in this way by Srivas and Miller [34], Hosabettu et al. [15], Jacobi and Kröning [16], Müller and Paul [28], Kröning and Paul [17], and Lahiri et al. [19]. Sawada [32] similarly proved that if an implementation is fed with bubbles, it will eventually get

flushed. However, note that a buggy processor, where the architectural state elements are always disabled, may pass the check that stall signals will eventually become *false*, and that the pipeline will eventually get flushed, as well as satisfy the safety correctness criterion (where formula $equality_0$ will be *true*), but will fail the liveness check done here. Using a different theorem-proving approach, Manolios [23] also accounted for liveness by proving that a given state can be reached from a flushed state after an appropriate number of steps. McMillan [27] used circular compositional reasoning to check the liveness of a reduced model of an out-of-order processor with ALU and move instructions. His method requires the manual definition of lemmas and case-splitting expressions; the manual reduction of the proof to one that involves two reservation stations and one register; and the manual introduction of fairness assumptions for the abstracted arbiter. The approaches in the above nine papers will require significant manual work to apply to the models that are automatically checked for both safety and liveness in the current paper. Aagaard et al. [1] formulated a liveness condition, but did not present results.

Henzinger et al. [14] also enriched the specification, using a different method than ours, but had to do that even to prove safety of a 3-stage pipeline with ALU and move instructions. Biere et al. [3][4] enriched a model with a witnessing mechanism that records whether a property has been satisfied, thus allowing them to model check liveness of a communication protocol as safety. Pnueli et al. [29] proved the liveness of mutual-exclusion algorithms by deriving an abstraction, and enriching it with conditions that allowed the efficient liveness check in a way that implies the liveness of the original model. A method for indirect proof of liveness for pipelined processors was presented in [42]—see Sect. 5. Another approach suitable for proving both safety and liveness of pipelined processors was proposed in [24], but was not applied to designs with multicycle functional units.

4 EUFM, Positive Equality, and Efficient Translation to CNF

The syntax of EUFM [7] includes *terms* and *formulas*. Terms are used to abstract word-level values of data, register identifiers, memory addresses, as well as the entire states of memory arrays. A term can be an Uninterpreted Function (UF) applied to a list of argument terms, a term variable, or an *ITE* operator selecting between two argument terms based on a controlling formula, such that *ITE*(*formula*, *term1*, *term2*) will evaluate to *term1* if *formula* = *true*, and to *term2* if *formula* = *false*. The syntax for terms can be extended to model memories by means of the functions *read* and *write* [7][39]. Formulas are used to model the control path of a microprocessor, and to express the correctness condition. A formula can be an Uninterpreted Predicate (UP) applied to a list of argument terms, a Boolean 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 combined by Boolean connectives. We will refer to both terms and formulas as *expressions*. UFs and UPs are used to abstract the implementation details of functional units by replacing them with "black boxes" that satisfy no particular properties other than that of *functional consistency*—that equal values of the inputs to the UF (UP) produce equal output values.

The efficiency from exploiting Positive Equality is due to the observation 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. A maximally diverse interpretation is one where the equality comparison of a term variable with itself evaluates to *true*; that of a p-term variable with a syntactically distinct term variable (a p-equation) evaluates to *false*; and that of a g-term variable with a syntactically distinct g-term variable (a g-equation) could evaluate to either *true* or *false*, and can be encoded with Boolean variables [10][30][41].

In the formal verification tool flow, we can apply an optimization [44] that produces Boolean formulas with many ITE-trees. An ITE-tree can be translated to CNF with a unified set of clauses [44], without intermediate variables for outputs of ITEs inside the tree. ITE-trees can be further merged with one or more levels of their AND/OR leaves that have fanout count of 1. We can also merge other gate groups [43][44]. Merging of ITE-trees and other gate groups results in fewer variables and clauses, i.e., reduced solution space, and so less Boolean Constraint Propagation (BCP) and fewer cache misses.

5 Indirect Proof of Liveness

This section summarizes one of the results from [42]. To avoid the validity checking of the monolithic liveness correctness formula (1), which becomes complex for designs with long pipelines and many features, we can prove liveness indirectly:

THEOREM 1. *If after n implementation steps, $equality_0$ = false under a maximally diverse interpretation of the p-terms, and the safety property is valid, then the liveness property is valid under any interpretation.*

Note that under an interpretation that is not a maximally diverse interpretation of the p-terms, the condition $equality_0$ may become *true*, e.g., in the presence of software loops, or if multiple instructions raise the same exception and so update the PC with the same exception-handler address. However, the liveness condition (1) will be still valid, since it can only get disjuncted with other formulas that result from equations between syntactically distinct p-terms that become equal under an interpretation that is not a maximally diverse interpretation of the p-terms.

Since $equality_0$ is the conjunction of the pair-wise equality comparisons for all architectural state elements, it suffices to prove that one of those equality comparisons is *false* under a maximally diverse interpretation of the p-terms. In particular, we can prove that pc_0 = *false*, where pc_0 is the equality comparison between the PC state after the implementation side of the diagram (see Fig. 1), and the PC that is part of the initial specification state. Note that choosing the Register File or the Data Memory instead would not work, since they are not updated by each instruction, and so there can be infinitely long instruction sequences that do not modify these state elements. Note that proving *forward progress—that the PC is updated at least once after n implementation steps*, i.e., proving pc_0 = *false* under a maximally diverse interpretation of the p-terms—is done without the specification. However, the specification is used to prove safety, thus inductively the correctness for any number of steps.

6 Processor Benchmarks and Their Liveness

Experiments will be conducted with variants of a 5-stage pipelined DLX processor [13] that can execute ALU, branch, load, and store instructions. The 5 pipeline stages are: Fetch, Decode, Execute, Memory, and Write-Back. The Execute stage contains an ALU that can take either a single cycle or up to m cycles to compute the result of an ALU instruction. The actual latency may depend on the instruction opcode, the values of the data operands, etc., and so the choice between 1 and m cycles is made non-deterministically [37] in order to account for any actual implementation of the ALU. The processor benchmarks are named DLX-ALU4, DLX-ALU8, ..., and DLX-ALU2048, for values of m equal to 4, 8, ..., and 2048, respectively. The branch instructions have both their target address and their direction (indicating whether the branch is taken or not taken) computed in the Execute stage in a single cycle. ALU results, data memory addresses, branch targets, and branch directions depend on the instruction opcode, and two data operands read from the Register File (in the Decode stage) at locations specified by source register identifiers. ALU and load instructions also have a destination register identifier, indicating the Register File location where the result will be stored. Data hazards are avoided by forwarding logic in the Execute stage. While the ALU is computing the result of a multicycle operation, the instructions in previous stages are stalled, and bubbles are inserted in the Memory stage.

To illustrate the choice of number of steps, n, for the liveness proof of one of the above benchmarks where the ALU has a maximum latency of m cycles, note that the longest delay before such a processor fetches a new instruction that is guaranteed to be completed is $m + 3$ cycles. This will happen if the Decode stage contains a branch that will be taken, but the Execute stage contains an ALU instruction that will take m cycles. Then, the branch will be stalled for $m - 1$ cycles, followed by one cycle to go through Decode, another cycle to go through Execute (where the branch target and direction will be computed), a third cycle to go through Memory (where the PC will be updated with the branch target, and all subsequent instructions that are in previous pipeline stages will be cancelled), and a fourth cycle to fetch a new instruction that is guaranteed to be completed since the pipeline will be empty by then. Thus, a correct version of these processors has to be simulated symbolically for $m + 3$ steps in order to fetch a new instruction that is guaranteed to be completed.

7 Placeholder for Abstracting Multicycle Functional Units for Proving Safety and Liveness

Multicycle functional units are abstracted with a placeholder that is suitable for proving both safety and liveness (see Fig. 2), a modified version of a placeholder suitable for proving only safety [37]. Uninterpreted function ALU abstracts the functionality of the replaced multicycle functional unit.

In Fig. 2, when signal Flush is *false* (during regular symbolic simulation) and a multicycle instruction is in the pipeline stage of the abstracted functional unit, as indicated by signal Take_m being *true* in that stage, then the chain of $m - 1$ latches will be used to delay the multicycle computation for m cycles before the result of that computation is allowed to continue to the next stage. When signal Flush becomes *true* during the

computation of the abstraction function by flushing, then the chain of $m-1$ latches will be cleared on the next clock cycle; signal Complete will become *true* for 1 clock cycle as long as the placeholder contains a valid instruction in flight, thus completing that instruction; and signal Stall will be *false*, thus allowing the instructions in the previous pipeline stages to advance. Hence, this placeholder of a multicycle functional unit can be used for proving both safety (by setting Flush to *false* for one cycle of regular symbolic simulation, and then setting Flush to *true* in order to quickly complete partially executed instructions during flushing) and liveness (by setting Flush to *false* for as many cycles as required, and then setting Flush to *true* in order to quickly complete partially executed instructions during flushing). Multicycle memories are abstracted similarly, by using a memory model instead of the uninterpreted function ALU.

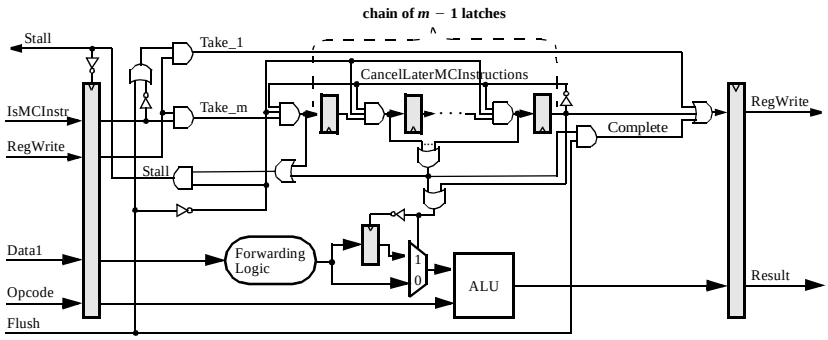


Fig. 2. Abstracting a multicycle ALU with a placeholder suitable for proving both safety and liveness of the pipelined processor. The latency is 1 cycle when signal Take_1 is *true* (i.e., RegWrite is *true* and IsMCInstr is *false*) or m cycles when signal Take_m is *true* (i.e., RegWrite is *true* and IsMCInstr is *true*). The chain of $m-1$ latches delays signal IsMCInstr for m cycles in the stage of the functional unit. The previous pipeline stages are stalled by signal Stall when the functional unit takes more than 1 cycle for an operation. Signal CancelLaterMCInstructions avoids the need to impose and check an invariant that at most one latch in the chain has value *true*.

An alternative implementation of the placeholder is without the feedback loop of signal CancelLaterMCInstructions that clears the $m-1$ latches when a multicycle operation completes. Instead, constraints are imposed that at most one of the $m-1$ latches contains a valid instruction. These constraints have to be checked for invariance after 1 cycle of regular symbolic simulation.

8 Abstracting the Branch Targets and Directions

The indirect proof of liveness (see Sect. 5) checks that the PC is modified by at least one new instruction after n implementation steps. However, for this proof it does not matter what the actual values of the branch targets and branch directions are when the PC is updated. The safety proof already guarantees that those values will be correct. Thus, we can abstract each instruction's branch target and direction with an oracle term and an oracle formula, respectively, such that there is a 1-to-1 relation between the instruction and its oracles. This can be done by either extending the Instruction Memory to produce the oracles, or introducing a new uninterpreted function and a new

uninterpreted predicate, respectively, that depend on the PC of the instruction.

The oracles for the branch target and direction are propagated along the processor pipeline in the same way as the instruction's opcode. These oracles are used in the Execute stage only when proving liveness, by being connected (e.g., by means of a multiplexor controlled by a signal indicating whether the proof is for liveness) to the signals for the branch target and branch direction, respectively, which are otherwise computed by an uninterpreted function and an uninterpreted predicate, respectively, when proving safety. The introduction of auxiliary state in a processor can be viewed as *design for formal verification*. Note that the oracle branch target and branch direction can be used as abstractions for the final actually chosen branch target and branch direction, respectively, in a design where several branch targets and branch directions are prioritized for each instruction. This abstraction technique, using oracles to abstract result terms and formulas, is general and applicable to other functional units, once it is proven that their operands are provided correctly for each instruction.

Since the above abstractions make the branch targets and directions independent of the data operands, we can perform *automatically* a cone-of-influence reduction to simplify the processor for its liveness proof by removing any circuitry that does not affect the PC—the only architectural state element in the EUFM formula for the indirect proof of liveness. That allows us to remove the uninterpreted function and uninterpreted predicate abstracting the functionality of, respectively, the functional unit computing the branch target and that computing the branch direction in the Execute stage; the forwarding logic for those functional units; the uninterpreted function abstracting the functionality of the multicycle ALU in the Execute stage, since the produced result no longer affects the updating of the PC; the forwarding logic for the multicycle ALU; the Register File and the Data Memory, since the data operands that they produce no longer affect the updating of the PC; and all connections for transferring of data operands. What is left after an automatic cone-of-influence reduction is a *timing abstraction* of the pipelined processor with multicycle functional units. The timing abstraction does not depend on the data operands in the original implementation, but only on signals that affect the stalling and squashing of the oracle branch targets and oracle branch directions in the reduced pipelined design.

LEMMA 1. If the timing abstraction of a pipelined processor model satisfies the condition $pc_0 = false$ under a maximally diverse interpretation of the p-terms after n steps, then that condition is also satisfied by the pipelined processor model itself.

Sketch of the proof: Because of the way that the oracles are generated and then propagated along the processor pipeline together with the instruction opcode, there is a 1-to-1 correspondence between each instruction and its oracles. Also, since the oracles are not constrained in any way—i.e., the oracles for the branch targets and branch directions are arbitrary terms and formulas, respectively—then each such oracle can be viewed as a "placeholder" for the actual value of a branch target or a branch direction, respectively. The safety proof already guarantees that any actual branch target and branch direction will be computed correctly. Furthermore, since no control decisions are made based on the values of the oracles for the branch targets, then those oracle terms will be classified as p-terms, allowing us to exploit Theorem 1. (In the case of

pipelined processors with branch prediction, where the actual branch targets are compared for equality with predicted branch targets in order to correct a branch misprediction, we can use special abstractions that turn the actual and predicted branch targets into p-terms [42].) Hence, a liveness proof based on arbitrary terms for the branch targets and arbitrary formulas for the branch directions will account for any outcome of the branches, and will ensure that the PC will be updated by at least one new instruction after any sequence of instructions executed during n implementation steps. Then, there will be no execution scenario of stalling or cancelling of instructions over n implementation steps, resulting in violation of the indirect liveness condition, pc_0 = *false*, under a maximally diverse interpretation of the p-terms. □

9 Abstracting the Branch Targets of New Instructions with a Dedicated Term Variable

Recall that we want to prove that formula pc_0 is *false* under a maximally diverse interpretation of the p-terms, which implies that $equality_0$ = *false*, and thus from Theorem 1 that the liveness condition holds (see Sect. 5). That is, we want to prove that the representation of pc_0 as $(PC^0_{Spec} = PC^*_{Spec})$ is *false* under a maximally diverse interpretation of the p-terms, where PC^0_{Spec} is the PC term after flushing the implementation along the specification side of the diagram in Fig. 1, and PC^*_{Spec} is the PC term after n regular implementation steps followed by flushing along the implementation side of the diagram.

After abstracting the branch targets with arbitrary terms, the term for PC^0_{Spec} will be a nested-*ITE* expression that has as leaves the term variables for branch targets of instructions that are initially in the pipeline. The term for PC^*_{Spec} will too be a nested-*ITE* expression that also has as leaves all of those term variables, as well as the terms for the branch targets of new instructions fetched during the n regular implementation steps. Because of modeling restrictions [35][36], all branch targets will appear as p-terms. Thus, in formula pc_0 the branch target p-terms of new instructions will be compared for equality with only branch target p-terms of instructions that are initially in the pipeline. Since the branch target p-terms of new instructions are syntactically distinct from the branch target p-terms that are initially in the pipeline, then such low-level equations will simplify to *false* when evaluating pc_0 under a maximally diverse interpretation of the p-terms. The only low-level equations in pc_0 that will evaluate to *true* are those where both arguments are the same p-term variable that is initially in the pipeline. Hence, the value of pc_0 under a maximally diverse interpretation of the p-terms will be preserved if all branch target p-terms of new instructions are considered to be in the same equivalence class, representing branch target p-terms that are syntactically distinct from those that are initially in the pipeline. *This observation allows us to abstract the branch targets of newly fetched instructions with the same dedicated fresh term variable*. By reducing the number of distinct p-term variables that are leaves of the nested-*ITE* arguments of equation pc_0, we will improve the efficiency of evaluating pc_0 under a maximally diverse interpretation of the p-terms.

Note that along the implementation side of the diagram, the PC is also updated with SequentialPC terms produced by an uninterpreted function that maps the current PC

term to a term for the sequential instruction address. Hence, applications of that uninterpreted function will also appear as leaves of term PC^{*}_{Spec}. Applying the above reasoning, we can replace all applications of that uninterpreted function with the dedicated fresh term variable used to abstract the branch targets of newly fetched instructions, since the PC is not updated with its sequential values during flushing along the specification side of the diagram. However, this will result in updating the PC with the dedicated fresh term variable on many clock cycles, and then in fetching the same symbolic instruction from the Instruction Memory on the next cycles. In order to prove liveness for an arbitrary instruction sequence executed over *n* implementation steps, we can abstract the Instruction Memory with a generator of arbitrary values [37], thus producing a completely arbitrary symbolic instruction and associated oracles on every clock cycle. As before, we will prove that there is no execution scenario that will prevent the fetching and completion of at least one new instruction.

10 Avoiding the Abstraction Function Along the Specification Side of the Diagram

Instead of checking that ($PC^{0}_{Spec} = PC^{*}_{Spec}$) is *false* under a maximally diverse interpretation of the p-terms, we can check that (*new_PC_var* = PC^{*}_{Spec}) is *true* under a maximally diverse interpretation of the p-terms, thus proving that the PC is overwritten with the dedicated fresh term variable *new_PC_var* after all execution sequences of length *n*. If that holds, then PC^{*}_{Spec} evaluates to *new_PC_var* under a maximally diverse interpretation of the p-terms, so that ($PC^{0}_{Spec} = PC^{*}_{Spec}$) is equivalent to ($PC^{0}_{Spec}$ = *new_PC_var*), which will be *false* under a maximally diverse interpretation of the p-terms, since *new_PC_var* is not a leaf of PC^{0}_{Spec} because by the definition of flushing *new_PC_var* is not written to the PC along the specification side of the diagram for liveness. Thus, we avoid the specification side of the diagram, since PC^{0}_{Spec} is no longer needed.

Additionally, we can automatically introduce an auxiliary circuit that when simulated symbolically will construct a formula that is equivalent to the formula (*new_PC_var* = PC^{*}_{Spec}) but is much simpler to evaluate. Intuitively, we can push the equation (*new_PC_var* = PC^{*}_{Spec}) to the leaves of PC^{*}_{Spec}, where PC^{*}_{Spec} is a nested-*ITE* expression with leaves that are term variables representing branch targets, and *ITE*-controlling formulas that are the enabling conditions for the updates of the PC along the implementation side of the diagram for liveness. Then, we can introduce an auxiliary circuit where a new latch is used to track whether each new update of the PC has value that is equal to *new_PC_var*, such that the latch is initialized with *false*, since the initial PC value is syntactically different from *new_PC_var*. This latch is updated under the same conditions that control the PC updates, but with the formula (*new_PC_var* = *new_PC_term*), where *new_PC_term* is the new term that is written to the PC in that clock cycle. Furthermore, we can apply automatically a retiming transformation [25][26], and move the equation (*new_PC_var* = *new_PC_term*) across pipeline latches that provide versions of *new_PC_term* in different clock cycles. The effect is to replace the term-level signal for a version of *new_PC_term* in each pipeline latch with a bit-level signal, indicating whether the initial version of *new_PC_term* in

that pipeline latch is syntactically equal to *new_PC_var*. This transformation replaces the term-level signal for branch targets with a bit-level signal, having initial values *false* in all pipeline latches (since the term variables representing the initial state of branch targets in pipeline latches are syntactically distinct from *new_PC_var*), while the value of this signal in the first pipeline stage is *true* (since the original term-level signal there is exactly *new_PC_var* that is fed both to the PC instead of the SequentialPC and to the first pipeline latch). This transformation is applied entirely automatically. Thus, we obtain a modified circuit, where a new latch records whether the PC has been updated with *new_PC_var*, such that the new latch is controlled by the enable signal for the original PC (that is no longer needed), but is updated with formulas. The formula built in the new latch represents directly the result from evaluating ($new\_PC\_var = PC^*_{Spec}$) under a maximally diverse interpretation of the p-terms, and thus avoids the increase in memory and CPU time necessary for an EUFM decision procedure to evaluate ($new\_PC\_var = PC^*_{Spec}$). Damm et al. [9] also reduced the domain to {0, 1} when formally verifying pipelines with a certain structure.

11 Results

The processor benchmarks from Sect. 6 were first checked for safety—each benchmark required less than 0.2 seconds—and then for liveness—see Tables 1 – 5. The term-level symbolic simulator TLSim [46] was used to symbolically simulate all models. The resulting EUFM correctness formulas were translated to equivalent propositional formulas by the decision procedure EVC [46] that then applied efficient translations to CNF [43][44][45]. Equations between g-term variables were encoded with the e_{ij} encoding [10]. The SAT-solvers siege_v4 [31] and BerkMin621 [11][12]—two of the top performers in the SAT'03 competition [21]—were used for all experiments; siege_v4 was faster on all of the resulting CNF formulas, but could not process a formula with more than 2^{19} CNF variables (see Table 2)—that formula was solved with BerkMin621. The computer was a Dell OptiPlex GX260 with a 3.06-GHz Intel Pentium 4, having a 512-KB on-chip L2-cache, 2 GB of memory, and running Red Hat Linux 9.0.

From Table 1, the previous method for indirect proof of liveness [42] (see Sect. 5) scaled up to the model with a 32-cycle ALU, DLX-ALU32, for which the proof took 2,483 seconds.

Table 2 shows the results after abstracting the branch targets and branch directions with oracles, thus making the branch targets and directions independent from the data operands, and then performing automatically a cone-of-influence reduction to eliminate all logic associated with the computation, transfer, and storage of operands (see Sect. 8). The time for the automatic cone-of-influence reduction was less than 0.1 second for each benchmark and is included in the time for symbolic simulation with TLSim. This approach produced *3 orders of magnitude speedup* for DLX-ALU32, reducing the total time for the liveness check from 2,483 seconds to 1.6 second (1,552× speedup). For this benchmark, the CNF variables were reduced almost 3×, the clauses more than 10×, and the literals almost 50×. The approach enabled scaling up to the model with a 128-cycle ALU, for which the proof took 258 seconds.

Table 3 presents the results after using a dedicated fresh term variable for all branch targets of newly fetched instructions, and abstracting the Instruction Memory with a generator of arbitrary values. This approach produced an order of magnitude speedup of the liveness check for the model with a 64-cycle ALU, DLX-ALU64, reducing the total time from 50 seconds to 2.75 seconds. The speedup was more than 8× for the model with a 128-cycle ALU, DLX-ALU128, for which the total time was reduced from 258 seconds to 30 seconds. Most importantly, this approach enabled scaling up to the model with a 256-cycle ALU, for which the proof took 393 seconds.

Table 1. Results from the previous method for indirect proof of liveness by proving $pc_0 = false$ under a maximally diverse interpretation of the p-terms [42].

Processor	CNF			Formal Verification Time [sec]			
	Variables	Clauses	Literals	TLSim	EVC	SAT	Total
DLX-ALU4	3,249	32,239	142,749	0.02	0.49	0.05	0.56
DLX-ALU8	7,905	102,699	555,272	0.03	5.72	0.22	5.97
DLX-ALU16	18,597	381,414	2,937,085	0.03	42.53	1.36	44
DLX-ALU32	63,285	2,137,614	26,113,861	0.06	2,462.94	20.20	2,483

Table 2. Results from indirect proof of liveness after also abstracting the branch targets and branch directions with oracles, and performing a cone-of-influence reduction.

Processor	CNF			Formal Verification Time [sec]			
	Variables	Clauses	Literals	TLSim	EVC	SAT	Total
DLX-ALU32	23,735	171,974	551,002	0.04	1.24	0.32	1.60
DLX-ALU64	165,159	1,195,077	3,841,960	0.08	12.31	37.59	50
DLX-ALU128	784,587	6,198,558	20,833,828	0.19	100.68	157.13[a]	258
DLX-ALU256	——	——	——	0.39	> mem.	——	——

a. BerkMin621 was used, since siege_v4 cannot process CNFs with more than 2^{19} variables.

Table 3. Results from indirect proof of liveness after also using a dedicated fresh term variable for all branch targets of newly fetched instructions, and abstracting the Instruction Memory with a generator of arbitrary values.

Processor	CNF			Formal Verification Time [sec]			
	Variables	Clauses	Literals	TLSim	EVC	SAT	Total
DLX-ALU32	1,827	34,979	105,580	0.04	0.29	0.09	0.42
DLX-ALU64	5,635	220,243	663,052	0.08	1.79	0.88	2.75
DLX-ALU128	19,395	1,566,643	4,708,684	0.18	18.26	11.11	30
DLX-ALU256	71,491	11,832,947	35,532,748	0.46	234.36	158.13	393

Table 4. Results from indirect proof of liveness after also abstracting the multicycle ALU with a placeholder without feedback loops.

Processor	CNF			Formal Verification Time [sec]			
	Variables	Clauses	Literals	TLSim	EVC	SAT	Total
DLX-ALU32	1,991	23,857	72,519	0.05	0.26	0.11	0.42
DLX-ALU64	5,959	130,369	392,887	0.11	1.08	0.61	1.80
DLX-ALU128	20,039	854,369	2,566,551	0.33	8.60	2.60	12
DLX-ALU256	72,775	6,181,281	18,550,615	1.78	92.30	36.24	130

Table 5. Results from indirect proof of liveness after also implementing the equation with the dedicated fresh term variable for the new PC values as an auxiliary circuit and avoiding the specification side of the diagram.

Processor	CNF			Formal Verification Time [sec]			
	Variables	Clauses	Literals	TLSim	EVC	SAT	Total
DLX-ALU32	837	6,562	20,768	0.05	0.09	0.08	0.22
DLX-ALU64	1,783	21,278	63,129	0.11	0.22	0.35	0.68
DLX-ALU128	3,422	71,234	226,927	0.32	0.78	0.89	1.99
DLX-ALU256	6,995	259,793	914,040	1.77	3.46	13.95	19
DLX-ALU512	13,907	978,001	3,465,464	8.24	19.79	43.22	71
DLX-ALU1024	27,731	3,790,673	13,483,512	71	252	210	533
DLX-ALU2048	51,276	16,992,534	55,198,573	899	4,117	1,042	6,058

Table 4 presents the results after abstracting the multicycle ALU with a placeholder without feedback loops, and using a constraint to restrict the initial state of that placeholder so that it contains at most one valid instruction in the chain of $m - 1$ latches. Checking the invariance of this constraint took less than 1 second for each of the benchmarks. This approach resulted in 3× speedup of the liveness check for the model with a 256-cycle ALU, DLX-ALU256, reducing the total time from 393 seconds to 130 seconds. Furthermore, while the CNF variables increased only slightly, the CNF clauses and literals were almost halved for DLX-ALU128 and DLX-ALU256.

Table 5 presents the results after implementing the equation with the dedicated fresh term variable for the new PC values as an auxiliary circuit and avoiding the specification side of the diagram (see Sect. 10). The auxiliary circuit was introduced automatically—that required less than 0.2 seconds for each benchmark and the exact time is included in the time for symbolic simulation with TLSim. This approach resulted in a 6.5× speedup for the model with a 256-cycle ALU, DLX-ALU256, reducing the total time from 130 seconds to 19 seconds, while the CNF variables, clauses and literals were reduced by an order of magnitude. Most importantly, this approach enabled the scaling for the model with a 2048-cycle ALU, DLX-ALU2048, for which the liveness check took 6,058 seconds. *Note that the speedup for DLX-ALU32 is 4 orders of magnitude relative to the previous method for indirect proof of liveness* (see Table 1).

12 Conclusions

Presented was an approach for proving liveness of pipelined processors with multicycle functional units, without the need for the user to set up an inductive argument. The method scaled for a 5-stage pipelined DLX with a 2048-cycle ALU, and resulted in 4 orders of magnitude speedup for a design with a 32-cycle ALU, compared to a previous method for indirect proof of liveness [42]. Given that functional units in recent state-of-the-art processors usually have latencies of up to $20-30$ cycles, and rarely up to 200 cycles, the presented approach will enable the automatic formal verification of liveness for realistic pipelined processors targeted to embedded and DSP applications. Future work will improve the scaling of the new approach.

References

[1] M.D. Aagaard, N.A. Day, and M. Lou, "Relating multi-step and single-step microprocessor correctness statements," *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, Springer-Verlag, November 2002.

[2] M.D. Aagaard, B. Cook, N.A. Day, and R.B. Jones, "A framework for superscalar microprocessor correctness statements," Software Tools for Technology Transfer (STTT), Vol. 4, No. 3 (May 2003).

[3] A. Biere, C. Artho, and V. Schuppan, "Liveness checking as safety checking," Electronic Notes in Theoretical Computer Science 66, 2002.

[4] V. Schuppan, and A. Biere, "Efficient reduction of finite state model checking to reachability analysis," International Journal on Software Tools for Technology Transfer (STTT), Vol. 5, No. 2–3, Springer-Verlag, March 2004.

[5] R.E. Bryant, S. German, and M.N. Velev, "Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic," ACM Transactions on Computational Logic (TOCL), Vol. 2, No. 1 (2001).

[6] R.E. Bryant, and M.N. Velev, "Boolean satisfiability with transitivity constraints," ACM Transactions on Computational Logic (TOCL), Vol. 3, No. 4 (October 2002), pp. 604–627.

[7] J.R. Burch, and D.L. Dill, "Automated verification of pipelined microprocessor control," *CAV '94*, June 1994.

[8] J.R. Burch, "Techniques for verifying superscalar microprocessors," *33rd Design Automation Conference (DAC '96)*, June 1996.

[9] W. Damm, A. Pnueli, and S. Ruah, "Herbrand Automata for Hardware Verification," *9th International Conference on Concurrency Theorey (CONCUR '88)*, D. Sangiorgi and R. de Simone, *eds.*, Springer-Verlag, LNCS 1466, 1988.

[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)*, LNCS 1427, Springer-Verlag, June 1998, pp. 244–255.

[11] E. Goldberg, and Y. Novikov, "BerkMin: A fast and robust sat-solver," *DATE '02*, March 2002, pp. 142–149.

[12] E. Goldberg, and Y. Novikov, SAT-solver BerkMin621, June 2003.

[13] J.L. Hennessy, and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, San Francisco, 2002.

[14] T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "Decomposing refinement proofs using assume-guarantee reasoning," *International Conference on Computer-Aided Design (ICCAD '00)*, 2000.

[15] 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)*, LNCS 1633, Springer-Verlag, 1999.

[16] C. Jacobi, and D. Kröning, "Proving the correctness of a complete microprocessor," *30. Jahrestagung der Gesellshaft für Informatik*, Springer-Verlag, 2000.

[17] D. Kröning, and W.J. Paul, "Automated pipeline design," *Design Automation Conference (DAC '01)*, June 2001.

[18] S. Lahiri, C. Pixley, and K. Albin, "Experience with term level modeling and verification of the M•CORE™ microprocessor core," *International Workshop on High Level Design, Validation and Test (HLDVT '01)*, November 2001.

[19] S.K. Lahiri, S.A. Seshia, and R.E. Bryant, "Modeling and verification of out-of-order microprocessors in UCLID," *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, Springer-Verlag, November 2002.

[20] L. Lamport, "Proving the correctness of multiprocess programs," IEEE Trans. on Software Engineering," Vol. 3, No. 2 (1977).

[21] D. Le Berre, and L. Simon, "Results from the SAT'03 solver competition," *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03)*, 2003. http://www.lri.fr/~simon/contest03/results/

[22] S. Malik, A.R. Wang, R.K. Brayton, and A. Sangiovani-Vincentelli, "Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment," *International Conference on Computer-Aided Design*, November 1988.

[23] P. Manolios, "Mechanical verification of reactive systems," Ph.D. Thesis, Computer Sciences, Univ. of Texas at Austin, 2001.

[24] P. Manolios, and S.K. Srinivasan, "Automatic verification of safety and liveness for XScale-like processor models using WEB refinements," *Design, Automation and Test in Europe (DATE '04)*, Vol. 1, February 2004.

[25] J. Matthews, and J. Launchbury, "Elementary microarchitecture algebra," *Computer-Aided Verification (CAV '99)*, N. Halbwachs, and D. Peled, *eds.*, LNCS 1633, Springer-Verlag, July 1999, pp. 288–300.

[26] J.R. Matthews, "Algebraic specification and verification of processor microarchitectures," Ph.D. Thesis, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, October 2000.

[27] K.L. McMillan, "Circular compositional reasoning about liveness," Technical Report, Cadence Berkeley Labs, 1999.

[28] S.M. Müller, and W.J. Paul, *Computer Architecture: Complexity and Correctness*, Springer-Verlag, 2000.

[29] A. Pnueli, J. Xu, and L. Zuck, "Liveness with (0, 1, infinity)-counter abstraction," *CAV '02*, LNCS 2404, Springer-Verlag, July 2002.

[30] A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel, "The small model property: how small can it be?", Journal of Information and Computation, Vol. 178, No. 1 (October 2002), pp. 279–293.

[31] L. Ryan, Siege SAT Solver v.4. http://www.cs.sfu.ca/~loryan/personal/

[32] J. Sawada, "Verification of a simple pipelined machine model," in *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers, Boston/Dordrecht/London, 2000.

[33] H. Sharangpani, and K. Arora, "Itanium processor microarchitecture," IEEE Micro, Vol. 20, No. 5 (2000).

[34] M.K. Srivas, and S.P. Miller, "Formal verification of an avionics microprocessor," Tech. Report CSL-95-4, SRI International, 1995.

[35] M.N. Velev, and R.E. Bryant, "Exploiting positive equality and partial non-consistency in the formal verification of pipelined microprocessors," *36th Design Automation Conference (DAC '99)*, June 1999.

[36] M.N. Velev, and R.E. Bryant, "Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic," *CHARME '99*, LNCS 1703, September 1999, pp. 37–53.

[37] M.N. Velev, and R.E. Bryant, "Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction," *37th Design Automation Conference (DAC '00)*, June 2000.

[38] M.N. Velev, "Formal verification of VLIW microprocessors with speculative execution," *Computer-Aided Verification (CAV '00)*, E.A. Emerson, and A.P. Sistla, *eds.*, LNCS 1855, Springer-Verlag, July 2000.

[39] M.N. Velev, "Automatic abstraction of memories in the formal verification of superscalar microprocessors," *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, LNCS 2031, April 2001, pp. 252–267.

[40] M.N. Velev, and R.E. Bryant, "Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors," Journal of Symbolic Computation (JSC), Vol. 35, No. 2 (February 2003).

[41] M.N. Velev, "Automatic abstraction of equations in a logic of equality," *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX '03)*, LNAI 2796, Springer-Verlag, September 2003.

[42] M.N. Velev, "Using positive equality to prove liveness for pipelined microprocessors," *Asia and South Pacific Design Automation Conference (ASP-DAC '04)*, January 2004.

[43] M.N. Velev, "Efficient translation of Boolean formulas to CNF in formal verification of microprocessors," *Asia and South Pacific Design Automation Conference (ASP-DAC '04)*, January 2004.

[44] M.N. Velev, "Exploiting signal unobservability for efficient translation to CNF in formal verification of microprocessors," *Design, Automation and Test in Europe (DATE '04)*, February 2004.

[45] M.N. Velev, Comparative Study of Strategies for Formal Verification of High-Level Processors, 22nd International Conference on Computer Design (ICCD '04), October 2004, pp. 119–124.

[46] M.N. Velev, and R.E. Bryant, "TLSim and EVC: A term-level symbolic simulator and an efficient decision procedure for the logic of equality with uninterpreted functions and memories," International Journal of Embedded Systems (IJES), Special Issue on Hardware-Software Codesign for Systems-on-Chip, 2004.