EPEX: Processor Verification by Equivalent Program Execution

Lucas Klemmer
Institute for Complex Systems
Johannes Kepler University
Linz, Austria
lucas.klemmer@jku.at

Daniel Große
Institute for Complex Systems
Johannes Kepler University
Linz, Austria
daniel.grosse@jku.at

ABSTRACT

Verifying processors has been and still is a major challenge. Therefore, intensive research has led to advanced verification solutions ranging from ISS-based reference models, (cross-level) simulation down to formal verification at the RTL. During the verification of the processor implementation at the *Instruction Set Architecture* (ISA) level, test stimuli, i.e. test programs are needed. They are either created manually or with the aid of sophisticated test program generators. However, significant effort is required to produce thorough test programs.

In this paper, we devise a novel approach for processor verification by *Equivalent Program EXecution* (EPEX). Our approach is based on a new form of equivalence checking: Instead of comparing the architectural states of two models which execute the same program P, we derive a second, but equivalent program $\hat{P}$ from P (wrt. to a formal ISA model), and check that executing P and $\hat{P}$ will produce equal architectural states on the *same* design. We show that EPEX can easily be used in a simulation-based verification environment and broadens existing tests automatically. In a RISC-V case study using different core configurations of the well-known VexRiscv core, we demonstrate the bug-finding capabilities of EPEX.

CCS CONCEPTS

• **Hardware → Integrated circuits**; **Equivalence checking**; **Simulation and emulation**; • **Computer systems organization → Architectures**.

KEYWORDS

Processor Verification, Formal ISA model, RISC-V, Equivalence Checking, Functional Verification, Simulation

ACM Reference Format:
Lucas Klemmer and Daniel Große. 2021. EPEX: Processor Verification by Equivalent Program Execution. In *Proceedings of the Great Lakes Symposium on VLSI 2021 (GLSVLSI '21), June 22–25, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3453688.3461497

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GLSVLSI '21, June 22–25, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8393-6/21/06…$15.00
https://doi.org/10.1145/3453688.3461497

1 INTRODUCTION

With the advent of the RISC-V *Instruction-Set Architecture* (ISA) [29, 30] and the principles of open-source arriving in the semiconductor industry, teams started to design their own processor or accelerator as RISC-V is a free and open ISA. The RISC-V ISA standard is maintained by the non-profit RISC-V foundation. Moreover, configurability and extendability were important design factors of the RISC-V ISA. However, while this opens up a lot of opportunities, two major issues occur: (1) there is not much field-proven verification experience for RISC-V processors and (2) the flexibility of the RISC-V ISA wrt. ISA extensions results in significant additional verification effort. During processor design, effective verification of the *Register Transfer Level* (RTL) implementation at the ISA level is a mandatory task. Here, instructions are checked on the complete processor and this also includes initialization code/boot code, peripheral access etc. Hence, at the ISA level test programs (binaries) are needed which are simulated on the design. While they are still very often created manually, also methods for automatic generation of test programs have been proposed to allow for thorough verification. Prominent examples include approaches using constraint solving techniques, guided random generation, or approaches such as quick error detection. We provide more details in the discussion on related work. Overall, however, significant effort is required to produce thorough test programs.

In this paper, we devise a novel approach for processor verification by ***Equivalent Program EXecution*** (EPEX). Our approach is based on a new form of equivalence checking: Instead of comparing the architectural states of two models which execute the same program P, we *iteratively* perform two integrated steps for verification: (1) We replace a single instruction of P by one or more instructions; more formally, we *locally* determine for the next to be executed instruction P_i of P an equivalent instruction sequence $\hat{P_{ij}}$ of length j. For this task, we have created a *formal ISA model*[1] for which we constrain the current architectural state and next architectural state when executing P_i, and eventually make use of an *Satisfiability Modulo Theories* (SMT) solver. (2) we check that executing $\hat{P_{ij}}$ and P_i will produce equal architectural states on the *same* design.

This general procedure of EPEX provides improved processor verification which is further strengthened by exploiting two major degrees of freedom coming with the EPEX principle: (1) we can substitute the next instruction with exactly one other instruction, or we can insert longer linked instructions sequences which will increase the variety of instructions influencing the test result. (2) when searching for an instruction replacement we can guide the

[1] Our formal ISA model is available at https://github.com/ics-jku/epex-formal-rv32-model.

SMT-solver such that P and $\hat{P}$ will activate very different control and data paths of the processor implementation and hence enabling to compare typically unrelated logic/operations against each other.

In total, EPEX broadens a given test program automatically. EPEX increases the chance in detecting bugs as for example the result of a compute-instruction is now compared to the result of a jump which has to be identical per construction on the architectural states but obviously checks very different micro-architectural states.

We show that EPEX can easily be used in a simulation-based verification environment. In a RISC-V case study using different core configurations of the well-known VexRiscv core, we demonstrate the bug-finding capabilities of EPEX.

This paper is structured as follows: Section 2 discusses related work. Our EPEX approach is introduced in Section 3. The experimental evaluation is presented in Section 4. Finally, the paper is concluded in Section 5.

2 RELATED WORK

For processor verification various approaches aiming at the generation of test programs have been proposed. A strong focus was put on the separation of the test generator from the architecture description. This includes for instance approaches using constraint solving techniques [9, 12]. [24] presented an optimized test generation framework which propagates constraints among multiple instructions effectively. The test program generator of [13] uses a constraint-based coverage model describing execution paths of individual instructions. Alternative approaches integrate coverage-guided test generation via Bayesian networks [17] and other machine learning techniques [11, 23] as well as fuzzing [26]. However, these approaches are either not designed for RTL verification, require a lot of effort until thorough tests are generated, or do not target the RISC-V ISA.

Recently, several RISC-V specific verification approaches emerged. In [3, 5], the RISC-V foundation provides official hand-written test-suites. They are typically used as a starting point when implementing a RISC-V processor. A model-based test generation approach is the RISC-V *Torture Test* [6] framework. It uses Scala and generates tests based on randomized instruction sequence templates. [20] is another model-based approach that employs a constraint-based specification for test generation. For verification at the *Instruction Set Simulator* (ISS) level, approaches leveraging coverage-guided fuzzing have been proposed recently [19, 22]. Googles' *RISC-V DV* [7] is another test generation approach that leverages SystemVerilog in combination with the *Universal Verification Methodology* (UVM) to continuously generate RISC-V instruction streams based on constrained-random specifications. In [21] an efficient cross-level testing approach for processor verification has been proposed which generates an endless instruction stream on-the-fly during simulation. While all these approaches are very sophisticated and aim for closing the remaining coverage gap after decent verification progress, EPEX is complementary as existing tests can be broadened automatically and the verification check is extended across different data and control paths.

Besides methods for test generation, also formal approaches aiming at RISC-V have been developed. Notable are *riscv-formal* [4] and the *OneSpin 360 DV* RISC-V verification app [2] which are based on advanced model checking techniques. However, for using *riscv-formal* the user has to implement the RISC-V formal interface, and *OneSpin 360 DV* is only commercially available. Beyond this, initially proposed for post-silicon verification only, *Quick Error Detection* (QED) has been enhanced and recently a symbolic form has been proposed, i.e. *Symbolic Quick Error Detection* (SQED). SQED targets pre-silicon verification of processors [28] and has been extended and applied to RISC-V processors [16]. As such it checks whether the outputs produced by executing a particular instruction sequence match if the sequence is executed twice, assuming the inputs to the two sequences also match. For this check SQED leverages bounded model checking. While SQED offer several advantages as a formal technique, there are also some limitations: Single instruction checks have to be performed for each instruction, i.e. each instruction has to be completely formally verified on the core. In contrast, EPEX can uncover bugs even in a basic data path operation, as a comparison to other unrelated logic is possible. Moreover, SQED involves splitting the register file in half and using one half for the original instructions and the second half for a duplicated sequence of instructions. This however imposes practical restrictions, as for example tests cannot use the full register range. Overall, while formal approaches can provide correctness guarantees, they are significantly more difficult to use than simulation-based methods and, due to their complexity and potential scalability issues, should be complemented by simulation-based methods.

Also in the formal context solutions to formalize the RISC-V ISA semantics have been introduced, for example [8, 27]. They allow leveraging theorem proving to reason about the RISC-V ISA semantics and generate simulation back ends. Unfortunately, even if mentioned, no formal ISA model in propositional logic is generated or available which we would have integrated in EPEX. Therefore, we created our own model.

3 EPEX

In this section, we first describe the basic idea of our approach in Section 3.1. Then, we describe the EPEX flow in Section 3.2. Afterwards, we detail the core ingredients in the remaining sections.

3.1 Basic Idea

When verifying a processor implementation at the ISA level, test programs (binaries) are needed. As the ISA of a processor defines the interface between the hardware and the software, processing an instruction of a test program transforms the architectural state s, i.e. the state visible to the programmer, to a new architectural state s'. We leverage this programmers view abstraction at the ISA level as follows: We take an instruction of a test program P starting in s leading to the successor architectural state s'. Now we use a formal ISA model and automated reasoning (SMT) to determine a different instruction, which, when executed on s, results in the same successor architectural state s' as the original instruction. This process is repeated for every instruction of P. By this, we have transformed P to $\hat{P}$ according to the ISA specification and know that the behavior of both programs is equivalent on the architectural states. Hence, instead of classical equivalence checking where two models are compared on identical input, we now consider one processor implementation (which is instantiated twice), simulate the equivalent programs P and $\hat{P}$, respectively, and compare the

```
0996: ...
# assume t1 = 2, t2 = 2046, t3 = 0
1000: add t3, t1, t2 # t3 = t1 + t2 = 2048
1004: ...
```

(a) Excerpt of test program P

```
0996: ...
# assume t1 = 2, t2 = 2046, t3 = 0
1000: slli t3, t1, 10 # shift left t1 by 10
1004: ...
```

(b) Excerpt of equivalent test program $\hat{P}1$

```
0996: ...
# assume t1 = 2, t2 = 2046, t3 = 0
1000: j 1044 # set pc to 2044
1004: ...

2044: jal t3, -1040 # jump back and link t3 =
    2044 + 4 = 2048
```

(c) Excerpt of equivalent test program $\hat{P}2$

Figure 1: Example for broadening of a test program

architectural states along the execution. Consequently, we denote our novel approach for processor verification as *Equivalent Program EXecution* (EPEX).

Taking the idea further, EPEX can be extended naturally to search for instruction sequences instead of only a single instruction used as replacement to form $\hat{P}$. In general, EPEX offers the following advantages: (a) existing tests can be broadened automatically, (b) the search can be guided to activate different control and data paths when determining an instruction replacement, and (c) general purpose software can be converted into enhanced test programs.

We now provide two concrete examples to illustrate EPEX and in particular the broadening of test programs. First, we consider the replacement of the current instruction with a single instruction. Thereafter, we provide an example using two instructions for the equivalent program transformation.

Example 1. *Figure 1a shows an excerpt of the test program P together with the memory addresses where the instructions of P are located. Let us consider the* `add` *instruction to be processed in a given architectural state. This architectural state has been reached by executing all instructions up to and including address 0996. We only list the relevant subset of this architectural state, i.e. the registers $t1 = 2$, $t2 = 2046$ and $t3 = 0$. After the execution of* `add`*, the target register $t3$ has been set to* 2048*. In Figure 1b an equivalent program, as found by EPEX employing formal methods, is shown where the* `add` *instruction has been replaced by the shift left logical immediate. More precisely, the execution of* `slli` *shifts $t1$ by* 10 *and stores the result of* 2048 *into $t3$. By this, both architectural states, after executing up to and including* `add` *of P and up to and including* `slli` *of $\hat{P}1$, respectively, are identical. This example demonstrates that the compute-instruction* `add` *can be replaced by another compute-instruction (here* `slli`*) in this particular scenario/architectural state. By this, EPEX broadens an existing test program P in the sense that typically unrelated logic, i.e. in particular different control and data path operations are executed and their results are finally compared.*

We now look at a more complex example.

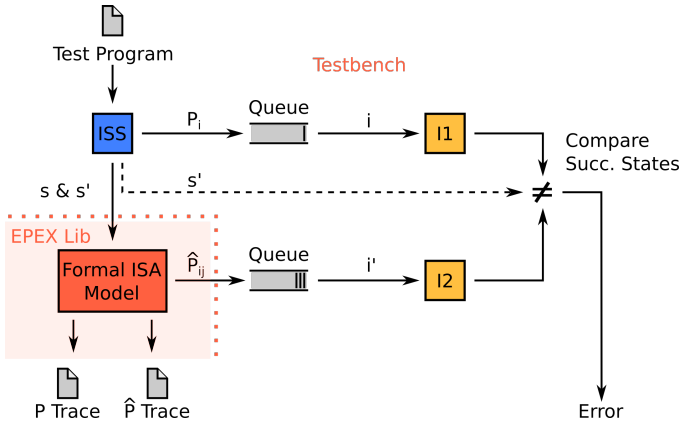

Figure 2: EPEX flow

Example 2. *Again, we consider the same test program P of Figure 1a. However, this time we search for an instruction replacement consisting of two instructions. The result using EPEX is shown in Figure 1c. The determined test program uses two jump instructions and a free memory location. In detail, from address 1000 a relative jump (*`j`*) to address 2044 is performed. For this location a jump-and-link instruction (*`jal`*) has been generated which performs a relative jump to 1004 (the next instruction after the original add) and sets the target register $t3$ as sum of the current program counter and 4, resulting in the needed value 2048. This new test program shows that an equivalent test program can be determined which now relates a compute-instruction with two instructions from a completely different instruction class, i.e. control flow operations. These kinds of test programs will neither be created manually, nor random generators are able to produce them.*

As both examples have demonstrated, EPEX opens up a novel approach for the verification of processors by broadening test programs. We introduce the EPEX flow in the next section.

3.2 EPEX Flow

The EPEX flow is depicted in Figure 2. EPEX is split into two components: (1) The (processor) testbench controls the iterative execution of the original test program and drives the simulation of the two processor instances, denoted as $I1$ and $I2$ in Figure 2, respectively. (2) The generation of equivalent instructions with the formal ISA model is performed via the EPEX library. This library is not tied to any particular processor implementation and can be easily connected to other cores.

EPEX starts with a user-supplied test program as input (top in Figure 2). From the formal ISA model we have also derived an ISS. Into this ISS, the test program is loaded by the testbench. Then, the ISS starts to execute the program step by step. For each step, the architectural states s and s', and the executed original instruction P_i are saved.

The formal ISA model is constrained such that the initial state of the model matches s and the final state matches s'. This constrained model is then given to the Z3 SMT-solver which finds assignments for the unconstrained instruction variables that lead from s to s'. The values of the instruction variables form an instruction sequence

$\hat{P_{ij}}$ which is equivalent to the original instruction P_i. The instruction sequence $\hat{P_{ij}}$ consists, depending on additional constraints (see Section 3.4), of either a single instruction or multiple linked instructions. Both, P_i and $\hat{P_{ij}}$ are pushed to instruction queues which are connected to the corresponding processor instances *I1* and *I2*, respectively. Also, P_i and $\hat{P_{ij}}$ are dumped into separate trace files by the EPEX library. This enables later testing via a cache mechanism without the cost of querying the formal model again.

The processor instances fetch their instructions not from memory but from the queues (gray in the center of Figure 2). This allows controlling the instructions that are fetched and executed by *I1* and *I2*. After *I1* has finished the execution of P_i and *I2* has finished the execution of $\hat{P_{ij}}$, the architectural states of *I1*, *I2* and the *ISS* are compared. If the architectural states do not match, a bug either in *I1* or *I2* has been discovered, and the EPEX test is terminated and marked as failed. Recall that the ISS has been derived from the formal ISA model and is therefore correct by construction. However, *I1* and *I2* (instances of the processor at hand) execute different, but on the architectural states equivalent, programs as generated by EPEX. Therefore in this case the mismatch has to result from the activation of different control and/or data paths of the processor via the programs P_i and $\hat{P_{ij}}$, respectively.

In the following we detail the core components, i.e. the formal ISA model and constraints for guidance of the search. Finally, the major implementation aspects are summarized.

3.3 Formal ISA Model

In this section, we introduce the formal ISA model which we have developed and which is used for the generation of equivalent programs.

Finding equivalent instruction sequences is a special case of the program synthesis problem [25] which is up to today a challenging task. Formal methods, such as SMT-solving, are commonly used in program synthesis, see for example [18].

While multiple projects of formal RISC-V ISA specifications emerged, nothing is available which is suitable for automated reasoning. To minimize the effort for the creation of the formal ISA model we decided to create it in C. This additionally allows to directly derive the ISS needed for EPEX. To integrate our formal model into EPEX, we use CBMC [14] to transform the C description into an SMT-lib v2.0 [10] instance. Hence, efficient reasoning becomes possible via the well known SMT-solver Z3 [15].

To save run-time in the EPEX generation phase the SMT-lib instance produced by CBMC is post-processed to include placeholders for the architectural states s and s', respectively. During an EPEX run, these placeholders are replaced by the actual values of s and s'. Depending on the number of instructions that are to be generated, the model is unrolled. Each unwinding generates a new free 32-bit variable that encodes the instruction to be executed. Moreover, additional constraints to guide the search in case of generating an instruction sequence longer than one are necessary which we discuss in the following section.

3.4 Guidance Constraints

If one instruction is replaced by an instruction sequence, the number of possible instruction sequences grows rapidly. Thus, some form of guidance is necessary since many possible sequences even contain "uninteresting" instructions (e.g. different de-facto NOP variations like *or x0, x0, x0*). To guide the formal search towards generating interesting instructions we propose the following two constraints: The *effect constraint* specifies that the value of the destination register $s'[rd]$ in the successor architectural state s' is different from the value of the same register in the predecessor state s and by this reduces the number of de-facto NOPs. Since not every instruction produces a result in a destination register (e.g. branches), the *effect constraint* is not always required. Overall we get formally:

Definition 1 (Effect Constraint).
$has\_result(current\_instr_p) \rightarrow s'[rd] \neq s[rd]$.

If instruction sequences are generated, the *chain constraint* guides the search to produce sequences with instructions that build on each other's results. This is reflected by relating the variables representing the registers along the generated instruction sequence P_{ij} with the index position p. Essentially, we formulate that consecutive instructions of the instructions sequence must use the result of the previous one, if one is produced:

Definition 2 (Chain Constraint).
$has\_result(current\_instr_p) \rightarrow (rs1_{p+1} = rd_p \vee rs2_{p+1} = rd_p)$

Besides this essential constraints, there are several options to drive the search in certain directions. We also did some experiments for this, but a thorough investigation is left for future work.

3.5 Implementation

As already shown in Figure 2, EPEX consists of the EPEX library (containing the formal ISA model) and the processor testbench composed of the ISS, the processor instances, and the component for comparison of the architectural states. This separation makes the EPEX approach easily portable. The testbench needs to provide certain functionality to support EPEX. Explicit control over the fetched instructions is necessary. Moreover, it is essential that the exact time point a processor commits an instruction can be obtained, an common requirement for many verification approaches (e.g. [4, 21]).

4 RESULTS

In this section, we present the experimental evaluation of our proposed EPEX approach. For the experiments we use a modular RISC-V implementation for FPGAs which is provided by the VexRiscv project [1]. The VexRiscv cores are built in SpinalHDL. SpinalHDL is a domain specific language in Scala which allows describing RTL by using object-oriented and functional programming. The VexRiscv cores are completely configurable from a bare-bone minimal CPU to a full-fledged Linux capable processor using a very flexible plugin concept. Finally, for a concrete core configuration Verilog is generated.

In the first part of the experiments (Section 4.1), we reconsider the basic examples from Section 3. Afterwards, we demonstrate the bug-finding capabilities of EPEX in a set of mutation-based experiments (Section 4.2). Finally, we show for multiple examples that executing P and $\hat{P}$ on the VexRiscv processor leads to a significant variation in the control paths.

4.1 Basic EPEX Examples

We come back to the test program excerpt described in Example 1. For this experiment we selected a minimal VexRiscv configuration, but the goal was to speed up shifting of the core. During the design and integration of a barrel shifter in the RISC-V core, the shifting was split in three operations: In case of a left shift operation, it first reverses the input. Then, it performs a right shift with variable number of steps. Finally, in case of a left shift operation, it reverses the output again. Additionally, a configuration option was integrated to define the location of the final reverse operation to be either in the execute stage (reduces the logic depth) or in the memory stage of the pipeline. This has been modeled very nicely in SpinalHDL, however, due to a copy and paste error the operation was not reversed correctly. Running our test program $\hat{P1}$ (Figure 1b), which has been generated by EPEX, easily revealed this bug. Please note, that it was immediately clear that the bug was not caused by the `add` instruction of the original program P (Figure 1a) executed on instance $I1$: The comparison of the `add` result to the ISS gave the same output; the mismatch could be seen on the incorrectly computed register value $t3$ of the equivalent program $\hat{P1}$ (Figure 1b) which executed the `slli` instruction on instance $I2$. Of course, also dedicated shift tests would have found this bug. However, the advantage of EPEX is that existing tests are *automatically* broadened and hence serve as an additional test.

Looking at the test program excerpt from Example 2 (`add` replaced by a `jump` and a `jal`), and one of the first VexRiscv cores from the GitHub repository, we first observed no problem when using EPEX and comparing the execution of P and $\hat{P2}$, respectively. However, we instructed EPEX to generate more test programs. An alternative test program $\hat{P3}$ contained:

```
1000: jalr zero, 2045(zero).
```

For this program EPEX reported an error. The reason is that the `jalr` instruction adds an immediate to its source register, then has to clear the LSB of the sum, and then jumps to the resulting address. In the VexRiscv implementation however, the LSB was not set to zero and hence the program counters differed. The problem occurs when odd addresses are generated via EPEX. This bug is nowadays well-known in the RISC-V community and has been found by *riscv-formal* [4] on the VexRiscv core at hand. Nonetheless, this demonstrates that EPEX can be used to generate relevant tests via relating a compute-instruction with two instructions from a completely different instruction class, namely control flow operations in this example.

4.2 Mutation Experiments

To evaluate the bug-finding capabilities of EPEX we performed a set of mutation experiments which we describe in the following.

Experimental Setup. For the experiments we used an area efficient VexRiscv configuration. We automatically created mutations of the Verilog code of the core. Mutants were created by toggling bit constants and modifying comparison operators (e.g. replacing greater than by greater than or equal). Overall this resulted in 275 mutants.

EPEX Generation and Results. We now consider three tests of the VexRiscv test-suite. For each test EPEX needs on average about five minutes to generate an equivalent program. The obtained results for our approach are summarized in Table 1. The first column reports the name of the test. The second column gives the number of instructions. In the third column the number of mutants are reported which have been found by the original test. The last column reports the number of mutants identified with EPEX. As can be seen, EPEX is able to find nearly 8% more of the mutants than the original test on average. To achieve this result, we configured EPEX to replace each instruction with up to three alternative instructions for the generation of the equivalent test program.

Table 1: Results for VexRiscv Tests

Test case	#Instrs	Orig	EPEX
ADDI	263	81	87
SRAI	234	84	88
JALR	218	84	93

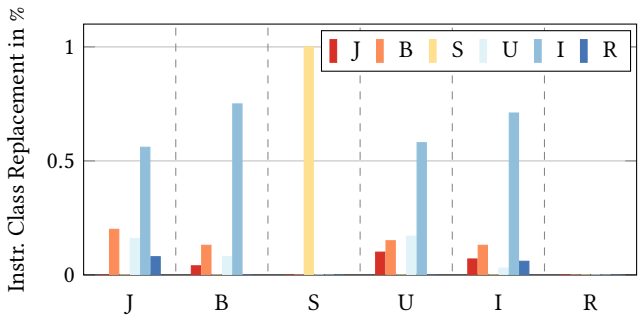

Figure 3: Exemplary distribution of instruction class replacements for ADDI test (cf. Table 1)

EPEX Test Broadening Result. We found a very interesting mutant, which was not detected even with extended cross-level testing (includes full compliance test-suite, Dhrystone and CoreMark). In contrast to this, EPEX was able to discover the mutant by using the three EPEX tests shown Table 1. This demonstrates that broadening given tests using EPEX already increases the test quality even when using a few basic tests only.

EPEX Test Diversity. To illustrate the diversity resulting from the EPEX test programs, we look on the *RISC-V instruction format*. It is divided into six classes[2]: **R**egister/register, **I**mmediate, **U**pper immediate, **S**tore, **B**ranch, and **J**ump. The instruction format has been carefully designed to share and save logic. Moreover, it is important to note that all instructions of a class share most of the data path and have an almost identical control; this is because the instructions within a class use the same number of registers, immediates etc. but of course differ in the specific operation. Based on this classification we now illustrate the effect of EPEX when generating an equivalent test program. For each class, the percentage of replacements into other classes for the ADDI test is shown in Figure 3.

For example, consider the class U (on the x-axis the fourth instruction format class): As can be seen an instruction from this

[2]See page 16, Figure 2.3 of "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA V20191213" [29]

Table 2: EPEX Control Path Variation on VexRiscv

Test	EPEX	#Instrs	Gen Ratio	Avg. Δ	Max. Δ
Compliance	1	8,834	100%	7.2%	36.0%
Compliance	1-4	12,923	100%	15.1%	45.3%
Dhrystone	1	38,798	10%	10.8%	33.0%
Dhrystone	1-4	41,931	10%	13.4%	38.0%

class is replaced in 10% with an instruction from J, in 15% with an instruction from B, in 0% of the cases with an instruction from S, in 17% of the cases with another instruction within this class, in 58% with an instruction from I, and in 0% of the cases with an instruction from R. Please note that instructions from the class S are intentionally never replaced as integrating a memory model into the formal model would make it much harder to solve for the SMT-solver. Also, the ADDI test does not contain any instructions of class R. Overall, this plot demonstrates that a good diversity in the sense of replacing an instruction with an instruction from another class can be achieved. Therefore, independent of a RISC-V core implementation, diverse testing becomes possible using EPEX.

4.3 Control Path Variation

Finally, we consider the control path variation produced by EPEX. We used the same experimental setup as in the previous section. As tests, we utilized the compliance test-suite and the Dhrystone benchmark. Table 2 lists the obtained results. The first column lists the name of the test. Column *EPEX* reports the number of instructions we configured to be replaced during equivalent program generation via the formal ISA model for an instruction. The column *#Instrs* shows the number of executed instructions. Then, column *Gen Ratio* gives the ratio of instructions of P that where selected for EPEX replacement. The next column *Avg.* Δ reports the average control path variation (stated in percent) triggered by $\hat{P}$ in comparison to P. For this, we carefully identified the select signals of the multiplexers which control the data path in the VexRiscv implementation along the pipeline stages. In the last column *Max.* Δ, the maximum delta found is provided, respectively. As can be seen EPEX causes a strong variation of the control paths of up to 45%. The obtained percentage values clearly demonstrate that large parts of the control paths differ during the execution of P on instance *I1* and $\hat{P}$ on instance *I2*, respectively. Overall, the strong control path variation shows that EPEX compares typically unrelated logic/operations against each other which enables a wide testing.

5 CONCLUSIONS

EPEX is a novel approach for processor verification that automatically broadens existing test programs. For this task a formal ISA model and SMT-reasoning is utilized to generate an equivalent test program. EPEX is based on a new form of equivalence checking: Executing both programs, i.e. original and equivalent, iteratively on two instances of the same processor design has to always result in equal architectural states. We demonstrated the bug-finding capabilities of EPEX on the VexRiscv processor. Broadening a few tests already resulted in the detection of more bugs.

In conclusion, EPEX is a promising new approach to processor verification. In particular the feature of relating the execution of instructions which activate different control and data paths has great potential. For future work we will investigate the replacement of longer instruction sequences in more detail. Furthermore, we will devise new guidance constraints in combination with novel coverage metrics to measure the difference in logic activity when running both programs on the two processor instances.

REFERENCES

[1] 2021. GitHub - VexRiscv: A FPGA friendly 32 bit RISC-V CPU implementation. https://github.com/SpinalHDL/VexRiscv.
[2] 2021. OneSpin 360 DV RISC-V Verification App. https://www.onespin.com/solutions/risc-v.
[3] 2021. RISC-V Compliance Task Group. https://github.com/riscv/riscv-compliance.
[4] 2021. RISC-V Formal Verification Framework. https://github.com/YosysHQ/riscv-formal.
[5] 2021. RISC-V ISA Tests. https://github.com/riscv/riscv-tests.
[6] 2021. RISC-V Torture Test Generator. https://github.com/ucb-bar/riscv-torture.
[7] 2021. RISCV-DV. https://github.com/google/riscv-dv.
[8] 2021. RISCV Sail Model. https://github.com/rems-project/sail-riscv.
[9] Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimon, Michael Vinov, and Avi Ziv. 2004. Genesys-Pro: innovations in test program generation for functional processor verification. *D&T* (2004), 84–93.
[10] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *SMT Workshop*.
[11] Niklas Bruns, Daniel Große, and Rolf Drechsler. 2020. Early Verification of ISA Extension Specifications Using Deep Reinforcement Learning. In *GLSVLSI*. 297–302.
[12] Brian Campbell and Ian Stark. 2014. Randomised Testing of a Microprocessor Model Using SMT-Solver State Generation. In *Formal Methods for Industrial Critical Systems*. 185–199.
[13] Mikhail Chupilko, Alexander Kamkin, Artem Kotsynyak, and Andrei Tatarnikov. 2017. MicroTESK: Specification-Based Tool for Constructing Test Program Generators. In *HVC*.
[14] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *TACAS*. 168–176.
[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. 337–340. Available at https://github.com/Z3Prover/z3.
[16] Keerthikumara Devarajegowda, Mohammad Rahmani Fadiheh, Eshan Singh, Clark W. Barrett, Subhasish Mitra, Wolfgang Ecker, Dominik Stoffel, and Wolfgang Kunz. 2020. Gap-free Processor Verification by S2QED and Property Generation. In *DATE*. 526–531.
[17] Shai Fine and Avi Ziv. 2003. Coverage directed test generation for functional verification using Bayesian networks. In *DAC*. 286–291.
[18] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. *SIGPLAN Not.* 46, 6 (June 2011), 62–73.
[19] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. Closing the RISC-V Compliance Gap: Looking from the Negative Testing Side. In *DAC*. 1–6.
[20] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. Towards Specification and Testing of RISC-V ISA Compliance. In *DATE*. 995–998.
[21] Vladimir Herdt, Daniel Große, Eyck Jentzsch, and Rolf Drechsler. 2020. Efficient Cross-Level Testing for Processor Verification: A RISC-V Case-Study. In *FDL*. 1–7.
[22] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. 2019. Verifying Instruction Set Simulators using Coverage-guided Fuzzing. In *DATE*. 360–365.
[23] Charalambos Ioannides, Geoff Barrett, and Kerstin Eder. 2011. Feedback-Based Coverage Directed Test Generation: An Industrial Evaluation. In *HVC*. 112–128.
[24] Yoav Katz, Michal Rimon, and Avi Ziv. 2012. Generating instruction streams using abstract CSP. In *DATE*. 15–20.
[25] Zohar Manna and Richard J. Waldinger. 1971. Toward Automatic Program Synthesis. *Commun. ACM* 14, 3 (March 1971), 151–165.
[26] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *ISSTA*. 261–272.
[27] SiFive. 2021. Formal Specification of RISC-V ISA in Kami. https://github.com/sifive/RiscvSpecFormal.
[28] Eshan Singh, David Lin, Clark W. Barrett, and Subhasish Mitra. 2018. Logic Bug Detection and Localization Using Symbolic Quick Error Detection. *TCAD* (2018).
[29] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley.
[30] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley.