Vol.2021-ARC-244 No.37
Vol.2021-SLDM-194 No.37
Vol.2021-EMB-56 No.37
2021/3/26

ソフトプロセッサの相互検証に関するケーススタディ

松川 達哉[1]　藤枝 直輝[1,a)]

概要：ソフトプロセッサの検証では，しばしばプロセッサシミュレータとソフトプロセッサとの間で命令実行結果を比較することが行われる．システムコールを含み，ファイルなどを扱う実用的なアプリケーションに対しては，シミュレータではシステムコールを模倣する対応法があるが，この方法をソフトプロセッサに適用するのは困難であった．本稿では，SystemVerilog を C/C++ 言語に変換する Verilator というツールを用いて，こうしたアプリケーションを用いたソフトプロセッサの検証を容易に行う手法を提案する．また，本手法を RISC-V ソフトプロセッサである kronos の検証に適用したケーススタディについて述べる．

A case study on cross-verification of soft processor

Tatsuya MATSUKAWA[1]　Naoki FUJIEDA[1,a)]

Abstract: Verification of a soft processor is often conducted by comparing results of instruction execution between a processor simulator and a soft processor. To simulate a practical application that includes system call, a simulator can emulate the execution of system call; however, it is difficult for a soft processor to adopt this approach. In this paper, we propose a method to apply system call emulation easily to a soft processor using Verilator, which converts SystemVerilog to C/C++ language. We also present a case study where the proposed method is applied to Kronos, a RISC-V soft processor.

1. はじめに

FPGA (Field-Programmable Gate Array) に実装して使用されるソフトプロセッサは，様々な場面で用いられる．FPGA ベンダーが提供するソフトプロセッサに，Xilinx 社の MicroBlaze [1] や，Intel（旧 Altera）社の Nios II [2] がある．これらはそれぞれの FPGA 向けに最適化されており，FPGA で周辺回路を含むシステムを設計するときには，周辺回路をソフトウェアで制御するためにしばしば用いられる．また，機能を限定するかわりに記述のシンプルさを追求したソフトプロセッサは，計算機工学を学ぶ学生がプロセッサの内部構造を理解する助けになる [3], [4]．さらに，プロセッサの新しいアーキテクチャやマイクロアーキテクチャの有効性を実証するために，高性能なソフトプロセッサが開発される場合もある [5]．

1 愛知工業大学
Aichi Institute of Technology
a) nfujieda@aitech.ac.jp

こうしたソフトプロセッサの開発を効率化する方法として，あらかじめ命令レベルで動作するプロセッサシミュレータを作成しておき，シミュレータとプロセッサとで命令トレースを比較する方法がある [3]．なお，以降「シミュレータ」はプロセッサシミュレータを，「プロセッサ」はソフトプロセッサを，それぞれ指すこととする．この方法では，シミュレータとプロセッサとそれぞれで，どの命令が実行され，それによりレジスタやメモリのどこに何が書き込まれたかを逐一記録し，トレースファイルを作成する．その後，出来上がった 2 つのファイルの差分を取ることで，実装に誤りのある箇所を速やかに見つけ出せる．検証には，アセンブリで書かれた短いテストプログラムを用いることが多い．同様の方法で，C 言語などで書かれた，より実用的なプログラムを用いた検証ができれば，開発効率をさらに高められる．

しかし，実用的なプログラム，とりわけファイルを扱うプログラムを用いた検証で問題になるのが，システムコールの扱いである．シミュレータでシステムコールに対応す

© 2021 Information Processing Society of Japan

表 1　本稿で対象とするシステムコールの一覧.

関数名	機能
`open()`	ファイルを開く
`read()`	ファイルからデータの読み出し
`write()`	ファイルへのデータの書き込み
`close()`	ファイルを閉じる
`lseek()`	ファイルの読み書き位置の変更
`exit()`	プロセスの終了
`brk()`	プロセスのヒープ領域の変更

る方法は 2 つある．1 つは，システムコールを模倣する，すなわちシステムコールに相当する処理をシミュレータが代行する方法である．もう 1 つは，OS も含んだシステム全体をシミュレートする方法である．あるいは，そもそもシステムコールを用いずに，ファイルの内容をあらかじめ配列などに格納するよう検証対象のプログラム自体を書き換えてしまう方法も考えられる．ただ，いずれの方法も，プロセッサに適用するのは困難であるか，煩雑な手続きが必要となる．

本研究では，ハードウェア記述言語の一種である SystemVerilog を C/C++ 言語に変換する Verilator [6] というツールに注目する．システムコールを模倣する方法をプロセッサに適用する上での障害は，シミュレータとプロセッサとで記述言語が異なることである．そのため，たとえシミュレータでシステムコールが模倣できたとしても，その記述をプロセッサの開発に応用することが難しい．Verilator でプロセッサの回路記述を C/C++ に変換してしまえば，プロセッサのテストベンチを C/C++ で記述できる．そのため，もしシミュレータを C/C++ で記述していれば，プロセッサでも同じようにシステムコールの模倣が実現できる可能性がある．

本稿では，上述した考えの実現可能性を立証するため，RISC-V（RV32I 命令セット）ソフトプロセッサである kronos に対して，システムコール模倣の機能を追加する方法を示し，その結果について報告する．そこではまず，kronos と基本的に等価な動作をする命令レベルのプロセッサシミュレータを作成する．次に，このシミュレータに対していくつかのシステムコールを模倣する機能を追加する．その後，kronos のテストベンチに対して，先のソースコードを極力流用しながら，同等のシステムコール模倣の機能を追加する．最後に，MiBench [7] のいくつかのアプリケーションに対して，シミュレータとプロセッサとの間での相互検証を行う．

2. システムコールの模倣

UNIX の API では，個々の OS の機能はシステムコールとよばれ，その処理はカーネルに実装されている．C 言語のプログラムが OS の管理する資源，すなわちファイルやプロセス，メモリ割当てなどにアクセスしようとするときには，システムコールを用いて OS に処理を委託しなければならない．本稿で対象とするシステムコールを**表 1** に示す．本稿では主にファイルを扱うアプリケーションを扱うので，対象とするシステムコールも，ほとんどがファイルへのアクセスに関連するものである．

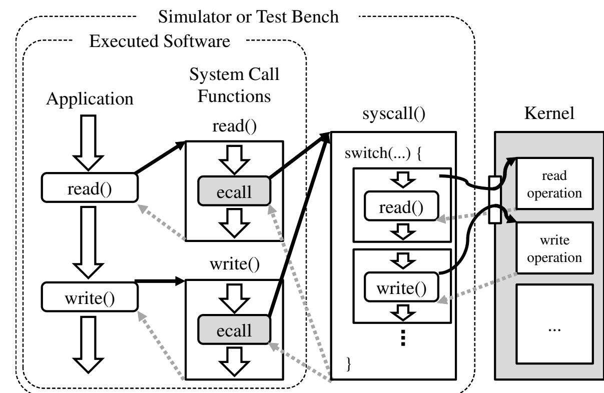

図 1　システムコール模倣の動作の概略.

プロセッサシミュレータの gem5 [8] や SimMips [3] では，こうしたシステムコールに対する扱いが異なる 2 種類の動作モードを持っている．システムコールを模倣するモードは，gem5 では SE (System call Emulation) モード，SimMips では App モードと名づけられている．OS も含めたシステム全体をシミュレートするモードは，gem5 では FS (Full System) モード，SimMips では OS モードという．

図 1 に，システムコールを模倣するときの，プロセッサ上で動作するプログラムとシミュレータとの間での処理の流れを示す．図の左半分はプロセッサ上で動作するプログラムの処理の流れ，右半分はシミュレータ（あるいはプロセッサのテストベンチ）の処理の流れである．

通常，シミュレータはプログラム中の命令を順番に実行する．アプリケーションが `read()` や `write()` などのシステムコール関数に差し掛かると，プログラムは必要な引数を準備してから，システムコール命令（RISC-V では ecall とよばれる）を実行する．このとき，システム全体をシミュレートしている場合は，単に特権モードへの変更やプログラムカウンタの書き換えなど，システムコール命令の仕様通りの処理を行えばよい．この場合のシステムコールの処理は，プロセッサ上で動作するカーネルによって行われる．システムコールを模倣する場合は，命令の実行は一旦中断され，シミュレータ上の模倣のための関数（以下 `syscall()` 関数とよぶ）が呼び出される．

`syscall()` 関数では，プロセッサの特定のレジスタの値を読み取り，システムコールの種類を特定する．そして，システムコールごとの固有の処理を行う．ここでは，対応するシステムコールの実行に必要な引数をプロセッサのレジスタやメモリから取り出し，システムコールを実行し

© 2021 Information Processing Society of Japan

Vol.2021-ARC-244 No.37
Vol.2021-SLDM-194 No.37
Vol.2021-EMB-56 No.37
2021/3/26

（シミュレータを実行している環境の OS が呼び出される），その結果をプロセッサのレジスタやメモリに書き戻す．場合によっては，引数に含まれる定数の定義が異なることがあり，その際は引数の読み替えなどの処理も必要である．その後，システムコール命令の次の命令から，プロセッサの実行を再開する．

3. kronos シミュレータの作成

3.1 対象の選定

本研究では，主にファイルを扱う実用的なアプリケーションを対象に，シミュレータとプロセッサとの間での相互検証を容易にする方法の提案をめざす．その方法の実現可能性を立証するため，本稿では，既存のソフトプロセッサである kronos [9] を土台に，シミュレータを作成したあと，各々にシステムコール模倣の機能を追加する．

kronos を土台とするに至るまでには，いくつかの技術的な選択を要した．まず，本研究はシミュレータやプロセッサの作成そのものが目的ではないので，両方をフルスクラッチで作成するのは目的に対する労力が大きすぎる．また，公開されているシミュレータの多くは，高性能化や高機能化のため複雑な内部構造をもっている．そのため，既存のシミュレータに合わせてプロセッサを作成するのも，やはり労力が大きい．そのため，既存のプロセッサに合わせてシミュレータを作成することとした．

プロセッサの選択にあたっては，まず使用する命令セットを選択した．その結果として，我々は RISC-V の最も基本的な命令セットである RV32I を選択した．RV32I は基本整数命令セットであり，整数の算術・論理・シフト演算，分岐命令やジャンプ命令など，基本的な命令をひととおり備える．また，システムコール命令（ecall）も RV32I に含まれている．その一方で，実装が複雑化しやすい乗算命令や除算命令，浮動小数点命令などの算術演算は含まれていない．簡潔に言えば，必要最低限の命令のみを備えているということが，RV32I を選択した理由である．

ソースが公開されていて RV32I に対応するソフトプロセッサは，kronos，Maestro [10] などいくつか存在する．本稿ではその中でも，SystemVerilog で記述されており，RV32I の互換性テストの実行に Verilator を使用していることが確認できた kronos を採用した．

なお，C++をベースとしたハードウェア記述言語に ArchHDL [11] がある．これは，レジスタ転送レベルのハードウェア記述を C++ の文法を利用して行うことで，論理シミュレーションの高速化とテストベンチの柔軟性の向上を図るものである．C++ でテストベンチを記述できるという点から，ArchHDL でも本研究の目的は達成できる可能性がある．しかしながら，ArchHDL の実行環境や，ArchHDL で記述された手頃なプロセッサを見つけることができなかったため，今回は使用を見送った．

```
switch (OP) {
  case 0b01100: // R-TYPE
    aluop = ((funct7 >> 2) & 0b1000) | funct3;
    src1 = reg[rs1];
    src2 = reg[rs2];
    alu(OP, src1, src2, aluop, &result);
    reg[rd] = result;
    break;
  case ...
```

図 2 kronos シミュレータの命令実行部の抜粋．

3.2 シミュレータの実装

作成したシミュレータは，RV32I の命令のうち，メモリフェンス命令（fence, fence.i），ブレークポイント命令（ebreak），およびステータスレジスタ関連命令（csr*）を除く RV32I 命令を実行できるものである．システムコール命令（ecall）が実行されると，システムコールの模倣のためのルーチンが動作する．単一プロセッサの命令レベルシミュレータでは，メモリのアクセス順序が入れ替わることはない．そのため，メモリフェンス命令は何の効果も及ぼさない．また，ブレークポイント命令やステータスレジスタ関連の命令，kronos プロセッサに実装されている特権命令（mret，wfi）は，いずれも割込みの存在が前提となるため，今回は実装を見送った．

図 2 に，作成した kronos シミュレータの命令実行にかかる部分の一部を示す．フェッチした命令をフィールドごとに分割したら，オペコードフィールドの上位 5 ビットの値 OP によって，命令の種類による場合分けを行う．算術論理演算（R 形式）のオペコードであれば，ALU での操作の種類を判別し，2 つのソースオペランドをレジスタから読み出してから，ALU での演算（`alu()` 関数）を行う．そしてその結果をレジスタに書き戻す．つまり，場合分けのあとは，プロセッサにおけるデコードステージ後半（オペランドフェッチ）以降の処理を個別に記述する．

命令レベルのシミュレータの記述では，プロセッサの各ステージでの動作を個別の関数で記述するスタイルも考えられる [3]．そうしたスタイルは，重複した記述を避けられる，ソースコードを読む際の見通しが立てやすいといった利点がある一方，事前にプロセッサの挙動の整理が必要で，開発の見通しを立てづらい．本稿では効率的な開発を優先し，図 2 に示した記述スタイルを採用した．

3.3 シミュレータの検証

kronos シミュレータの検証には，RISC-V 財団が公開する RV32I の互換性テスト（Compliance Test）をもとに，kronos 向けのヘッダファイルやリンカスクリプトの追加が施されたもの [12] を使用する．このテストは，各命令の挙動が RISC-V の仕様と一致しているかを検査する，アセンブリで書かれた短いプログラムの集合であり，kronos プロ

© 2021 Information Processing Society of Japan

```
switch (a7) {
  case 63: // read
    fd = fd_ary[*a0];
    *a0 = read(fd, (int *)mem_addr + (a1 >> 2 & 0xffff), a2);
    break;
  case ...
```

図 3 kronos シミュレータに実装した syscall() 関数の抜粋.

```
switch (top->regwr_data) {
  case 63: // read
    fd = fd_ary[a0];
    top->kronos_compliance_top__DOT__u_dut__DOT__u_if__DOT__u_rf__DOT__REG[10] =
      read(fd, &top->kronos_compliance_top__DOT__u_mem__DOT__MEM[0]+ (a1 >> 2 & 0xffff), a2);
    top->kronos_compliance_top__DOT__u_dut__DOT__u_ex__DOT__u_csr__DOT__mtvec = top->PC + 4;
    break;
  case ...
```

図 4 kronos のテストベンチに実装した syscall() 関数の抜粋.

セッサはそれら全てのプログラムで合格している．kronos シミュレータは，命令を実行するたびに，実行された命令のプログラムカウンタ，16 進数の機械語，命令名，書き込み先オペランドとその値をトレース情報として出力する機能をもつ．C++で記述された kronos プロセッサのテストベンチにも同等の機能を付加した．シミュレータとプロセッサのそれぞれで出力されたトレースファイルの差分を取り，シミュレータの検証を行った．

検証の結果, I-ECALL-01, I-EBREAK-01, I-MISALIGN_JMP-01，I-MISALIGN_LDST-01 の 4 つを除いた全てのプログラムで，両者のトレースは一致した．kronos シミュレータでは，ecall 命令はシステムコールを模倣するためのトリガであり，ebreak 命令は実装していない．また，kronos プロセッサでは 4 バイトのワード境界をまたぐ分岐命令やロード・ストア命令が実行されると例外が発生し，例外ハンドラによるソフトウェア処理でこれらに対処する．しかし，こうした例外処理も kronos シミュレータには実装していない．したがって，これらでトレースが一致しないのは意図した挙動である．

以上の結果から，kronos シミュレータは，意図的に実装を省いた割込み・例外処理に関する部分を除き，kronos プロセッサと命令レベルの全ての挙動が一致することが確認できた．

4. システムコールの模倣の実装

4.1 kronos シミュレータへの実装

本節では，シミュレータとプロセッサの双方にシステムコール模倣の機能を追加する．kronos シミュレータでは，ecall 命令が実行された（オペコードの上位 5 ビットが 0b11100，かつオペコード以外のフィールドが全て 0）場合，syscall() 関数を実行することにより，システムコールの模倣の処理へと移る．

図 3 に，kronos シミュレータの syscall() 関数から，read() システムコールにかかる部分を抜粋したものを示す．RISC-V では，システムコールの種類を示す番号はレジスタ a7 (x17) に格納され，引数は a0(x10) から順に格納されている．syscall() 関数ではこれらをあらかじめ読み出し，変数に保存している．

ファイルディスクリプタは，常に決まった順番で値を返せるように，読み替え処理を行っている．プログラムから open() システムコールが呼ばれると，シミュレータを実行している環境の open() システムコールを実行する．このとき得られたファイルディスクリプタをそのままプログラムに返すのではなく，その値を配列 fd_ary に保存した上で，プログラムには 3 番から始まる連番を返す．read() を含むその他のファイルを扱うシステムコールを模倣するときは，これとは逆の変換を行うために，配列 fd_ary の値を読み出す．

読み替えが終わったら，実際にシミュレータを実行している環境の open() システムコールを実行する．第 2 引数にみられる mem_addr は，データメモリに相当する配列へのポインタである．

4.2 kronos のテストベンチへの実装

Verilator を使用すると，一部を除く信号の値の読み書きを，配列や変数の読み書きの形でテストベンチから行える．これにより，ecall 命令が実行されていることを検知し，syscall() 関数を実行し，結果に応じて信号値やメモリの内容を書き換えれば，kronos プロセッサのテストベンチにシステムコールの模倣の機能を追加できる．なお，テストベンチには，RV32I の互換性テスト用に作られたものを引き続き使用する．

図 4 に，kronos のテストベンチの syscall() 関数から，read() システムコールにかかる部分を抜粋したものを示

© 2021 Information Processing Society of Japan

表 2　kronos シミュレータおよびプロセッサでのプログラムの実行時間.

プログラム名	命令数 [$\times 10^3$]	kronos シミュレータ		kronos プロセッサ	
		実行時間 [s]	実行速度 [KIPS]	実行時間 [s]	実行速度 [KIPS]
basicmath	10,000	46.6	215	66.5	150
bitcount	10,000	46.3	216	64.7	155
qsort	n/a	n/a	n/a	n/a	n/a
susan (smooth)	10,000	46.3	216	67.2	149
susan (edge)	8,092	37.2	217	53.8	150
susan (corners)	4,295	19.9	216	28.4	151

す．図 3 との主な変更点には下線を付す．kronos プロセッサのマイクロアーキテクチャはパイプライン方式を採用している．そのため，必要なデータがまだパイプライン上にのみ存在する，すなわちデータハザードが生じるおそれがある．具体的には，ecall 命令が実行されている時点では，システムコールの種類を示す a7 レジスタにはまだ書き込みが行われていない．そのため，レジスタへの書き込みデータ（regwr_data）の値を a7 レジスタの値のかわりに使用する．

レジスタへ値を書き込む，あるいはデータメモリに相当する配列へアクセスする際は，それぞれ kronos プロセッサのレジスタファイルやデータメモリを直接参照する．本来ならば，こうした記述は関数へと抽象化することにより，シミュレータとテストベンチとの間で揃った記法とすることも可能であると考えられる．そうした改善は今後の課題である．最後に，kronos プロセッサの ecall 命令による分岐先（mtvec 制御レジスタ）の値を，ecall 命令の次の命令を指し示すように書き換えることで，read() システムコールを模倣する処理が完了する．

5. 評価

5.1 評価方法

システムコールの模倣を追加した kronos シミュレータおよびプロセッサに対して，ファイルの操作を含むベンチマークを用いて相互検証を行う．プログラムはそれぞれ，終了するか 1000 万命令実行されるまで実行される．その間のトレースを比較することにより，シミュレータとプロセッサの動作が一致することを確認する．また，シミュレータの実行とプロセッサの論理シミュレーションに要する時間も併せて測定する．

対象のアプリケーションは，組込みシステム向けのベンチマークである MiBench [7] の automotive カテゴリーのプログラムである．このカテゴリには，基本的な数学関数の計算を行う basicmath，ビット列中の ‘1’ の数を数える bitcount，整列プログラムの qsort，画像のフィルタ処理を行う susan から構成される．なお，susan はフィルタのそれぞれ異なる 3 つ（smooth，edge，corners）のプログラムに分かれる．

開発および評価は，Core i5-9500 プロセッサおよび 16 GB のメモリを搭載した PC に，Windows 10 上 (Windows Subsystem for Linux) の Ubuntu 20.04 の環境を構築して行った．GCC のバージョンは，x86-64 向けのネイティブコンパイラでは 9.3.0，RV32I 向けのクロスコンパイラでは 10.1.0 である．Verilator はバージョン 4.037 を使用した．

5.2 結果

各プログラムに対して相互検証を行った結果，qsort と bitcount を除く全てのプログラムでシミュレータとプロセッサの命令レベルの動作は一致した．qsort は fscanf() 関数の実行に必要な fstat() システムコールが未実装であるため，正常に動作しなかった．また，bitcount は時刻を取得する gettimeofday() システムコールが未実装であるため，その戻り処理の部分でのみ動作が一致しなかった．また，bitcount にはプログラム中に処理時間を出力する部分があるが，同様の理由により処理時間はいずれの場合でも 0.00 秒と表示された．

表 2 に，シミュレータおよびプロセッサで，各プログラムの実行に要した時間，および命令の実行速度を示す．実行速度の単位は KIPS (Kilo Instruction Per Second) である．qsort は正常に動作しなかったため，n/a と表記している．プログラムにあまり依存せず，シミュレータでは 215 KIPS 程度，プロセッサでは 150 KIPS 程度の実行速度が得られた．これは，実用的な時間で相互検証が可能であることを示している．

6. おわりに

本研究の目的は，Verilator を用いて，SystemVerilog で記述されたソフトプロセッサにおけるシステムコールの模倣を実現し，ファイルなどを扱う実用的なアプリケーションを用いたソフトプロセッサの検証を容易化することである．本稿では，その実現可能性を立証するために，kronos ソフトプロセッサを用いたケーススタディについて報告した．その結果，MiBench のいくつかのアプリケーションに対して，実際に検証が可能であることを示した．

現時点では限られた数のシステムコールしか実装していないため，MiBench の qsort のように，期待通りに検証できないアプリケーションが多く存在すると思われる．今後は，対応するシステムコールを増やしていくとともに，シ

© 2021 Information Processing Society of Japan

Vol.2021-ARC-244 No.37
Vol.2021-SLDM-194 No.37
Vol.2021-EMB-56 No.37
2021/3/26

ステムコールに関連する記述をなるべく共通化していくことが課題である.

参考文献

[1] Xilinx Inc.: *MicroBlaze Processor Reference Guide*, User Guide UG984 (v2019.1) (2019).
[2] Intel Corp.: *Nios II Processor Reference Guide*, NII-PRG 2020.10.22 (2020).
[3] 藤枝直輝, 渡邉伸平, 吉瀬謙二：研究・教育に有用な MIPS システムシミュレータ SimMips，情報処理学会論文誌，Vol. 50, No. 11, pp. 2665–2676 (2009).
[4] 末吉敏則，久我守弘，柴村英智：KITE マイクロプロセッサによる計算機工学教育支援システム，電子情報通信学会論文誌，Vol. J84-D-1, No. 6, pp. 917–926 (2001).
[5] Mitsuno, S., Kadomoto, J., Koizumi, T., Shioya, R., Irie, H. and Sakai, S.: A High-Performance Out-of-Order Soft Processor Without Register Renaming, *30th International Conference on Field-Programmable Logic and Applications*, pp. 73–78 (online), DOI: 10.1109/FPL.50879.2020.00022 (2020).
[6] Snyder, W.: Verilator, (online), available from ⟨`https://www.veripool.org/wiki/verilator`⟩ (accessed 2021-02-10).
[7] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T. and Brown, R. B.: MiBench: A free, commercially representative embedded benchmark suite, *2001 IEEE International Workshop on Workload Characterization*, pp. 3–14 (online), DOI: 10.1109/WWC.2001.990739 (2001).
[8] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D. and Wood, D. A.: The gem5 Simulator, *ACM SIGARCH Computer Architecture News*, Vol. 39, No. 2, pp. 1–7 (online), DOI: 10.1145/2024716.2024718 (2011).
[9] Pinto, S.: Kronos RISC-V, (online), available from ⟨`https://sonalpinto.github.io/kronos/`⟩ (accessed 2021-02-10).
[10] Chrisóstomo, J. V.: Maestro, (online), available from ⟨`https://github.com/Artoriuz/maestro`⟩ (accessed 2021-02-10).
[11] Sato, S., Kobayashi, R. and Kise, K.: ArchHDL: A Novel Hardware RTL Modeling and High-speed Simulation Environment, *IEICE Transactions on Information and Systems*, Vol. E101-D, No. 2, pp. 344–353 (online), DOI: 10.1587/transinf.2017RCP0012 (2018).
[12] Pinto, S.: GitHub - SonalPinto/riscv-compliance, (online), available from ⟨`https://github.com/SonalPinto/riscv-compliance`⟩ (accessed 2021-02-15).

© 2021 Information Processing Society of Japan