一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
CPSY2024-45,DC2024-116(2025-03)

システムコール模倣による ソフトプロセッサ相互検証手法の要件導出と拡張

杉山　皓星†　　藤枝　直輝††

† 愛知工業大学大学院 工学研究科　〒470–0392 愛知県豊田市八草町八千草 1247
†† 愛知工業大学 工学部　〒470–0392 愛知県豊田市八草町八千草 1247
E-mail: †, ††{v23711vv, nfujieda}@aitech.ac.jp

あらまし　ソフトプロセッサの開発手法の 1 つに，シミュレータとプロセッサとの間で実行結果を比較する，相互検証がある．ファイル操作などを含む実用的なプログラムでの相互検証は，パイプラインの実装ミスなどを検出し，検証のカバレッジを広げる手段として有効である．松川らは，Verilator を用いることで，システムコールを模倣する部分のコードを共通化できる相互検証手法を提案した．しかし，松川らの研究には，単一のプロセッサにしか対応していない，限られたプログラムにしか対応していないなどの問題点がある．そこで本研究では，相互検証手法において，プロセッサや模倣処理が満たすべき要件の導出を行う．また，この要件をもとに，相互検証手法が適用できるように，別のプロセッサに対する拡張を施す．

キーワード　システムコール，ソフトプロセッサ，相互検証

Derivation of Requirements for Soft Processor Cross-Verification Methodology using System Call Emulation and Its Extension

Kosei SUGIYAMA† and Naoki FUJIEDA††

† Graduate School of Engineering, Aichi Institute of Technology
†† Faculty of Engineering, Aichi Institute of Technology
†, †† 1247 Yachigusa, Yakusa-cho, Toyota-shi, 470–0392 Japan
E-mail: †, ††{v23711vv, nfujieda}@aitech.ac.jp

Abstract　Cross-Verification is one of the development methods for soft processors, where execution results are compared between a simulator and a processor. The use of practical programs including file operations is effective there in detecting implementation errors and expanding verification coverage. Matsukawa et al. proposed a method using Verilator, where a simulator and a processor could share code for system call emulation. A major problem of the method was that it only supported a single processor and a limited number of programs. In this study, we derive requirements that processors and emulation enviromnents must meet to adopt the method. We then extend two distinct processors to meet the requirements.

Key words　System Call, Soft Processor, Cross-Verification

1. はじめに

FPGA (Field-Programmable Gate Array) を使ったハードウェア開発が注目されることで，ソフトプロセッサが利用される例が増えている．ハードウェア記述言語 (HDL) で記述され，主に FPGA に実装されるプロセッサのことを，ソフトプロセッサとよぶ. ソフトプロセッサは，FPGA 上に実装した専用回路や様々な周辺回路をソフトウェア制御するために，しばしば用いられる．また，ソフトプロセッサを開発することは，計算機アーキテクチャの学習や，新しいアーキテクチャの有効性検証のためにも有用である．以下，本稿で単にプロセッサと表記した場合は，ソフトプロセッサのことをさす.

また，近年オープンソースアーキテクチャである RISC-V を採用したプロセッサが，教育用途や企業でのシステム開発などで広く用いられているようになった．RISC-V は設計の単純さや柔軟性・拡張性の高さという利点をもち，ソフトウェア開発のためのツールチェインのサポートも充実している．FPGA の 2 大ベンダーである Xilinx (AMD) と Altera (Intel) は，それぞれ

MicroBlaze V [1]，Nios V [2] で RISC-V を採用している．また，計算機アーキテクチャの教科書の中にも，RISC-V を題材としていたり，RISC-V 対応版をラインナップに加えるものが現れている [3], [4].

プロセッサの開発手法の 1 つに，相互検証 [5] がある．相互検証では，検証対象のプロセッサと，プロセッサの動作を命令レベルで再現するシミュレータとの間で，命令の実行結果を比較する．具体的には，プロセッサとシミュレータに同じプログラムを実行させて，命令レベルの動作ログを出力させる．その後，それぞれの実行ログを比較して，各命令の動作が一致するかどうかを確認する．相互検証には，求められる要件が 2 つある．1 つは，シミュレータが，プロセッサが採用する命令セットに準拠した，正しい動作をすることである．もう 1 つは，シミュレータとプロセッサのテストベンチの両方が，命令レベルの動作ログを出力する機能をもつことである．後者の要件のため，バイナリ変換などを用いた，高速動作することを主眼として設計されたシミュレータは，相互検証の用途にはそぐわない．

相互検証には，プロセッサの開発段階に応じた様々な命令列を用いる．開発の初期段階では，個々の命令の動作を確認するため，人の手で記述された短いテストプログラムを使用する．次に，命令の特定の組み合わせによって起きる不具合を発見するために，多量の命令列を使用する．その際，検証のカバレッジを高めるため，ランダムな命令列を自動生成する手法がしばしば用いられる [5], [6]．しかしこの手法では，実際の応用プログラムで出現する命令列をカバーしきれないおそれがある．そこで，より効果的な相互検証を行うためには，ファイル操作などを含む，高水準言語で記述された応用プログラムを用いることが効果的である．

ファイルの操作などを含む，実用的なプログラムを使った相互検証では，システムコールの扱いが問題となる．システムコールを含むプログラムの相互検証手法として，考えられる方法は 2 つある．1 つは，OS を含むシステムごとプロセッサを動作させることである．もう 1 つは，システムコールに相当する処理を検証環境に代行させることである．既存のシミュレータの 1 つである gem5 には，前者に対応する全システム (FS; Full System) モードと後者に対応するシステムコール模倣 (SE; System call Emulation) モードがある [7]．前者の方法には，目的に対して技術的コストが大きいという問題がある．後者の方法には，シミュレータでは容易に実現できても，ハードウェア記述のテストベンチからソフトウェアのシステムコール模倣処理をそのまま実行するのは困難である，という問題がある．

システムコール模倣を用いた相互検証の問題点に対し，松川ら [8] は，Verilator を用いた方法を提案した．Verilator は Verilog HDL を C/C++ に変換することで，ハードウェア記述を高速に検証するツールである．Verilator による変換を使用すると，テストベンチは C/C++ で記述できる．C/C++ で記述されたテストベンチは，システムコールを通じて検証環境の OS を呼び出せる．そこで，テストベンチでプロセッサが実行している命令を監視して，システムコール命令の実行を検出したら，システムコール模倣の処理を行うこととする．そうすれば，テストベンチからもシステムコール模倣が実現でき，しかもそのためのプログラムコードは，大部分をシミュレータのものと共通化できる．先行研究では，検証アプリケーションとして MiBench [9] の automotive カテゴリのプログラムを対象としている．また，検証対象のプロセッサとして kronos [10] を採用している．

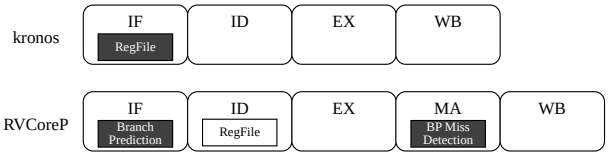

図 1　本稿で対象とするプロセッサ．

本研究の目的は，松川らの手法による相互検証を，最低限の記述変更で実現するための方法論を確立することである．先行研究は，単一のプロセッサと，限られた種類の応用プログラムにしか対応していなかった．そこで本稿では，複数のプロセッサと，より多くの応用プログラムに対応させるために，まずプロセッサや検証環境に求める要件を明確化する．そして，その要件にしたがって，プロセッサやテストベンチに修正を加え，実行速度や記述の変更量を評価する．本研究の内容は，対応する応用プログラムを増やすための拡張を含むが，本稿では，導出した要件と，プロセッサやテストベンチに加えた修正点について主に述べる．

2. 対象プロセッサ

本稿では，kronos [10] と RVCoreP [11] という 2 つのプロセッサを対象に，相互検証を行う．いずれも SystemVerilog または Verilog HDL で記述され，RISC-V 命令セットを採用したパイプラインプロセッサである．これらを選択した理由は 4 節で述べる．図 1 に，これらのプロセッサのパイプライン構成の概略を示す．以下，典型的な 5 段パイプラインプロセッサ [4] との差異を説明する．

kronos [10] のパイプラインは，IF (Instruction Fetch)，ID (Instruction Decode)，EX (EXecution)，WB (Write Back) の 4 段で構成される．典型的な 5 段パイプラインに存在する MA (Memory Access) は存在せず，kronos では対応する処理は EX ステージで行う．また，制御ハザードによる影響を最小化するために，レジスタファイル (RegFile) からのデータの読み出しは IF ステージで行う．図中には記載していないが，kronos には CSR (Control and Status Register) が実装されている．実行命令数は，そのうちの 1 つである `minstret` に格納される．

RVCoreP [11] では，1 バイトや 2 バイトのロード・ストア命令に対応するために，EX ステージでストア命令の処理を，MA ステージでロード命令の処理を行う．また，RVCoreP は分岐予測器 (Branch Prediction) を備えており，分岐予測ミスの判定 (BP Miss Detection) を MA ステージで行っている．なお，本研究では，オリジナルの RVCoreP ではなく，Thiem らによって外部データメモリが使用できるように拡張されたバージョンの RVCoreP [12] を使用する．

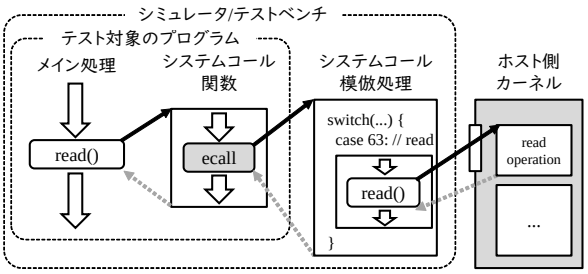

図 2　システムコール模倣処理の流れの概略図 [8].

3. システムコール模倣による相互検証

松川らは，Verilator を利用することで，システムコール模倣を適用した上でプロセッサの相互検証を行えることを示した [8]．この手法では，Verilator を使用して，Verilog HDL や SystemVerilog で記述されたプロセッサを C/C++ に変換する．これにより，テストベンチも C/C++ で記述することができ，メモリやレジスタへのアクセスは配列の読み書きへと変換できる．テストベンチでは，実行プログラムの書き込みやクロック信号の生成，実行ログの生成，システムコール模倣などを行う．テストベンチで実行ログを出力する場合，1 サイクルごとに特定のパイプラインレジスタの値を読み取る．もし有効な命令が存在していれば，その命令の機械語，実行結果とその書き込み先を出力する．松川らの検証では，プロセッサには kronos，テストケースには MiBench の automotive カテゴリのプログラムを採用している．クロスコンパイラの riscv-gnu-toolchain [13] を使用して，RISC-V 向けの実行バイナリを生成している．

システムコール模倣処理の流れを，図 2 に示す．相互検証を行うためには，まずメモリにプログラムを書き込み，プログラムカウンタをプログラムの先頭に設定する．次に，プロセッサにプログラムを実行させる．シミュレータの場合には，1 命令分のシミュレートを繰り返し行う．テストベンチの場合には，リセットパルスを一時的に与えたあと，周期的なクロックを与える．シミュレータでは 1 命令分のシミュレートのときに，テストベンチでは特定のパイプラインレジスタに有効な命令が存在するときに，命令の実行ログを出力する．またその際，RISC-V のシステムコール命令である `ecall` 命令が実行されたかどうかを調べる．もしシステムコール命令が実行されたのであれば，システムコール模倣処理に移行する．システムコールの種類は，レジスタ `x17` (`a7`) に格納される．また，システムコールの引数や返り値はレジスタ `x10` (`a0`) 以降に格納される．システムコール模倣処理では，システムコールの種類の特定と，対応するシステムコールをホスト側のカーネルに依頼する処理，プログラムカウンタの復帰処理を行う．ただし，実行されたシステムコールが `exit` であれば，シミュレーションを終了する．

4. 相互検証手法の要件導出

システムコール模倣手法による相互検証を行うためには，テストベンチやプロセッサが満たすべき要件がある．以下にその要件を示す．

（1） プロセッサが SystemVerilog または Verilog HDL で記述されていること．

（2） 実行ログの出力に必要な信号を，テストベンチから特定のタイミングですべて取得できること．

（3） プロセッサはシステムコール模倣処理を始める前に，それ以前のすべての命令の書き戻しを終えていること．

（4） プロセッサはシステムコール模倣処理を終えたら，パイプラインをフラッシュすること．

（5） システムコール模倣処理において，レジスタやメモリへの読み書きは関数呼び出しへと抽象化すること．

（6） システムコールが現在時刻を返す場合，実行のたびに変化しない時刻を返すこと．

（7） システムコールがファイルへのアクセス時刻を取得する場合，実行のたびに変化しない時刻を返すこと．

（8） システムコールがファイルの inode 番号を取得する場合，テストベンチとシミュレータとで同一のファイルにアクセスすること．

（9） テストベンチでレジスタやメモリに初期値をセットする前に，ハードウェアのリセットは完了していること．

（10） コマンドライン引数などを含む，レジスタやメモリにセットすべき初期値は，プログラムの実行を開始する前に書き込むこと．

このうち要件 1 と 2 は，システムコールの有無にかかわらず，そもそも Verilator を用いた相互検証を成立させるために必要な要件である．

要件 3 と 4 は，システムコール模倣により生じるデータハザードに対処するためのものである．要件 3 が必要なのは，システムコールの種類を表すレジスタ `x17` など，システムコール模倣で必要となるレジスタに書き込む命令が直前に存在する場合に，データハザードを引き起こしてしまうためである．要件 4 が必要なのは，システムコールの返り値を表すレジスタ `x10` など，システムコール模倣によって処理されたデータを扱う命令が直後に存在する場合に，やはりデータハザードを引き起こしてしまうためである．いずれの場合も，システムコール命令自体はオペランドをもたないため，プロセッサが通常備えるデータハザード検出ユニットで対処することは難しく，システムコール命令専用の対処が必要である．

要件 5 は，異なるプロセッサで相互検証を行うときに，相互検証環境の記述の変更を抑えるために必要な要件である．レジスタとメモリは開発対象のプロセッサやシミュレータによって実装が異なる．もしシステムコール模倣処理内でレジスタやメモリへのアクセスを抽象化せずに扱ってしまうと，開発対象のプロセッサ特有の記述があちこちに生まれてしまうことになる．この要件を加えることで，プロセッサやシミュレータの変更にともなう修正が最小限で済む．

要件 6 から 8 は，システムコール模倣処理によって，アーキテクチャ状態に差異が生まれる状況を回避するために必要な要件である．もしこれらの要件がなければ，プロセッサに問題がないにもかかわらず，相互検証の結果が異なるという事態が起こりうる．例えば，システムコールが現在時刻を取得してメモ

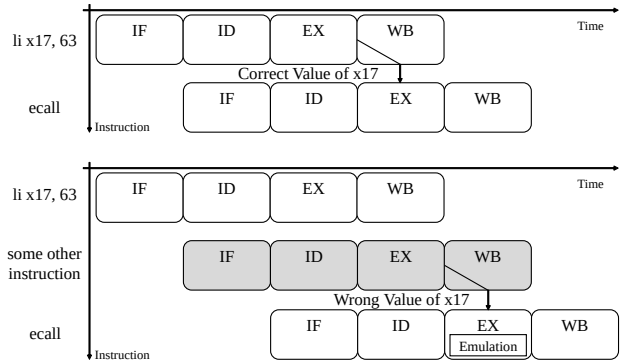

図 3　kronos で誤ったフォワーディングが発生する例.

リに書き込む場合，本当の現在時刻を書き込んでしまうと，その内容は実行するたびに異なってしまう．本手法ではテストベンチとシミュレータとは，それぞれ別々に実行することを想定している．そのため，これらの要件がなければ，それ以降の実行ログに差異が生まれてしまう．

要件 9 と 10 は，シミュレーション開始前のプログラムのロードや，スタックへの情報の書き込みに関するものである．プログラムカウンタやスタックポインタなどの値は，プロセッサによってはリセット入力がアサートされたときに初期化される．したがって，実行対象のプログラムに応じた初期化処理は，ハードウェアのリセットが行われてから，プログラムの実行を開始するまでの間に行う必要がある．

5. kronos の拡張と評価

本章では，4 節で述べた要件を満たすために，松川ら [8] が使用した kronos の相互検証環境に対して，追加で施した拡張について述べる．また，拡張後の kronos で相互検証を行うことで，その評価を行う．

5.1 kronos の拡張

kronos の拡張において最も重要なものは，データハザードに対する処理の変更である．システムコール模倣時のデータハザードについて，松川ら [8] は特定のステージからのフォワーディングによって対処していた．しかし，これは 4 節で定義した要件 3 を満たさない．そこで，システムコール模倣時のデータハザードへの対応を，フォワーディングからストールに変更する．

図 3 に，kronos で誤ったフォワーディングが起きる例を示す．松川らは，システムコールの種類をレジスタ `x17` に書き込む命令 (図中の `li` 命令) が，`ecall` 命令の 1 つ前に存在することを仮定していた．その仮定のもと，`ecall` 命令が EX ステージに到達したときに，システムコール模倣を行っていた．この場合，システムコールの種類は WB ステージからのフォワーディングで取得できる．しかし，松川らの仮定は常に満たされるものではなく，システムコールの種類や実行される状況によっては，2 つの命令の間に何か別の命令が現れる場合がある．この場合，フォワーディングでは誤った値を取得してしまう．

図 4 に，kronos の IF/ID ステージに実装したストール処理の抜粋を示す．この処理では，ID ステージでデコードされた命令が `ecall` 命令であり，かつ直前に ID ステージでストールが発生していない場合に，新たに定義したストール信号 `sys_stall` を 1 にする．そして，IF/ID ステージ間のパイプラインレジスタを更新する条件 `fetch_rdy` に，`sys_stall` が 1 でないことを追加する．これにより，`ecall` 命令が ID ステージで 1 サイクルだけストールする．その結果，`ecall` 命令が EX ステージに到達したとき，1 つ前の命令は WB ステージまで完了していることになる．これにより要件 3 を満たす．

```
assign sys_stall = (sysop == ECALL && decode_vld);
assign fetch_rdy = ((~decode_vld | decode_rdy) & ~stall)
                   & ~sys_stall;
```

図 4　kronos の IF/ID ステージに実装したストール処理の抜粋.

5.2 kronos の評価方法

5.1 節に述べたものを含む，4 節で述べた要件を満たすための拡張が施された kronos の記述量を測定し，全体の記述量に対する割合を求める．また，kronos の相互検証環境に対してプログラムの実行命令数と実行時間を測定し，kronos の相互検証のために行われるシミュレーションの実行速度を求める．そして，実際にログを比較することで相互検証を行う．テストプログラムには，MiBench のうち 14 種類のプログラムを使用する．ただし，実行命令数が 1000 万に到達した時点でシミュレーションを打ち切る．基準となるモデルには，松川らの開発した RV32I 命令セットシミュレータを使用する．

評価には，Core i5-13400 とメモリ 32 GB を搭載した Windows 10 マシンを使用した．マシン上の WSL2 (Windows Subsystem for Linux Version 2) に Ubuntu 20.04 を構築して検証を行う．なお, Verilator は 4.223, x86-64 向けのネイティブコンパイラ (g++) は 9.3.0，RV32I 向けのクロスコンパイラ (riscv-gnu-toolchain) は 11.1.0 のバージョンをそれぞれ使用した．

5.3 kronos の評価結果

kronos の総記述量は 1306 行であり，そのうち本研究で変更した箇所は 28 行である．これは，全体に対して 2.14 %であった．kronos での相互検証の結果は，使用した 14 種類のプログラムにおいて全て一致した．

表 1 に命令セットシミュレータ，表 2 に kronos のテストベンチの時間と検証速度を示す．シミュレータの実行速度は，プログラムにより最大で 20% 程度の差異を認めた．全プログラムでの平均実行速度は 1,035.38 KIPS であった．一方，kronos テストベンチでは実行時間の差異は 10% 程度にとどまり，平均実行速度は 749.85 KIPS であった．

6. RVCoreP の拡張と評価

本章では，新たな検証対象として，Thiem らによる拡張が施された RVCoreP [12] を用いる．システムコール模倣を含むテストベンチ部分は，kronos で使用したものをベースに，プロセッサに依存する記述を書き換えたものを使用する．その上で，4 節で述べた要件を満たすように，RVCoreP に対して拡張を施し，

表 1　命令セットシミュレータの実行時間と実行速度.

プログラム名	命令数 [$\times 10^3$]	実行時間 [ms]	実行速度 [KIPS]
basicmath	10,000	10,753	929.97
bitcount	10,000	10,241	976.47
qsort	10,000	10,651	938.88
susan (smooth)	10,000	10,280	972.76
susan (edge)	8,223	8,578	958.62
susan (corners)	4,490	4,036	1,112.49
jpeg (cjpeg)	10,000	9,077	1,101.69
jpeg (djpeg)	8,843	8,064	1,096.60
lame	10,000	9,021	1,108.52
patricia	10,000	9,022	1,108.40
dijkstra	10,000	8,998	1,111.36
stringsearch	197	184	1,070.65
sha	10,000	9,247	1,081.43
CRC32	10,000	9,275	927.50

表 2　kronos テストベンチの実行時間と実行速度.

プログラム名	命令数 [$\times 10^3$]	実行時間 [ms]	実行速度 [KIPS]
basicmath	10,000	13,287	752.62
bitcount	10,000	12,611	792.96
qsort	10,000	14,278	700.38
susan (smooth)	10,000	12,877	776.58
susan (edge)	8,223	10,694	768.94
susan (corners)	4,490	5,830	770.15
jpeg (cjpeg)	10,000	13,666	731.74
jpeg (djpeg)	8,843	11,932	741.12
lame	10,000	12,854	777.97
patricia	10,000	13,367	748.11
dijkstra	10,000	13,052	766.17
stringsearch	197	279	706.09
sha	10,000	13,567	737.08
CRC32	10,000	13,736	728.01

相互検証を通じた評価を行う.

6.1　RVCoreP の拡張

本節では，RVCoreP に施した拡張のうち，要件 2 を満たすために行った信号延長処理，要件 3 と 4 を満たすために行ったストール処理とフラッシュ処理について，それぞれ説明する.

まず，図 5 に，MA ステージに追加した信号延長処理の抜粋を示す．RVCoreP では，EX ステージではなく MA ステージの信号を，実行ログの出力や，システムコール命令が実行されたかどうかの判断に利用する．RVCoreP は，分岐予測ミスの判断を MA ステージで行い，分岐予測ミスがあればパイプラインをフラッシュする．つまり，EX ステージで実行ログを取得してしまうと，無効化されるはずの命令が実行ログに現れる可能性がある．一方で，RVCoreP ではストア命令に関する処理は EX ステージで行う．そのため，ストア先のアドレス D_ADDR や書き込み有効信号 D_WE は，MA ステージに送られることはない．そこで，これらの信号を EX/MA ステージ間のパイプラインレジスタに含める (図中ではそれぞれ D_ADDR_next と ExMa_D_WE) ことにより，実行ログの出力に必要なすべての信号を MA ステージで取得できるようにする.

```
reg [31:0] D_ADDR_next = 0;
reg [ 3:0] ExMa_D_WE = 0;
assign ExMa_D_ADDR = D_ADDR_next;
always @(posedge CLK) begin
    if (!D_STALL) begin
        D_ADDR_next <= D_ADDR;
        ExMa_D_WE   <= D_WE;
    end
end
```

図 5　RVCoreP の MA に実装した信号延長処理の抜粋.

```
wire IfId_syscall = ((IfId_ir[6:2] == 5'b11100)
                && (IfId_ir[14:12] == 3'b000)
                && (IfId_ir[19:15] == 5'b00000)
                && (IfId_ir[11:7]  == 5'b00000)
                && (IfId_ir[31:20] == 12'h000));
wire syscall_stall = ~(ExMa_syscall & ~ExMa_v)
                   & (IfId_syscall & IfId_v);
wire w_stall = IdEx_luse | syscall_stall;
```

図 6　RVCoreP の プロセッサコアに実装したストール処理の抜粋.

```
wire w_bmis = (ExMa_b
  & ((|ExMa_b_rslt) ? ExMa_b_rslt_t1 : ExMa_b_rslt_t2))
  | (ExMa_syscall & ExMa_v);
```

図 7　RVCoreP の プロセッサコアに実装したフラッシュ処理の抜粋.

次に，図 6 に，RVCoreP のプロセッサコアに実装したストール処理を示す．RCVoreP のデコーダは，そもそも ecall 命令をデコードできるようになっていない．そのため，まず ecall 命令をデコードしたときに 1 となる信号 IfId_syscall を定義し，これらを ID/EX ステージ間，EX/MA ステージ間のパイプラインレジスタに送る．そして，既存のストール条件 w_stall に新たな条件を加えるために，信号 syscall_stall を追加する．この信号は，有効な ecall 命令が IF ステージに到達したときに 1 となり，ストールにより無効化された ecall 命令が MA ステージに到達したときに 0 となる．すなわち，ecall 命令は ID ステージで 2 サイクルだけストールする．2 サイクルストールすることにしている理由は，当初は上述した信号延長処理を行わず，EX ステージで実行ログの出力を行おうとしていたためである.

最後に，図 7 に，RVCoreP のプロセッサコアに実装したフラッシュ処理を示す．RVCoreP では，分岐予測ミスを検出すると，既存のフラッシュ信号 w_bmis を 1 にする．これを，有効な ecall 命令が MA ステージに到達したときにも 1 となるように変更することで，システムコール模倣後のフラッシュを実現している.

6.2　RVCoreP の評価

RVCoreP の評価方法は 5.2 節に示した kronos のものと同じである．本稿では，先行研究で採用された kronos に加えて RVCoreP を Verilator に適用させることで相互検証を行う.

RVCoreP の総記述量は 332 行であり，そのうち本研究で変更した箇所は 66 行である．これは，全体に対して 19.88% であっ

表 3　RVCoreP テストベンチの実行時間と実行速度.

プログラム名	命令数 [$\times 10^3$]	実行時間 [ms]	実行速度 [KIPS]
basicmath	10,000	10,606	942.86
bitcount	10,000	10,261	974.56
qsort	10,000	11,145	897.26
susan (smooth)	10,000	10,650	938.97
susan (edge)	8,223	8,718	943.22
susan (corners)	4,490	4,757	943.87
jpeg (cjpeg)	10,000	10,598	943.57
jpeg (djpeg)	8,843	9,175	963.81
lame	10,000	10,586	944.64
patricia	10,000	10,590	944.29
dijkstra	10,000	10,338	967.31
stringsearch	197	214	920.56
sha	10,000	10,343	966.84
CRC32	10,000	10,541	1,054.10

た．RVCoreP の変更量は kronos の約 2 倍であった．変更率でみた場合，kronos との差はさらに大きく，kronos と比べて約 8 倍となった．その要因は大きく 2 つである．1 つは，RVCoreP がモジュールの切り分けを最低限しか行わないコンパクトな記述であることである．もう 1 つは，6.1 節に述べたほか，実行ログの出力に必要な信号をテストベンチと接続するために，プロセッサの出力を追加したことが原因である．これらの信号を出力として明示しないと，Verilator により未使用信号とみなされ，信号値を更新する処理が最適化により除去されてしまう．

図 3 に RVCoreP のテストベンチの実行時間と実行速度を示す．実行速度そのものについても，プログラムによる実行速度の差異についても，おおむねシミュレータの場合と kronos テストベンチの場合との中間の傾向を示した．実行速度は，14 種類のプログラムの平均で 953.28 KIPS であった．kronos の場合よりも実行速度が高速であるのも，RVCoreP の記述のコンパクトさが起因していると考えられる．

RVCoreP で相互検証を行った結果，bitcount，susan (edge, corners)，lame の 4 つのプログラムで，命令の実行ログに不一致を認めた．bitcount と susan (edge, corners) では，`ecall` 命令実行後の `lw` 命令で，取得するデータに差が生じた．本手法では，要件 6 や要件 7 を満たすために，システムコールが返す時刻を実行命令数から計算している．しかし，RVCoreP には実行命令数を保存するカウンタは実装されておらず，時刻に関わるデータを正しく取得できていない．この問題は，プロセッサ側ではなくテストベンチ側で実行命令数をカウントし，システムコール模倣に利用することで，解決できると考えられる．lame では，アドレス `0x5ae38` の命令が処理される際に，かわりにアドレス `0x1ae38` の命令が処理された．これは，RVCoreP のプログラムカウンタが 18 bit で宣言されていることが原因であると考えられる．

7. おわりに

本研究の目的は，松川らの手法による相互検証を，最低限の記述変更で実現するための方法論を確立することである．そのために，プロセッサと検証環境に求める要件を定義し，その要件を満たすよう kronos と RVCoreP の 2 つのプロセッサに対して修正を加えた．その結果，いずれのプロセッサに対しても，システムコール模倣による相互検証が行えることを示すことができた．

今後の課題としては，実行命令数が取得できることを新たな要件として定義し，改めて検証を行うことが挙げられる．また，現状テストベンチとシミュレータは，それぞれ別個に実行することで相互検証を行っている．これらを並行して実行する仕組みを実装し，シミュレータとテストベンチを実行しながら比較できるようにすることも，今後の課題である．

文　献

[1] Advanced Micro Devices Inc., “MicroBlaze V プロセッサリファレンスガイド,” User Guide UG1629 (v2024.1), 2024.

[2] Altera Corp., “Nios V Processor Reference Manual,” Technical Manual, no. 683632(v2024.10.07), 2024.

[3] 吉瀬謙二, RISC-V で学ぶコンピュータアーキテクチャ完全入門, 技術評論社, 東京, 2024.

[4] S. L. Harris and D. M. Harris, ディジタル回路設計とコンピュータアーキテクチャ RISC-V 版, 天野英春, 鈴木貢, 中條拓伯, 永松礼夫 (訳), エスアイビー・アクセス, 東京, 2022.

[5] V. Herdt, D. Grosse, E. Jentzsch, and R. Drechsler, “Efficient Cross-Level Testing for Processor Verification: A RISC-V Case-Study.” in 2020 Forum for Specification and Design Languages (FDL), pp.1–7, 2020.

[6] L. Klemmer and D. Grosse, “EPEX: Processor Verification by Equivalent Program Execution,” in 2021 Great Lakes Symposium on VLSI (GLSVLSI), pp.33—38, 2021.

[7] N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, and D.A. Wood, “The gem5 simulator,” ACM SIGARCH Computer Architecture News, vol.39, no.2, pp.1–7, 2011.

[8] 松川達哉, 藤枝直輝, ソフトプロセッサの相互検証に関するケーススタディ, 情報処理学会研究報告 2021-ARC-244, no.37, 2021.

[9] M. R. Guthaus J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, “MiBench: A free, commercially representative embedded benchmark suite,” in 4th IEEE International Workshop on Workload Characterization (WWC), pp.3-14, 2001.

[10] Pinto, S., “Kronos RISC-V,” https://sonalpinto.github.io/kronos/ 参照 Jan. 9, 2025.

[11] H. Miyazaki, T. Kanamori, M. Islam, and K. Kise, “RVCoreP: An optimized RISC-V soft processor of five-stage pipelining,” IEICE Transactions of Information and Systems, vol.103-D, no.12, pp.2494–2503, 2020.

[12] “RVCoreP,” https://github.com/thiemchu/rvcorep, 参照 Jan. 9, 2025.

[13] RISC-V International, “RISC-V GNU Complier Toolchain,” https://github.com/riscv-collab/riscv-gnu-toolchain, 参照 Jan. 9, 2025.