CC-001

RISC-V 拡張としての STRAIGHT ISA 向けコンパイラツールチェインの設計および評価
Design and Evaluation of a Compiler Toolchain for STRAIGHT ISA as a RISC-V Extension

杉田 脩* 門本 淳一郎* 入江 英嗣*
Shu Sugita Junichiro Kadomoto Hidetsugu Irie

1 はじめに

高性能プロセッサの性能向上が技術的に困難になる中，プロセッサアーキテクチャ全体を見直す手段として，新しい命令セットアーキテクチャ (ISA) の研究の重要性が増している．高性能プロセッサの発熱の問題が顕在化して以降，クロック周波数の向上や命令レベル並列性の拡張といった従来の性能向上手段は限界に近づいている．このため，近年はマルチコア化によって並列実行を進め性能向上が図られてきたが，その効果も Amdahl の法則が示すように逓減しつつある [9, 10]．こうした背景のもと，ISA の再設計を通じたアーキテクチャの革新が模索されており，新しい形式のアーキテクチャ [11, 12, 16] が提案されている．

一方で，新しい ISA の評価は，対応するシステムソフトウェアの整備が不可欠であるため，容易ではない．とりわけコンパイラ，アセンブラ，リンカといったツールチェインはプログラム評価に必須であるが，それらを独自に実装して既存 ISA と同等の水準まで成熟させるには多大な労力を要する [1, 18]．一般的な ISA と命令表現が異なる STRAIGHT ISA [11] は，このような開発負荷が高い新規 ISA の典型例である．

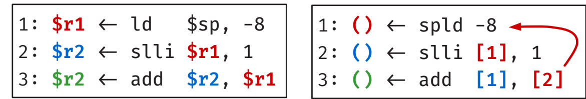
STRAIGHT ISA は，命令間距離を用いてオペランドを指定するアーキテクチャである．ここで命令間距離とは，「何命令前の命令の実行結果を計算に用いるか」のことである．一般的な命令セットアーキテクチャと STRAIGHT の ISA の違いを図 1 に示す．この指定方式の利点は，レジスタの上書きに起因する偽の依存が発生しないため，偽の依存を解消するリネーム機構を不要とすることにある．リネーム機構は電力効率が悪く，プロセッサコアの大規模化を妨げる大きな要因である [22, 26]．これを排することで，STRAIGHT は高性能プロセッサの大規模化とそれによる性能向上を可能とする．

STRAIGHT の有効性は，シミュレーション [13]，RTL 実装 [17]，ASIC 実装 [2] のそれぞれにおいて実証されている．しかし，評価に用いたベンチマークは C 言語で記述された比較的小規模なものに限定されている．これは，既存の開発環境 [24] の機能不足による制約が大きい．たとえばリンカはオブジェクトファイルの結合ができず分割コンパイルができない．また，使用しているコンパイラが LLVM 12 ベースと古く，新しいバージョンとの互換性に欠ける[1]ため，新しいツールや機能との連携が難しい．これは，新しい ISA に対するツールチェインの独自実装や，バージョン更新への追従に，高い開発コストが伴うことに起因している．

これに対し，本研究は STRAIGHT ISA を RISC-V の

* 東京大学

1 LLVM の IR は後方互換性を必ずしも持たない．たとえば，LLVM 17 においてポインタの表現のデフォルト方式が変更されている [15]．



(a) 従来型の ISA　(b) STRAIGHT ISA

図 1: 命令表現の違い．オペランドの指定に，従来型の ISA はレジスタ番号を用いるのに対し，STRAIGHT ISA は命令間距離を用いる．

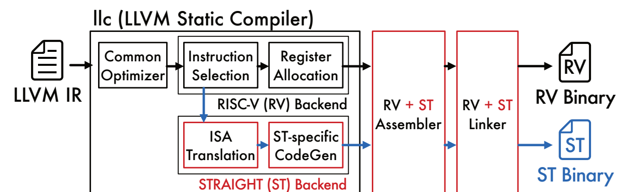


図 2: 提案手法によるコード生成フローの概略図．赤色で示した部分が提案手法で追加実装した部分である．また，青色の矢印で示したフローが STRAIGHT ISA 向けのコード生成フローである．

拡張命令セットとして位置づけ，既存のツールチェインを拡張することで STRAIGHT 向けツールチェインを実装する方式を提案する．ツールチェインの実装においては，通常 ISA ごとに命令定義やコード生成処理を独立に構築する必要がある．このような構成では，既存の ISA の実装や最適化の再利用には限界があった．一方で，本提案では，新規 ISA を既存 ISA の命令セットに統合することで，既存処理をほぼそのまま活用しつつ，固有の処理のみを分岐させて実装する（図 2）．これにより，開発負荷を抑えつつ，両 ISA に対して一貫した最適化や標準的な実行ファイル生成を可能にする基盤を実現する．

提案する設計に基いて STRAIGHT 向けのツールチェインを実装し，そのツールチェインの有用性について評価を行った．評価では，本ツールチェインによって，STRAIGHT において初めて可能となった機能や最適化に着目し，以下の 3 点を検証した．

1. 分割コンパイルを用いた大規模コード生成の実現
2. RISC-V の拡張命令セットの STRAIGHT への導入
3. 同一ツールチェインによる RISC-V プロファイルを用いた STRAIGHT のガイド付き最適化

これらを通じて，提案手法が新規 ISA に対して柔軟かつ実用的なコード生成を行えることを示した．

本研究の主な貢献は以下の通りである．

- 本研究は，既存 ISA のバックエンドの最適化機構を共有し，コンパイラ内部で命令セットの変換を行うことで新規 ISA 向けのコード生成を実現する，新たなコンパイラバックエンドの実装手法を示した．この手法は，静的バイナリ変換と異なり，STRAIGHT のような異なる命令表現を持つ ISA に対しても効率的なコンパイラ実装を可能とする．

Copyright © 2025 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

- 提案する設計により，非常に軽量な変更で，既存ISA の拡張命令セットを新規 ISA に導入可能であることを，実装と評価を通じて示した．
- 実装のベースとした ISA 上で得られたプロファイル情報を用いる，クロスアーキテクチャのガイド付き最適化について示し，STRAIGHT において実証した．これにより，提案する設計では新規 ISA でもガイド付き最適化を容易に行えることを示した．

2 背景

2.1 STRAIGHT アーキテクチャ

STRAIGHT アーキテクチャ [11] は，偽の依存が生じない命令セットを採用した，従来よりも省電力・高性能なアーキテクチャである．このアーキテクチャでは，各命令の実行結果の書き込み先をフェッチした順番に割り当てる．そして，レジスタオペランドの指定は命令間の距離，つまり参照元の命令と参照先の値を生成する命令との距離で行う．この方式では，値の上書きに起因する偽の依存が生じず，電力効率が悪くスケーラビリティに乏しい従来のリネーム機構は不要となる．これにより，STRAIGHT アーキテクチャはアウトオブオーダプロセッサのスケーラビリティを改善する．

STRAIGHT は，距離表現によるプログラムの記述を実現するために，従来の ISA と異なる仕様やコード生成手法を持つ．ここでは本研究の提案に関連した 1) 呼出規約，2) 特殊レジスタ SP，3) 距離制約の 3 点を説明する．それ以外の仕様や命令に関しては，オペランドの指定方法を除き，RISC-V の命令セット RV64G と同等である．

1) 呼出規約　サブルーチンを呼び出す際，あるいはサブルーチンから戻る際に，引数や戻り値を関数間でどのように受け渡すか規約を定める必要がある．通常この値の受け渡しはレジスタを介した方が高速となる．このレジスタによる受け渡しをレジスタ渡しと言う．例えば RISC-V では，第 1 引数は `$a0`，第 2 引数は `$a1` レジスタに格納するとしている．

一方，STRAIGHT は従来 ISA と同じ方式のレジスタ渡しを使うことができない．なぜなら，STRAIGHT は名前付きの汎用レジスタを持たないため，引数や戻り値を格納する場所をレジスタ名で指定できないからである．

これに対し，STRAIGHT では，以下のような距離に基づくレジスタ渡しの方法が提案されている．

- 関数コール命令 (`JAL`, `JALR`) の n 命令前の命令の結果を n 個目の引数とする．
- 関数リターン命令 (`RET`) の n 命令前の命令の結果を n 個目の戻り値とする．

2) 特殊レジスタ SP　STRAIGHT においてスタックポインタ (SP) は上書き可能な特殊レジスタとして用意される．この SP の値は，フロントエンドのみで処理可能な限られた命令でしか更新されない．そのため SP が上書き可能でも，STRAIGHT のリネームが不要という性質は保たれる．

SP に関する命令は 4 種類存在する．一つ目が SP を即値を足した値に更新する `SPADDi` 命令である．二つ目は `AUiSP` 命令で，これは即値 imm を取り $\mathrm{SP} + \mathrm{imm} \times 2^{20}$ を返す命令である．残りの二つは `SPLD/SPST` 命令で，SP との相対アドレスで指定したアドレスにロード/ストアを行う命令である．

3) 距離制約　STRAIGHT の機械語コードは，レジスタオペランドを距離で指定するために，以下の 2 つの距離に関する制約を満たす必要がある．

1. どのような実行パスを通っても同じ距離でオペランドを参照しなければならない．
2. 参照する距離は ISA で定められた上限値を超えてはならない．

STRAIGHT のコンパイラは命令配置という特有のコンパイル工程を行うことで，これらの制約を満たした距離表現のコードを生成する [13, 25]．

2.2 関連研究

新しい ISA に対するコンパイラをゼロから構築するのは多大な労力を要する．このため，新規 ISA の開発の初期段階など，ネイティブなコンパイラが未整備な状況においては，他の ISA 向けに生成した機械語コードを静的に変換して対象の ISA のコードを得るというアプローチが有効となる．このような技法を静的バイナリ変換 [1] という．一方で，静的バイナリ変換は，間接分岐など正確な制御フローの把握が困難であるため，変換精度と変換後の性能に限界がある．この課題を補うため，動的バイナリ変換で得られた実行時情報を利用して変換を行う手法が提案されている [5]．ただし，これらはホスト（新規 ISA）側の環境での実行が前提であり，コンパイラが未整備という問題設定においては適していない．

また，コンパイラから出力した補助情報を，コードの変換・最適化に利用するというアプローチが提案されている．Mortensen ら [18] は，マクロによって MIPS から SCALE ISA にバイナリ変換を行う際に，gcc の中間表現コードを最適化に利用する asopt (assembly optimizer) を提案した．しかし，命令の表現や呼び出し規約が異なる ISA に対しては，マクロでの変換がそもそも難しく，適用が難しい．

バイナリを直接変換するのではなく，柔軟なコード生成を実現するために，一旦中間表現 (IR) に戻してからターゲット向けのコードを生成するという手法およびツールが提案されている [4, 21]．このアプローチは，直接コンパイルするために必要なライブラリ実装がなくても，IR が得られるという利点がある．一方で IR から対象の ISA 向けにコードを生成するためには，依然としてターゲット ISA のコンパイラバックエンドが必要となる．このため，本研究の問題設定には適合していない．

このように，既存のバイナリ変換手法は，ソースコード不要という利点がある一方で，命令セットや ABI が大きく異なる ISA 間では変換の適用が難しく，柔軟なコード生成は困難である．結果として，新規 ISA 向けのコード生成にはコンパイラバックエンドの実装が不可欠となる．本研究では，その中でも実装コストを抑えることを目的とし，コンパイラ内部で ISA 間の命令変換を行うことで新規 ISA 向けのコード生成を行う手法を提案する．

3 STRAIGHT ツールチェインの設計

3.1 概要

STRAIGHT ISA のような新しい命令形式を持つ ISA に対して，新たにコンパイラツールチェインを開発することは多大な労力を要する．これは，実行バイナリ生成までのフローには多くの構成要素が関与しており，それらをすべて独自実装して正常に動作させることは容易では

Copyright © 2025 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

なく，またデバッグも難しいためである．実際，既存のツールチェイン [24] ではコンパイラバックエンド・アセンブラ・リンカを独自に実装しているが，評価に最低限必要な機能のみを持ち，十分な機能を備えていない．例えば，このコンパイラはジャンプテーブルを扱えない．また，リンカは複数のオブジェクトファイルをリンクできず，分割コンパイルができない．

そこで我々は新規 ISA のツールチェインを効率的かつ実用的に構築する新しいアプローチを提案する．提案手法では，新規 ISA を既存 ISA の拡張命令セットと位置付けて，既存 ISA のツールチェインを拡張する形で新規 ISA 向けのツールチェインを実装する．コンパイラは，ISA 間で中間最適化や命令選択のプロセスを共有しつつ，それぞれに固有の処理のみ分岐して実行するという構成を採用する．アセンブラとリンカは，新規 ISA に対応する拡張命令セットが選択されたとき，そのアセンブリを処理できるように拡張する．このように共通する処理や最適化を実績ある既存ツールチェインの実装に委ねることで，ISA 固有部分の開発に注力でき，それ以前の最適化の水準も揃えられる．

本研究では，RISC-V のツールチェイン実装をもとにした STRAIGHT ISA のツールチェイン設計について述べる．RISC-V をベース ISA として選択した理由は，2.1 節で述べたように，RV64G の各命令に対応する命令が STRAIGHT に存在するためである．

我々が実装したツールチェインの構成とその実行フローは図 2 の通りである．この構成では，STRAIGHT と RISC-V のコード生成を同じソフトウェア群を使って行う．これにより，生成されるコードの最適化水準や使用するライブラリが揃い，ISA 間の比較において統一的な評価が可能となる．以降では，3.2 節でコンパイラの設計を，3.3 節でアセンブラおよびリンカの設計を示す．

3.2　コンパイラの設計と実装

3.2.1　構成

RISC-V のバックエンドを最大限活用して STRAIGHT のバックエンドを実装するために，RISC-V でレジスタ割り付けを行う直前に STRAIGHT の処理に移行するという構成を採用する．これにより，命令選択やそれ以前のアーキテクチャ依存の最適化を適用できる．

RISC-V から STRAIGHT へのコード生成処理の切り替えは拡張名 `xstraight` の指定の有無によって行う．これを指定した場合，この拡張名はアセンブリ中に attribute として出力される．これを利用することで，出力が STRAIGHT の機械語コードであるという情報をアセンブラに渡すことができる．

本研究では，コンパイラ基盤 LLVM [14] に STRAIGHT のコンパイラバックエンドを実装した．以降の説明は LLVM への実装を前提としているが，GCC など他のコンパイラに対しても同様の機構により STRAIGHT のコンパイラバックエンドを構成できると考えられる．

3.2.2　コンパイルの流れ

図 3 に我々の設計における STRAIGHT コンパイラの最適化パスとその適用順序を示す．STRAIGHT において，プログラムを命令間距離による表現に変換するためには，変数をスタックに退避させるレジスタスピル (RegisterSpill) と，静的な距離で参照できるように命令の

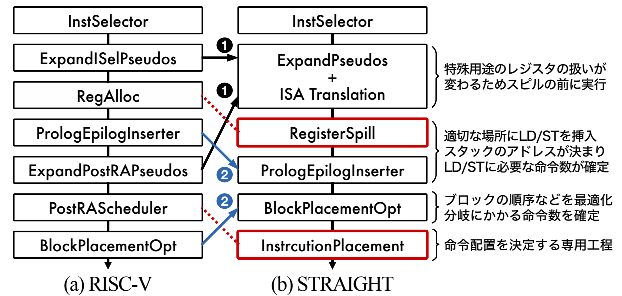


図 3: 実装したコンパイラの最適化パスの適用順序とその違い．本研究と関係のない最適化パスについては省略している．赤枠の最適化パスは STRAIGHT のコード生成において基本となるコンパイル工程である．破線はその工程と対応するコンパイル工程を示す．矢印は適用順序が入れ替っているものを示す．

順序と位置を調整する命令配置 (InstructionPlacement) を，この順序で行うことが不可欠である [25]．この図ではこれら二つの工程を赤色で示している．また，RISC-V と STRAIGHT における最適化パスの適用順序の変更を矢印で示している[2]．これらの変更の理由は，矢印の番号ごとに以下の通りである．

1. STRAIGHT はリターンアドレスを汎用レジスタで管理するなど特殊な値の扱いが変わるため，先に擬似命令を展開し，ISA を変換する必要がある．
2. 命令配置では命令の数が事前に決まっている必要があるため，相対アドレスの大きさによって必要な命令数が異なるスタックアクセスや制御フローの分岐については，命令配置より前に命令数を確定させなければならない．このため，それらを行う PrologEpologInserter と BlockPlacementOpt の適用順序を変更する必要がある．

次の 3.2.3 項において，本研究で新しく実装した ISA の変換処理について説明する．それ以降に行われる STRAIGHT 特有のコンパイル工程の詳細については，レジスタスピルに関しては [13] を，命令配置に関しては [13, 25] を参照されたい．

3.2.3　ISA の変換

本研究で提案するコンパイラの設計は，ベースとなる ISA の中間最適化の実装を活用しつつ，レジスタ割り付けの直前で処理を分岐させ，専用のコード生成処理へ移行するというものである．このとき，プログラムの中間表現は原則として仮想レジスタによる SSA 形式で表現されている．しかし，命令選択処理により，物理レジスタなどアーキテクチャに依存した要素も部分的に含まれている．そのため，処理を移行する際には，変換先のアーキテクチャの仕様に適合するようにアーキテクチャ依存部分を変換する必要がある．

STRAIGHT ISA は基本的に RISC-V の仕様を踏襲しているが，RISC-V から STRAIGHT への命令列の変換においても一部の修正を要する．特にその仕様の違いが顕著となるのが特殊用途のレジスタである．対象となるレジスタは以下の 3 種である．

- `$ra`：リターンアドレスを格納．

2　最適化パスごとに入力するプログラムの前提条件（SSA-form であることやレジスタ割付後であることなど）が異なるため，単純な順序の入れ替えのみでは最適化が適用できず，適用するタイミングに合わせた実装の変更が必要であることに注意されたい．

Copyright © 2025 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

```
1: $x2 = frame-setup ADDI $x2, -16
2: SD %5:gpr, $x2, 8    (%stack.0)
3: ...
4: %6:gpr = LD $x2, 8   (%stack.0)
5: $x2 = frame-destroy ADDI $x2, 16
```
→
```
1: frame-setup SPADDI -16
2: SPSD.64 %5:gpr, 8    (%stack.0)
3: ...
4: %6:gpr = SPLD.64 8   (%stack.0)
5: frame-destroy SPADDI 16
```

(a) SP について操作・利用する命令の基本的な変換

```
1: %7:gpr = LUI  (sprel_hi)
2: %8:gpr = ADDI $x2, %7:gpr
3: %9:gpr = LD %8:gpr, (sprel_lo)
```
→
```
1: %8:gpr = AUiSP (sprel_hi)
2: %9:gpr = LD.64 %8:gpr, (sprel_lo)
```

(b) SP 相対のオフセットが大きい場合のメモリアクセスの変換

図 4: SP に関連する命令の変換の様子．図の左側が RISC-V での表現，右側が STRAIGHT への変換後の表現．両側で同じ色の部分は対応関係にある．

- `$a0`：引数・戻り値を格納．`$a1` 以降も同様．
- `$sp`：スタックポインタ (SP) を格納．

これらに対して，我々は 1) 呼出規約の変換，2) SP の仕様変換，のそれぞれを行う処理を新たに実装した．以降では，それぞれの具体的な実装を述べる．

1) 呼出規約の変更　レジスタ割り付けの前の中間表現では，関数呼び出し命令などにおいて `$x1` や `$x10` といった物理レジスタ（それぞれ `$ra`, `$a0` に対応）が，レジスタオペランドに用いられている．これは，引数や戻り値といった特定用途の値について，呼出規約に従って所定の物理レジスタに配置されるよう，割り当てを固定するためである．

一方で STRAIGHT においては，リターンアドレスや関数の引数も，距離に基づいて参照されるレジスタ群上で，他の変数と同様に統一的に扱われる．このため，コード中の RISC-V 依存の物理レジスタ（`$x1`, `$x10` など）を，仮想レジスタに置き換える変換処理を新たに実装した．これにより，STRAIGHT の呼出規約に適合したコード生成が可能となる．

2) スピルと SP の対処　STRAIGHT においては，SP は特殊レジスタとして管理されており，SP に関連する操作は専用の命令を用いて行われる．このため，RISC-V に依存した SP 操作を，STRAIGHT の命令セットに変換する必要がある．

レジスタスピルなどによって挿入されたスタックアクセスを行う命令群は，PrologEpilogInserter によって `$x2` (`$sp`) を用いる形に変換される．本実装では，この `$x2` を手掛かりとして，対応する命令を STRAIGHT 専用の SP 命令に置換する変換処理を追加した．図 4 に，その具体的なアセンブリ表現の変換の例を示す．

3.3　アセンブラ・リンカの実装

本研究では，STRAIGHT に対応するアセンブラおよびリンカを GNU Binutils 2.42 [8] をベースに実装した．GNU Binutils とは，アセンブラやリンカなどを含む機械語処理用のツール群である．このツール群は，RISC-V GNU Compiler Toolchain に統合されており，RISC-V アーキテクチャを標準でサポートしている．

本実装は，RISC-V と STRAIGHT の実行バイナリを同一のツールチェインで生成可能とすることを目的としている．この方針に基づき，RISC-V を対象とした既存の処理系に対して，STRAIGHT の機械語を処理する機構を追加した．実装の主な内容は，1) 命令定義の追加，2) 機械語フォーマットの追加，3) リロケーション情報の追加の 3 点である．

1) 命令定義の追加　GNU Binutils では，アセンブリを解釈するために，命令定義テーブルがアーキテクチャごとに用意されている．本研究では，STRAIGHT の命令を解釈可能とするために，RISC-V のテーブルに STRAIGHT の命令定義を追加した．命令定義においては，各命令に分類（instruction class）を設定できる．そして，アセンブラは特定の分類に属する命令のみを解釈対象とするように指定できる．この機構を利用して，STRAIGHT 命令専用の分類を新たに定義し，拡張名として `xstraight` が指定された場合にのみ STRAIGHT 命令を解釈するように実装した．

2) 機械語フォーマットの追加　STRAIGHT と RISC-V の命令のフォーマットは，オペランドの数や各フィールドのビット幅や位置に違いがある．これに対応するため，STRAIGHT 専用の命令フォーマットの定義を追加し，アセンブリから正しく機械語に変換できるようにした．

3) リロケーション情報の追加　分岐先のラベルなどのシンボル参照は，リンク時に最終的なアドレスが決定する．このとき，リンカは決定したアドレスに合わせて機械語の該当部分を書き換える必要がある．これを正しく行うために，アセンブラで設定されるのがリロケーション情報である．

リロケーション情報の中に含まれるものの一つとして，リロケーションタイプがある．このタイプは，書き込む値（絶対/相対アドレス，上位/下位ビット）や，書き込むフィールドを定めるために用いられる．STRAIGHT と RISC-V では，同じ機能を持つ命令でもオフセットを書き込むビット位置が異なるものがあるため，STRAIGHT 専用のリロケーションタイプが必要となる．このため，STRAIGHT 専用のリロケーションタイプを定義し，対応する処理を新たに実装した．

4　応用事例による評価

本章では，提案するツールチェイン設計の有用性を，いくつかの事例を通じて検証する．ツールチェインは現在初期段階の実装であり，網羅的な性能評価は今後の課題である．そこで，大規模コードへの対応やベースとした ISA との連携といった，本設計によって新たに STRAIGHT で可能となった機能に着目し，基本的な動作とその効果，応用可能性を評価する．

まず 4.1 節では，バイナリ変換ツール Biotite [4] を介したコード生成について検証する．大規模なコードに対して，提案したコンパイラ設計が正しく機能すること，およびそのコンパイル時間を評価する．4.2 節では，RISC-V の Zicond 拡張の STRAIGHT への導入を検証し，4.3 節では RISC-V バイナリによるプロファイル情報を用いたガイド付き最適化を評価する．これらの応用事例において，生成されたコードの実行性能や命令数を RISC-V と比較することで，提案手法の有用性を明らかにする．

4.1　Biotite を用いた STRAIGHT 向けコード生成

4.1.1　概要

本研究で提案したツールチェインの設計が有効に機能することを実証するために，実験としてバイナリ変換ツール Biotite [4] を用いたコード生成の事例を評価する．Biotite は，RISC-V の実行バイナリを静的解析して

Copyright © 2025 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

LLVM-IR に変換するツールである．これにより，特定アーキテクチャ向けのライブラリが存在しないプログラミング言語においても，RISC-V 向けに静的コンパイルが可能であれば，そのアーキテクチャ向けにコード生成が可能となる．しかし，間接分岐に対応する処理を実現するために，巨大な switch-case 構造が必要となることで，中間表現のコードの構造が複雑化し，コード量が肥大化する．この影響により，分割コンパイルの出来ない既存の STRAIGHT の開発環境においては，大規模なプログラムの現実的な時間でのコンパイルは難しいという課題があった．

提案するツールチェインは，分割コンパイルを含む標準的なバイナリ生成機構を備えており，Biotite によって得られる大規模かつ複雑な中間表現に対しても，コード生成時の負荷を大きく軽減することができる．本実験では，このような中間表現を対象として，分割コンパイルを用いたコード生成が正しく行えるかを確認するとともに, コンパイル時間を測定することで, 提案手法の有効性を評価する．実験対象プログラムには，アーキテクチャ評価の標準ベンチマークである SPEC CPU 2017 [3] の整数演算ベンチマーク 10 個を用いた．SPEC CPU 2017 のプログラムは C，C++，Fortran で記述される．このうち C++ および Fortran は STRAIGHT 向けのライブラリ実装が存在しないため，それらで記述されたベンチマークは Biotite を用いることで初めて STRAIGHT 向けコード生成が可能となるプログラムと言える．

4.1.2 評価手法

各ベンチマークについて, 次のようにSTRAIGHT バイナリと比較用の RISC-V バイナリを Biotite を介して生成した．まず，RISC-V 向けにビルドした SPEC ベンチマークに対し，Biotite を適用し，LLVM IR ファイルを生成した．そして，生成した IR に対し，実装した STRAIGHT 向けツールチェインを用いて，STRAIGHT と RISC-V の実行バイナリを生成した．この際，分割コンパイルのテストとコンパイル時間の高速化を目的に，生成された IR ファイルからプログラム中の関数ごとに中間表現を抽出し，それらを 16 並列で分割コンパイルした．コンパイルオプションは最適化なしの `-O0` とした．実験は Ryzen 5950X を搭載した Ubuntu 環境で実施した．

4.1.3 評価結果

生成した STRAIGHT の実行バイナリが正しく動作することを検証するため，STRAIGHT 開発環境 [24] を用いてエミュレーションを実施した．その結果，全てのベンチマークにおいて，生成元の RISC-V バイナリの出力と，本実験で生成した STRAIGHT バイナリの出力が一致した．これにより，提案したコンパイラ設計とアセンブラ・リンカが正しく機能していることが確認できた．

各ベンチマークを分割コンパイルした際の，ファイルごとのコンパイル時間の総和を 表 1 に示す．`602.gcc_s` を除くベンチマークでは，STRAIGHT 向けのコンパイルは RISC-V の 3–11 倍の時間を要した．この時間の差は，STRAIGHT の命令表現に起因する，命令間距離の調整処理に由来する．STRAIGHT は命令がオペランドを相対的な位置で参照するため，コード生成時に距離の計算や更新が必要となり，RISC-V よりも多くの実行時間を要する．

一方，`602.gcc_s` は例外的に長いコンパイル時間を

表 1: Biotite を介したベンチマークのコード生成における RISC-V と STRAIGHT の総コンパイル時間の比較.

SPEC CPU 2017 Int ベンチマーク名	コンパイル時間の和 [s]		比
	RISC-V	STRAIGHT	
600.perlbench_s	79.6	838.5	10.5
602.gcc_s	267.2	36972.2	138.4
605.mcf_s	19.5	104.4	5.3
620.omnetpp_s	152.4	553.4	3.6
623.xalancbmk_s	258.1	888.9	3.4
625.x264_s	39.3	279.8	7.1
631.deepsjeng_s	22.5	120.6	5.4
641.leela_s	79.4	300.5	3.8
648.exchange2_s	31.0	179.0	5.8
657.xz_s	25.9	114.5	4.4

示し，STRAIGHT のコンパイル時間の総和は RISC-V の 138.4 倍となり，10 時間を超えた．この極端な差は，STRAIGHT において 160 万命令に達するような，特に巨大な関数が存在することに起因する．このような関数では，制御フロー合流時の距離調整や遠距離参照が頻発するため，計算負荷が増大し，RISC-V に比べてコンパイルに要する時間が大幅に増加したと考えられる．また，STRAIGHT のバックエンド実装が初期段階にあり，コンパイル時間の最適化が未成熟であることも RISC-V に対するコンパイル時間の比が増大した一因と考えられる．

なお，実装したツールチェインの分割コンパイル機能によって，`602.gcc_s` のコンパイル時間は総和の 53％に短縮することができている．これは，過度に肥大化したプログラムのコードに対しても，コード生成を現実的な時間内で実施できることを意味し，大規模プログラムを用いた最適化戦略の設計や評価における開発負荷を大幅に軽減するものである．このことは，提案するツールチェイン設計が，新規 ISA の開発を行ううえで有効かつ実用的な基盤となることを示している．

4.2 Zicond 拡張命令セットの導入

4.2.1 概要

本設計のメリットの一つに，実装する ISA にベースとした ISA の機能を導入することが容易となる点がある．ここではその例として，RISC-V の拡張命令セットの一つである Zicond 拡張 [19] を STRAIGHT に導入する方法を示し，その効果について検証する．

Zicond 拡張は `czero.eqz`, `czero.neq` の 2 つの命令を導入する拡張命令セットである．この二つの命令はいずれも一つのデスティネーション `rd` と二つのソース `rs1`, `rs2` を取り，その機能は以下の三項演算と等価である．

- `czero.eqz` : `rd` ← `(rs2 == 0) ? rs1 : 0`
- `czero.neq` : `rd` ← `(rs2 != 0) ? rs1 : 0`

これらの命令は条件付きの代入を可能とし，プログラム中の分岐を削減することができる．そして，分岐予測ミスに由来する性能低下の抑制につながる．

4.2.2 実装

RISC-V に存在する拡張であれば同じ命令選択処理を STRAIGHT でも活用できる．このため，STRAIGHT において Zicond 拡張を導入するために必要な変更は次の 2 つで完了する．

- Zicond 拡張に対応する命令定義の追加
- Zicond 拡張の命令について RISC-V から STRAIGHT に変換する処理の追加

これらを実装するために要したコードの変更は，LLVM

Copyright © 2025 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

表 2: 性能評価に用いたプロセッサパラメータ

	8-fetch	12-fetch	16-fetch
Front-end width (W)	8	12	16
Front-end latency	RISC-V: 7 cycles, STRAIGHT: 5 cycles		
Execution units	Int $\times$ $(W/2+4)$, Float $\times$ 4, Load $\times$ 3, Store $\times$ 2, iMul $\times$ 2, iDiv $\times$ 1, fDiv $\times$ 1		
Reorder buffer (R)	1024	2048	4096
Logical registers	RISC-V: Unified $\times$ R, STRAIGHT: Unified $\times$ $R+127$		
Physical registers	(Sufficient amount)		
Scheduler (S)	256	384	512
Load-store queue	Capacity: Load $S/2$, Store $3S/8$		
Branch predictor	8-component TAGE [20] 130-bit history, 8 KiB storage		
Branch target buffer	4-way, 8192 entries		
Return address stack	16 entries		
Mem. dep. predictor	Store Set [6], 9-bit producer ID, 4096 entries		
L1 cache	128 KiB + 128 KiB, 8-way, 3 cycle		
L2 cache	8 MiB, 16-way, 12 cycle		
Prefetcher	Stream prefetcher [23] at L2, distance 8, degree 2		
Main memory	80 cycle		

に対する 32 行の追加と BinUtils に対する 17 行の追加のみとなり，非常に軽量だった．

4.2.3 評価

Zicond 拡張セットの導入による効果を検証するため，STRAIGHT と RISC-V における Zicond 拡張の有無による実行サイクル数と実行命令数の比較評価を行った．実行サイクル数と実行命令数は，シミュレータ onikiri2 [27] を利用して計測した．シミュレーションに用いたプロセッサパラメータは表 2 に示す．また，コンパイラの初期評価として，ベンチマークには CoreMark [7] のイテレーション 20 回分を使用した．

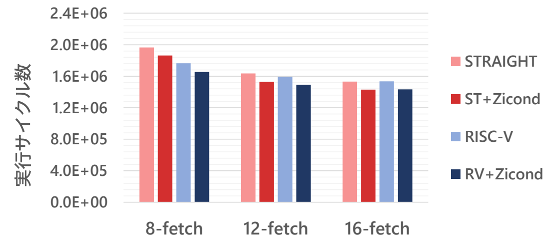
計測結果について，実行サイクル数を図 5 (a) に，種類ごとの実行命令数の内訳を図 5 (b) に示す．Zicond 拡張を導入することによる実行サイクル数の変化は，RISC-V と STRAIGHT ともに全条件で 6–7 % の削減が確認できた．これは，両方の ISA において実行される条件分岐命令が約 3 % 減少し，分岐予測ミスが減ったことで，プログラムの実行効率が上がったためである．

Zicond 拡張を導入した STRAIGHT と RISC-V について性能を比較すると，STRAIGHT は 8-fetch の条件で RISC-V の 89 %，12-fetch で 98 %，16-fetch で 100 % の実行性能を示した．8-fetch の条件において，STRAIGHT の実行性能が RISC-V に対し低下した主な要因は，STRAIGHT 特有のコンパイル工程の命令配置にある．STRAIGHT では，静的な距離でオペランドを参照するために MOVE 命令の挿入が不可欠である．この命令配置によって挿入された MOVE 命令は，STRAIGHT の実行命令数を RISC-V の 127 % に増加させる．フロントエンド幅や整数演算の実行ユニットが少ない条件では，この命令数の増加が性能に大きな影響を及ぼすため，実行効率が低下したと考えられる．一方で，12-fetch, 16-fetch の条件では，これらの追加命令が他のレイテンシの長い命令の実行間に十分に処理されるため，性能への影響がほぼ見られなかった．この MOVE 命令による性能影響は，コンパイラ実装をさらに成熟させることにより改善が可能となる．

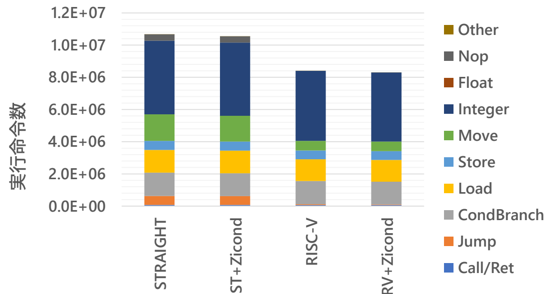
4.3 クロスアーキテクチャのガイド付き最適化

4.3.1 概要

本研究で提案するコンパイラ設計は，クロスアーキテクチャのガイド付き最適化 (PGO) を新たに実現する．PGO は，プログラムの実行プロファイルに基づいて行うコンパイラ最適化である．これは，ISA 自体のポテンシャルの評価や，コンパイラ最適化の設計に役立つ．なぜなら，コンパイラの最適化パスが適切に実装されていても，静的解析によって間違って予測された情報をもとに最適化した場合，その性能向上効果は限定的となってしまうからである．



(a) 各条件における実行サイクル数



(b) 種類ごとの実行命令数の内訳

図 5: STRAIGHT と RISC-V に Zicond 拡張を導入したときの CoreMark の実行サイクル数と実行命令数の変化．

本研究で検証するクロスアーキテクチャの PGO は，ベースとなる ISA で取得したプロファイル情報を新規 ISA の最適化に利用するという手法である．通常，異なる ISA で取得したプロファイル情報をアーキテクチャ依存の低レベルな最適化へ直接適用することは難しい．これは，アーキテクチャによってプログラムの処理や制御構造が変わる場合，プロファイル情報を活用できなくなるためである．しかし，本提案のコンパイラ設計では，新規 ISA をベース ISA の拡張として位置付け，命令選択を含む ISA レベルの最適化パスを共有する．そのため，同じ方法でコード生成を行う限り，共有する最適化パスにおいてプログラムの処理や制御構造が変わることはない．したがって，ベース ISA で取得したプロファイル情報は新規 ISA でも低レベル最適化に利用可能となり，より高い最適化効果が期待できる．

さらに，この手法は新規 ISA への PGO の導入コストを大幅に削減する．プロファイル情報を得るためにはファイル入出力やシステムコールの実行が可能なエミュレータ環境が必要となる．新規 ISA の初期開発段階において，このような環境は未整備であることが多く，PGO の実現は困難である．一方で本手法は，ベース ISA の既存のプロファイリング環境やエミュレータ環境を利用して，追加の開発コストなしに，そのプロファイル情報を新規 ISA の最適化に活用できる．

4.3.2 評価手法

異なる ISA でプロファイルを取得し，そのプロファイルを最適化に利用して生成した実行バイナリの評価を行うことで，クロスアーキテクチャ PGO の効果を検証す

Copyright © 2025 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

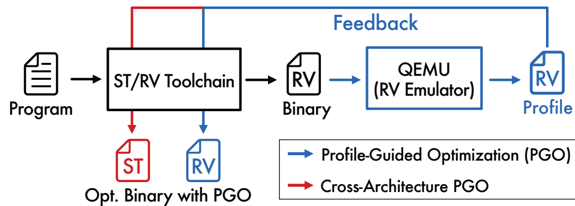


図 6: 本設計によるクロスアーキテクチャのガイド付き最適化のコード生成フローの概略図

る．用意したプロファイルは，実験を行った CPU でビルドした x86_64 バイナリから取得したものと，図 6 のように RISC-V 向けに一旦ビルドして QEMU によるエミュレーションで生成したものの 2 種類である．これらは，clang において `-fprofile-generate` オプションを付加してビルドしたプログラムを実行することで取得した．

PGO は `-fprofile-use=` オプションによってファイルを指定することで適用できる．この際，x86_64 バイナリによるプロファイルを PGO に利用した場合，プログラム中の一部の関数について，制御フローが異なっているためプロファイル情報を利用しないという警告が出力された．一方で，RISC-V バイナリのプロファイル結果を用いた場合は警告は出力されなかった．

評価は，Zicond 拡張を追加した STRAIGHT と RISC-V のバイナリを対象とし．それぞれの ISA に対して以下の 3 条件のバイナリを用意した．

- [w/o PGO] : PGO を全く行わない場合
- [w/x86_64] : x86_64 バイナリで取得したプロファイルで PGO を行う場合
- [w/RISC-V] : RISC-V バイナリのエミュレーションによるプロファイルを用いた場合（提案手法）

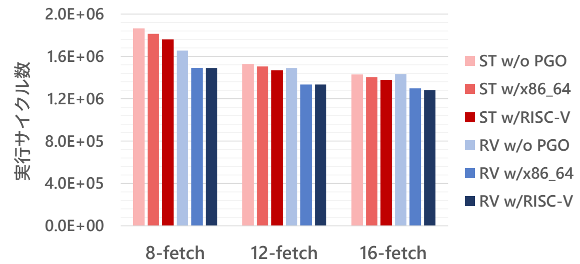
これらの実行バイナリに対して，シミュレーション実行を行い，実行サイクル数と実行命令数を計測した．実験環境と実験に使用したパラメータ，ベンチマークについては，4.2 節と同じものを用いた．

4.3.3　評価

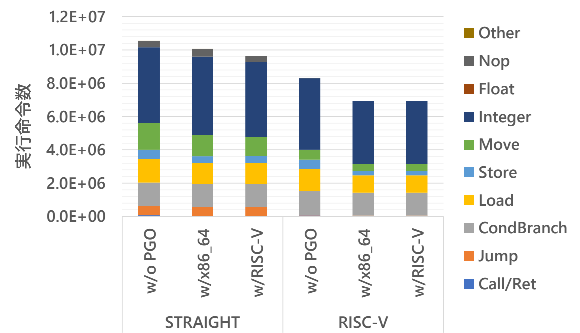
計測した実行サイクル数と実行命令数をそれぞれ図 7 (a), (b) に示す．STRAIGHT では，実行サイクル数と実行命令数がそれぞれ，w/o PGO, w/x86_64, w/RISC-V の順番に減少した．具体的には，実行サイクル数は w/x86_64 で 3–4 % 減少し，w/RISC-V で 4–5 % 減少した．実行命令数は w/x86_64 で 4.6 % 減少し，w/RISC-V で 8.8 % 減少した．これらの命令数削減の半分以上は，MOVE 命令の減少に起因している．RISC-V では，w/x86_64 と w/RISC-V ともに実行サイクル数が 10 % 減少し，実行命令数が 16.5 % 減少した．

測定結果から，(1) STRAIGHT よりも RISC-V の方が PGO による変化の幅が大きいこと，(2) STRAIGHT では w/x86_64 と w/RISC-V の結果に大きな違いが存在することがわかる．この 2 点について考察する．

(1) STRAIGHT よりも RISC-V の方が変化の幅が大きい理由は，この設計においては STRAIGHT が RISC-V のレジスタ割り当て以降の最適化を利用できないためと考えられる．今後，STRAIGHT のバックエンド実装を拡張し，より洗練されたものになれば，RISC-V と同等の最適化効果が得られる可能性がある．



(a) 各条件における実行サイクル数



(b) 各条件における種類ごとの実行命令数の内訳

図 7: プロファイル情報を用いたガイド付き最適化 (PGO) による実行サイクル数と実行命令数の変化．図中の w/o PGO は PGO を行わない場合，w/x86_64 と w/RISC-V はそれぞれ x86_64 バイナリによるプロファイルを利用する場合，RISC-V バイナリによるプロファイルを利用する場合を示す．

(2) STRAIGHT における w/x86_64 と w/RISC-V の結果の違いは，STRAIGHT における命令配置がプロファイル情報の影響を受けやすいことと，先述のプロファイル適用時の警告に由来する．命令配置では，異なる分岐パスにおけるレジスタオペランドの位置を揃えるために MOVE 命令が挿入される．この MOVE 命令は，実行頻度の高いパス上の命令位置に合わせて値の位置を MOVE 命令で調整することで削減可能である．このため，プロファイル情報の利用による実行命令数の削減効果は大きく，それに伴って実行性能も向上する．また，w/x86_64 では，Zicond 拡張を利用したことで中間表現における制御構造が変わり，プロファイル情報を利用できない関数が生じた．これらの結果により，そのような関数の実行において最適化の度合いが変わり，w/x86_64 と w/RISC-V で性能差が生じたと考えられる．

これらの結果により，提案するツールチェインの設計は，クロスアーキテクチャな PGO の適用において効果的であることが示された．なお，本研究において実装したツールチェインの構成では，コンパイラが LLVM ベース，アセンブラ・リンカが GNU GCC ベースとなっており，プロファイル情報をリンクタイム最適化に活用できない．これらを統一することができれば，より高い最適化効果が得られると考えられる．そのような最適化についての効果の検証は今後の課題とする．

5　おわりに

本研究では，新規 ISA に対するコンパイラツールチェイン開発という課題に対し，新規 ISA を既存 ISA の拡張命令セットとして位置付け，既存 ISA 向けツールチェ

Copyright © 2025 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

インを拡張する形で新規 ISA 向けツールチェインを実装するという新たな設計を提案した．この設計は，既存 ISA 向けのツールチェイン実装を最大限に活用することで，新規 ISA の開発負荷を軽減する．さらに，既存ツールチェインの持つ機能や最適化を新規 ISA にも容易に転用でき，新規 ISA 向けの実用的なツールチェインを効率的に実装できる．

この設計に基づき，本研究では，一般的な ISA と異なる命令表現を持つ STRAIGHT 向けのコンパイラツールチェインを，RISC-V ツールチェインの拡張として実装した．そして，STRAIGHT において初めて実現した機能や最適化として，(1) 分割コンパイルを用いた大規模コード生成の実現，(2) RISC-V の Zicond 拡張命令セットの STRAIGHT への導入，(3) クロスアーキテクチャのガイド付き最適化，の 3 点を検証・評価した．

今後の展望として，まずコード生成の自動最適化を通じた STRAIGHT アーキテクチャの最適化が考えられる．本研究により実用的な STRAIGHT 向けコンパイラツールチェインが実現し，中間最適化については RISC-V と同等の最適化手法を適用できるようになった．一方で，命令表現の特性上，STRAIGHT において最適な最適化オプションは異なる可能性が高い．そこで，自動最適化手法を用いて，STRAIGHT 固有の特性に最適化されたコード生成手法を探索・評価することが課題となる．これは，アーキテクチャのポテンシャルを引き出す重要な研究課題である．

また，STRAIGHT を RISC-V の拡張命令セットとして実装したことを生かした，RISC-V と STRAIGHT の混成プロセッサの実現も今後の展望の一つである．本研究の設計は，RISC-V に STRAIGHT を拡張命令として自然に統合できるため，システムの複雑化を抑えつつ，特性の異なる ISA 間の協調動作を効率的に実現できる可能性を示している．このような構成によって，異なる命令セットの特性を同一環境下で比較・検証できるような枠組みが実現し，ISA 設計とアーキテクチャ評価がさらに発展することが期待される．

謝辞

本研究は JSPS 科研費 23K28050, 23KJ0500 の助成による．

参考文献

[1] E. Altman, D. Kaeli, and Y. Sheffer. Welcome to the opportunities of binary translation. *Computer*, 33(3):40–45, 2000.

[2] T. Amano, J. Kadomoto, S. Mitsuno, T. Koizumi, R. Shioya, H. Irie, and S. Sakai. An Out-of-Order Superscalar Processor Using STRAIGHT Architecture in 28 nm CMOS. In *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2023.

[3] J. Bucek, K.-D. Lange, and J. v. Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 41–42, 2018.

[4] C. Chen, S. Sugita, Y. Nada, H. Irie, S. Sakai, and R. Shioya. Biotite: A High-Performance Static Binary Translator using Source-Level Information. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, CC '25, pages 167–179, 2025.

[5] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. FX!32: A Profile-Directed Binary Translator. *IEEE Micro*, 18(2):56–64, Mar. 1998.

[6] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *The 25th Annual International Symposium on Computer Architecture*, ISCA, pages 142–153, 1998.

[7] EEMBC. CoreMark, 2009. URL: `https://www.eembc.org/coremark/`.

[8] GNU Project. GNU Binutils 2.42, 2024. URL: `https://www.gnu.org/software/binutils/`.

[9] J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, Jan. 2019.

[10] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[11] H. Irie, T. Koizumi, A. Fukuda, S. Akaki, S. Nakae, Y. Bessho, R. Shioya, T. Notsu, K. Yoda, T. Ishihara, and S. Sakai. STRAIGHT: Hazardless Processor Architecture without Register Renaming. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, MICRO 51, pages 121–133, 2018.

[12] T. Koizumi, R. Shioya, S. Sugita, T. Amano, Y. Degawa, J. Kadomoto, H. Irie, and S. Sakai. Clockhands: Rename-Free Instruction Set Architecture for Out-of-Order Processors. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, pages 1–16, 2023.

[13] T. Koizumi, S. Sugita, R. Shioya, J. Kadomoto, H. Irie, and S. Sakai. Compiling and Optimizing Real-world Programs for STRAIGHT ISA. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 400–408, oct 2021.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, 2004.

[15] LLVM. Opaque Pointers — LLVM 17.0.1 documentation, 2023. URL: `https://releases.llvm.org/17.0.1/docs/OpaquePointers.html`.

[16] R. Matsuo, T. Koizumi, H. Irie, S. Sakai, and R. Shioya. Enhancing GPU Performance through Complexity-Effective Out-of-Order Execution using Distance-based ISA. *IEICE Transactions on Information and Systems*, advpub:2024EDP7203, 2024.

[17] S. Mitsuno, J. Kadomoto, T. Koizumi, R. Shioya, H. Irie, and S. Sakai. A High-Performance Out-of-Order Soft Processor Without Register Renaming. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 73–78, sep 2020.

[18] A. Mortensen, S. Pomerville, D. Whalley, S. Onder, and G.-R. Uh. Facilitating the Bootstrapping of a New ISA. In *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2023, pages 2–12, 2023.

[19] RISC-V International. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Version 20250508, 2025.

[20] A. Seznec and P. Michaud. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction-Level Parallelism*, 8:1–23, Feb. 2006. URL: `https://jilp.org/vol8`.

[21] B.-Y. Shen, W.-C. Hsu, and W. Yang. A Retargetable Static Binary Translator for the ARM Architecture. *ACM Trans. Archit. Code Optim.*, 11(2), June 2014.

[22] R. Shioya and H. Ando. Energy efficiency improvement of renamed trace cache through the reduction of dependent path length. In *2014 32nd IEEE International Conference on Computer Design (ICCD)*, pages 416–423, oct 2014.

[23] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA, pages 63–74, 2007.

[24] straight-dev, 2022. URL: `https://github.com/straight-dev/env/`.

[25] S. Sugita, T. Koizumi, R. Shioya, H. Irie, and S. Sakai. A Sound and Complete Algorithm for Code Generation in Distance-Based ISA. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, CC '23, pages 73–84, 2023.

[26] Y. Tatsumi and H. J. Mattausch. Fast quadratic increase of multiport-storage-cell area with port number. *Electronics Letters*, 35:2185–2187, 1999.

[27] K. Watanabe et al. Processor simulator onikiri2, 2005. URL: `https://github.com/onikiri/onikiri2`.

Copyright © 2025 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.