はじめに

この文章は RISC-V の命令マニュアルを @shibatchii が RISC-V アーキテクチャ勉強のためメモしながら訳しているものです。
原文は https://riscv.org/specifications/ にある riscv-spec-v2.2.pdf です。

英語は得意でないので誤訳等あるかもしれません。ご指摘歓迎です。
Google 翻訳、Bing 翻訳、Webilo 翻訳、Exclite 翻訳 を併用しながら翻訳し、勉強しています。

まずは意味が分からないところもあるかもしれませんが、ざっくり訳して 2 周位回ればまともになるかなと。
体裁とかは後で整えようと思います。

文章は以下の様に色分けしてます。

黒文字：翻訳した文書。
赤文字：@shibatchii コメント。わからないところとか、こう解釈したとか。
青文字：RISC-V にあまり関係なし。訳した日付とか、集中力が切れた時に書くヨタ話とか。

2018/03/26　@shibatchii

RISC-V 命令セットマニュアル
第 I 巻：ユーザーレベルの ISA
ドキュメントバージョン 2.2

編集者：Andrew Waterman 1 、Krste Asanovic 1,2
1 SiFive Inc.、
カリフォルニア大学バークレー校の EECS 部門 2 CS 課
andrew@sifive.com、krste@berkeley.edu
2017 年 5 月 7 日

アルファベット順の仕様全バージョン貢献者（訂正があれば編集者に連絡してください）：クレステ・アサノビック、リマス・アヴィジエニス、ジェイコブ・バッハマイヤー、クリストファー・エフ・バッテン、アレン・ジェイ・バウム、アレックス・ブラッドベリー、スコット・ビーマー、プレストン・ブリッグス、クリストファー・セリオ、デイビッド・キスナル、ポール クレイトン、パーマー・ダッベルト、ステファン・フロイデンベルガー、ジャン・グレイ、マイケル・ハンブルク、ジョン・ハウザー、デイヴィッド ホーナー、オルフ・ヨハンソン、ベン・ケラー、ユンサップ・リー、ジョセフ・マイヤーズ、リシュユール・ニヒル、ステファン・オレア、 アルバート・オー、ジョン・オースターハウト、デヴィッド・パターソン、コリン・シュミット、マイケル・テイラー、ウェズリー・ターブストラ、 マット・トーマス、トミー・ソーン、レイ・バン・デーウォーカー、メガン・ワックス、アンドリュー・ウォーターマン、 ロバート・ワトソン、そして、レイノルド・ザンヂジク。

次のように引用してください：「RISC-V 命令セットマニュアル、第 1 巻：ユーザーレベル ISA、ドキュメントバージョン 2.2 "、編集者アンドリュー・ウォーターマンとクレステ・アサノビック、RISC-V 財団、2017 年 5 月。

配布条件はこれですね。https://creativecommons.org/licenses/by/4.0/deed.ja
制限が少ない大変良いですね。

-2018/03/26

序文

これは、RISC-V ユーザーレベルのアーキテクチャを説明するドキュメントのバージョン 2.2 です。

このドキュメントには RISC-V ISA モジュールの次のバージョンが含まれています。

ベース	バージョン	凍結?
RV32I RV32E RV64I RV128I	2.0 1.9 2.0 1.7	Y N Y N
拡張	バージョン	凍結?
M A F D Q L C B J T P V N	2.0 2.0 2.0 2.0 2.0 0.0 2.0 0.0 0.0 0.0 0.1 0.2 1.1	Y Y Y Y Y N Y N N N N N N

今日まで、RISC-V 財団によって正式に批准された規格はありませんが、上記の "凍結"と表示されているコンポーネントは、曖昧さや仕様の穴を解決する以上に批准プロセス中に変更されることはありません。

凍結ってなっているところは「曖昧なところや穴があったときは修正される」けど、仕様は今後変わらないよ。ってことらしい。

このバージョンのドキュメントの主な変更点は次のとおりです。

・このドキュメントの以前のバージョンは、元の作者が作成したクリエイティブコモンズ帰属 4.0 国際ライセンスの下でリリースされました。このドキュメントの今後のバージョンは、同じライセンスでリリースされる予定です。
・すべての拡張子を標準的な順序で最初に置くように章を再編成しました。
・説明と解説の改善。
・LUI / JALR と AUIPC / JALR ペアのより高度なマクロオペレーションの融合をサポートするための JALR に関する暗黙のヒント提案を修正しました。
↑なんじゃろ。JALR のところでててくるかな。

-2018/03/27

・ロード予約/ストア条件付きシーケンスに対する制約の明確化。
・コントロールとステータスレジスタ（CSR）のマッピングの新しいテーブル。
・fcsr の上位ビットの目的と動作を明確にしました。
・FNMADD.fmt および FNMSUB.fmt 命令の説明が修正されました。これは、ゼロ結果の符号が正しくないことを示唆していました。
・命令 FMV.S.X と FMV.X.S は FMV.W.X と FMV.X.W に変わらなかったそれらの意味論とより一致するようにそれぞれ名前が変更されました。古い名前は引き続きツールでサポートされます。
↑意味が合うように命令を変えたってことか。
・より狭い（<FLEN）浮動小数点値の指定された作用は、NaN-ボクシング・モデルを使っているより広い f レジスターをおさえました。
↑広い f レジスタに拡張(移行)されたってことかな。
・FMA（1、0、qNaN）の例外作用を定めました。
・P 拡張が整数レジスタを使用した固定小数点演算の整数パック SIMD 提案に再加工される可能性があることを示すノートを追加しました。
・V ベクトル命令セット拡張のドラフト提案。
・N ユーザレベルのトラップ拡張の早期草案。
・拡張された疑似命令リスト。
・RISC-V ELF psABI 仕様に取って代わった呼び出し規約の章の削除 [1]。
・C の拡張機能は凍結され、バージョン番号は 2.0 に変更されました。

-2018/03/28

文書 版 2.1 の序文

これは、RISC-V ユーザーレベルアーキテクチャを記載している文書の版 2.1 です。
凍結されたユーザーレベル ISA ベースと拡張 IMAFDQ 版 2.0 がこの文書の前の版から変わらなかった点に注意してください [36]、しかし、いくつかの仕様の穴は修正されました、そして、文章は改善されました。
ソフトウェアの規則にいくつかの変更が加えられました。

・解説セクションへの多数の追加と改良。
・各々の章のための別々のバージョン番号
・非常に長い命令フォーマットで rd 指定子を移動させないようにするために、64 ビットを超える長い命令符号化への変更。
・CSR 命令は、後で浮動小数点セクション（および付随する特権アーキテクチャマニュアル）に導入されるのとは対照的に、カウンタレジスタが導入される基本整数フォーマットで記述されます。
・SCALL 命令と SBREAK 命令は、それぞれ ECALL と EBREAK に名前が変更されました。それらのエンコーディングと機能は変更されていません。
・浮動小数点 NaN 処理の明確化、および新しい標準 NaN 値
・オーバーフローする浮動小数点から整数への変換によって返される値の明確化。
・LR / SC の明確化は、シーケンス内の圧縮命令の使用を含む、成功と必要な失敗を許しました。
・整数レジスタ数を削減し、MAC 拡張をサポートする新しい RV32E ベースの ISA 提案。
・改訂された呼び出し規約。
・ソフトフロート呼び出し規約のためのリラックスしたスタックアライメント、 および RV32E 呼び出し規約の説明。

-2018/03/29

・C の圧縮された拡張のための修正案、バージョン 1.9。

バージョン 2.0 の序文

これは、ユーザー isa の仕様の 2 番目のリリースであり、我々は、ベースユーザーの isa plus の一般的な拡張機能 (すなわち、IMAFD) の仕様は、将来の開発のために固定されたままにするつもりです。
この ISA 仕様のバージョン 1.0 [35] 以降では、以下の変更が行われています。

・ISA はいくつかの標準拡張を持つ整数ベースに分割されています。
・命令フォーマットは、即時エンコードをより効率的にするために再編成されています。
・ベース ISA は、リトルエンディアンのメモリシステムを持ち、ビッグエンディアンまたはバイエンディアンが非標準のバリアントであると定義されています。
・アトミック命令拡張で、Load-Reserved / Store-Conditional（LR / SC）命令が追加されました。
・AMO および LR / SC は、リリース一貫性モデルをサポートできます。
・フェンス命令は、より細かいメモリと I / O 順序を提供します。
・fetch-and-XOR（AMOXOR）の AMO が追加され、部屋を作るために AMOSWAP のエンコーディングが変更されました。
↑部屋を作る って場所を空けるっていう感じか。
・AUIPC 命令は、PC に 20 ビットの上位の即値を追加し、現在の PC 値のみを読み取る RDNPC 命令を置き換えます。これにより、位置に依存しないコードが大幅に節約されます。
・JAL 命令は、明示的なデスティネーションレジスタを持つ U タイプフォーマットに移動し、J 命令は、RAL = x0 の JAL に置き換えられ、なくなりました。
これにより、暗黙のデスティネーションレジスタを持つ唯一の命令が削除され、ベース ISA から J-Type 命令フォーマットが削除されます。これに伴い、JAL の到達範囲が縮小されますが、ベース ISA の複雑さは大幅に軽減されます。
・JALR 命令の静的ヒントが削除されました。このヒントは、標準の呼び出し規約に準拠したコードの rd および rs1 レジスタ指定子では冗長です。
・ハードウェアを単純化し、補助情報を関数ポインタに格納できるように、JALR 命令は計算されたターゲットアドレスの最下位ビットをクリアするようになりました。
・MFTX.S 命令と MFTX.D 命令は、それぞれ FMV.X.S と FMV.X.D に名前が変更されました。同様に、MXTF.S および MXTF.D 命令は、それぞれ FMV.S.X および FMV.D.X に名前が変更されました。
・MFFSR および MTFSR 命令は、それぞれ FRCSR および FSCSR に名前が変更されました。
fcsr の丸めモードおよび例外フラグのサブフィールドに個別にアクセスするために、FRRM、FSRM、FRFLAGS、および FSFLAGS 命令が追加されました。
・FMV.X.S および FMV.X.D 命令は、rs2 ではなく rs1 からオペランドがソースになります。この変更により、データパス設計が簡素化されます。
・FCLASS.S および FCLASS.D 浮動小数点分類命令が追加されました。
・より単純な NaN 生成および伝播方式が採用されています。
・RV32I の場合、システムパフォーマンスカウンターは 64 ビット幅に拡張されており、上位および下位 32 ビットへの個別の読み取りアクセスが可能です。
・標準的な NOP および MV エンコーディングが定義されています。

-2018/03/30

・標準的な命令長エンコーディングは、48 ビット、64 ビット、および >64 ビット命令で定義されています。
・128 ビットアドレス空間バリアント RV128 の説明が追加されました。
・32 ビットの基本命令フォーマットの主なオペコードは、ユーザ定義のカスタム拡張のために割り当てられています。
・rd のデータをストアすることを示唆している誤植は、rs2 を参照するように修正されました。

-2018/04/01

内容

序文

↑Atomic ってどういう意味だろう？ なんて訳すのがいいのか。

-2018/04/02

-2018/04/03

1 ページ空き。次ページへ。

第 1 章

前書き

RISC-V（「リスク 5」と発音）は、コンピュータアーキテクチャの研究と教育をサポートするためにもともと設計された新しい命令セットアーキテクチャ（ISA）であり、
私たちが今望んでいるのは、業界標準の自由で解放されたアーキテクチャです。
RISC-V を定義する私たちの目標は次のとおりです。：

・学界や産業界が自由に利用できる完全に解放された ISA です。
・実際の ISA シミュレーションまたはバイナリ変換だけでなく、直接ネイティブハードウェア実装に適しています。
・ISA は特定のマイクロアーキテクチャスタイル（例えばマイクロコード化、インオーダー、デカップル、アウトオブオーダー）や実装技術（フルカスタム、ASIC、FPGA など）を「オーバーアーキテクチャー」することなく、効率的にこれらのいずれかの実装を可能にします。
・ISA は、汎用のソフトウェア開発をサポートするために、カスタマイズされたアクセラレータまたは教育目的のためのベースとして使用可能な小さな基本整数 ISA とオプションの標準拡張子に分かれています。
・改訂された 2008 IEEE-754 浮動小数点標準のサポート [14]。
・広範なユーザーレベルの ISA 拡張機能と特殊なバリアントをサポートする ISA。
↑specialized variants. ってどう訳す？ 特別な変形、変異体、変数、展開
・アプリケーション、オペレーティングシステムカーネル、およびハードウェア実装の 32 ビットおよび 64 ビットアドレス空間の両バリエーション。
・異種マルチプロセッサを含む、高度に並列なマルチコアまたは多くのコアの実装を支援する ISA。
↑multicore と manycore の違いは？ どっちも沢山のコアのように思えるが
・オプションの可変長命令は、使用可能な命令エンコーディング空間を拡張と、オプションの高密度命令エンコーディングを提供し、パフォーマンス、静的コードサイズ、およびエネルギー効率を向上させます。
・ハイパーバイザー開発を容易にする完全仮想化 ISA
・新しいスーパーバイザーレベルとハイパーバイザーレベルの ISA デザインを使用して実験を簡単にする ISA。
↑experiments 実験、試み なんだけどなんかしっくりこない

私たちのデザイン決定に関する論評(解説)は、この段落のように書式化されており、読者が仕様書そのものに興味があればスキップすることができます。

RISC-V という名前は、UC Berkeley（RISC-I [23]、RISC-II [15]、SOAR [32]、SPUR [18]の最初の 4 つ）から 5 番目の主要な RISC ISA 設計を代表するものです。
さまざまなデータ並列アクセラレータを含む一連のアーキテクチャ研究の支援が ISA 設計の明白な目標で、 "バリエーション"と "ベクトル"を表すローマ数字 "V"の使用についてもしゃれ(だじゃれ)を言います。
↑1 つの文になってるけど、2 つの文として考えた方がいいような。～明確な目標です。と ～言います。

我々は、実際のハードウェア実装の研究アイデアこの仕様書の初版以来 11 種類のシリコン製作を完了している）に特に関心のある研究と教育のニーズを支援するために RISC-V を開発しました。 (RISC-V プロセッサの RTL デザインは、Berkeley の複数の学部および大学院クラスで使用されています)。
我々の現在の研究では、従来のトランジスタスケーリングの終了によって課された電力制約によって推進される、特殊(専門)で異種なアクセラレータへの移行に特に関心があります。
私たちは、私たちの研究努力を築くための非常に柔軟で拡張性のあるベース ISA を求めました。

我々が繰り返し尋ねられた質問は、なぜ新しい ISA を開発するのかということです。 既存の商用 ISA を使用する最大の明白な利点は、研究と教育に活用できる開発ツールと移植されたアプリケーションの両方の、大規模で広くサポートされているソフトウェアエコシステムです。
↑既存の ISA はツールとアプリケーションがたくさんあるよ。ってことかな
その他の利点としては、大量の文章と例題(チュートリアル)があります。
しかし、商用の命令セットを研究と教育に使用した経験では、実際にはこれらの利点が小さく、欠点を上回らないことです。

-2018/04/05

・商用 ISA は専有です。
オープンな IEEE 標準[2]である SPARC V8 を除き、商用 ISA のほとんどの所有者は、知的財産を慎重に保護し、自由に利用可能な競争的実装を歓迎しません。
これは、ソフトウェアシミュレータのみを使用した学術研究や教育では大きな問題は少ないですが、実際の RTL 実装を共有しようとするグループにとっては大きな懸念事項でした。
また、商用 ISA 実装のいくつかのソースを信頼したくないが、独自のクリーンルーム実装を作成することは禁止されているエンティティにとって、大きな懸案事項です。
すべての RISC-V 実装に第三者の特許侵害がないことを保証することはできませんが、RISC-V 実装者を訴えるしようとしないことを保証することができます。
・商用 ISA は特定の市場分野でのみ普及しています。
執筆時点での最も明白な例は、ARM アーキテクチャがサーバ空間で十分にサポートされていないことと、インテル x86 アーキテクチャ（または他のほとんどすべてのアーキテクチャ）がモバイル領域で十分にサポートされていないことです。インテルと ARM は互いの市場区分(領域)に参入しようとしています。
もう 1 つの例は、拡張可能なコアを提供するが、埋め込み領域に焦点を当てている ARC と Tensilica です。
この市場分割は、特定の商用 ISA を実際にはソフトウェアエコシステムが特定のドメインにのみ存在し、他のユーザーのために構築しなければならないという利点を希釈します。
↑特定の所だけのソフトウェアエコシステムなので、ほかのユーザーのためにならないよ ってことかな
・商業的な ISA が出て行きます。
以前の研究基盤は、もはや普及していない（SPARC、MIPS）、あるいはもはや生産段階でない（Alpha）、商用の ISA まわりに構築されました。
これらは、活発なソフトウェア生態系の利益を失い、ISA および支援ツールに関する長引く知的財産の問題は、関心のある第三者が ISA の支援を継続する能力を妨げます。
オープン ISA は人気を失うかもしれないが、利害関係者はいずれもエコシステムの使用と開発を続けることができます。
・一般的な商用 ISA は複雑です。
主要な商用 ISAs（x86 および ARM）は、ハードウェアで一般的なソフトウェアスタックおよびオペレーティングシステムをサポートするレベルを実装するのに非常に複雑です。
さらに悪いことに、ほとんどすべての複雑さは、真に効率を改善する機能ではなく、間違った、あるいは少なくとも旧式の ISA デザインの決定によるものです。
・商用 ISA だけでは、アプリケーションを起動するには十分ではありません。
市販の ISA を実装する努力を費やしたとしても、その ISA 用に既存のアプリケーションを実行するだけでは不十分です。
ほとんどのアプリケーションでは、ユーザーレベルの ISA だけでなく、実行するための完全な ABI（アプリケーション バイナリ インターフェイス）が必要です。
ほとんどの ABI はライブラリに依存しており、オペレーティングシステムの支援に依存しています。
既存のオペレーティングシステムを実行するには、OS が期待する管理レベルの ISA とデバイスインタフェースを実装する必要があります。
これらは通常、ユーザレベルの ISA よりもはるかに明確ではなく、実装がかなり複雑です。

・一般的な商用 ISA は拡張性のために設計されていませんでした。
支配的な商用 ISA は、特に拡張性のために設計されておらず、その結果、命令セットが大きくなったため、命令エンコーディングの複雑さが増しています。
Tensilica（Cadence 社買収）や ARC（Synopsys 社買収）などの企業では、拡張性を重視して ISA とツールチェーンを構築していますが、汎用コンピューティングシステムではなく組み込みアプリケーションに注力しています。
・変更された商用 ISA は新しい ISA です。
私たちの主な目標の 1 つは、主要な ISA 拡張を含むアーキテクチャ研究をサポートすることです。
小さな拡張機能でもコンパイラを修正し、拡張機能を使用するためにソースコードからアプリケーションを再構築する必要があるため標準の ISA を使用する利点は少なくなります。
新しいアーキテクチャ状態を導入するより大きな拡張では、オペレーティングシステムを変更する必要があります。
最終的に、変更された商用 ISA は新しい ISA になりますが、ベース ISA のすべての遺産障害を引継ぎます。

私たちの立場は、ISA はおそらくコンピューティングシステムの中で最も重要なインターフェースであり、そのような重要なインターフェースが独自(専有)のものでなければならない理由はありません。
支配的な商用 ISA は、30 年以上前に既によく知られている命令セットの概念に基づいています。
ソフトウェア開発者はオープンな標準ハードウェア目標を目標とすることができ、商用プロセッサ設計者は実装品質を競うべきです。

我々はハードウェアの実装に適したオープンな ISA 設計を最初に検討しているところはありません。
↑我々は最初から遠いです。って言ってるんだけどいまいちわからん。
我々はまた、OpenRISC アーキテクチャ[22]に最も近いものとして、他の既存のオープン ISA 設計を検討しました。
私たちはいくつかの技術的な理由から OpenRISC ISA を採用することに反対しました。

・OpenRISC はより高性能な実装が複雑な条件コードと分岐遅延スロットを持っています。
・OpenRISC は、固定の 32 ビットエンコーディングと 16 ビット即値を使用しており、これにより、より高密度の命令エンコーディングが排除され、後で ISA を拡張するためのスペースが制限されます。
・OpenRISC は 2008 年の IEEE 754 浮動小数点標準の改訂をサポートしていません。
・OpenRISC 64 ビットデザインは、私たちが始めたときに完成していませんでした。

白紙の状態から始めることによって、すべての目標を達成した ISA を設計することができましたが、当初計画していたよりもはるかに労力がかかりました。
ドキュメント、コンパイラツールチェーン、オペレーティングシステムポート、参照(基準)ISA シミュレータ、FPGA 実装、効率的な ASIC 実装、アーキテクチャ試験項目群、教材など、RISC-V ISA インフラストラクチャを構築するために多大な労力を投じました。
このマニュアルの最後の版以降、学界と産業界で RISC-V ISA かなりの取り込みが行われており、我々は標準を保護し促進するために非営利の RISC-V 財団を創設しました。
RISC-V 財団ウェブサイト（http：// riscv.org）には、RISC-V を使用した財団メンバーシップと様々なオープンソースプロジェクトに関する最新情報が掲載されています。

RISC-V マニュアルは 2 つの巻で構成されています。
この巻は、オプションの ISA 拡張を含むユーザーレベルの ISA 設計について説明します。
2 番目の巻は特権アーキテクチャを提供します。

このユーザーレベルのマニュアルでは、特定のマイクロアーキテクチャー機能や特権アーキテクチャーの詳細に依存することを排除することを目指しています。
これは、明快さと、代替実装のための最大の柔軟性を可能にするためです。

-2018/04/05

1.1 RISC-V ISA の概要

RISC-V ISA は、任意の実装に存在しなければならない基本整数 ISA と、基本 ISA へのオプションの拡張として定義されます。
基本整数 ISA は、分岐遅延スロットがなく、オプションの可変長命令エンコーディングをサポートしている点を除いて、初期の RISC プロセッサのものと非常によく似ています。
ベースはコンパイラ、アセンブラ、リンカ、およびオペレーティングシステム（追加の管理者レベルの操作を含む）のための合理的なターゲットを提供するのに十分な最小限の命令セットに注意深く制限されているので、便利な ISA およびソフトウェアツールチェーン "スケルトン" よりカスタマイズされたプロセッサ ISA を構築することができます。

各基本整数命令セットは、整数レジスタの幅および対応するユーザアドレス空間のサイズによって特徴付けられます。
2 つの主要な基本整数の変形、RV32I と RV64I があり、それぞれ第 2 章と第 4 章で説明します。これらは 32 ビットまたは 64 ビットのユーザーレベルのアドレス空間を提供します。
ハードウェアの実装およびオペレーティングシステムは、ユーザープログラムに対して RV32I と RV64I の一方または両方のみを提供する場合があります。
第 3 章では、小型マイクロコントローラをサポートするために追加された RV32I ベース命令セットの RV32E サブセットの変形について説明します。
第 5 章では、将来の 128 ビットユーザアドレス空間をサポートする基本整数命令セットの RV128I 変形について説明します。
基本整数命令セットは、符号付き整数値に対して 2 の補数表現を使用します。

大規模システムでは 64 ビットのアドレス空間が必要ですが、32 ビットのアドレス空間は多くのエンベデッド・デバイスやクライアント・デバイスにとって今後も数十年の間十分なままであり、メモリ・トラフィックとエネルギー消費を削減することが望まれます。
さらに、教育目的では 32 ビットのアドレス空間で十分です。
最終的には 128 ビットのアドレス空間が必要になりますので、これを RISC-V ISA フレームワークに収めることができます。

基本整数 ISA はハードウェア実装によってサブセットになるかもしれませんが、より特権層によるオペコードトラップとソフトウェアエミュレーションは、ハードウェアによって提供されない機能を実装するために使用する必要があります。

基本整数 ISA のサブセットは教育目的には役立つかもしれませんが、ベースが逸脱しているため、実際のハードウェアの実装をサブセット化するインセンティブはほとんどなく、メモリの不整合のサポートを省略し、すべての SYSTEM 命令を単一のトラップとして扱います 。
↑ここは何をいっているのかいまいちわからん。要約(意訳)すると、基本整数 ISA は教育には役立つかも。でもこの基本はメモリのミスアライン(整列)を省略してるし、すべてのシステム命令を 1 トラップのように扱ってるので、実際のハードウェア実装のサブセットの動機(報酬)は少ないよ。 ってとこか

RISC-V は、豊富なカスタマイズと特殊化をサポートするように設計されています。
基本整数 ISA は、1 つまたは複数のオプションの命令セット拡張で拡張できますが、基本整数命令は再定義できません。
RISC-V 命令セット拡張機能を標準および非標準拡張に分割します。
標準拡張は一般的に有用であり、他の標準拡張と競合しないようにする必要があります。
非標準拡張は高度に専門化されているか、他の標準または非標準拡張と競合する可能性があります。
命令セット拡張は、基本整数命令セットの幅に応じて若干異なる機能を提供することがあります。
第 21 章では、RISC-V ISA を拡張するさまざまな方法について説明します。
また、RISC-V ベース命令と命令セット拡張の命名規則も開発しました（第 22 章で詳しく説明しています）。

より一般的なソフトウェア開発をサポートするために、整数の乗算/除算、アトミック演算、単精度浮動小数点演算と倍精度浮動小数点演算を提供する標準拡張のセットが定義されています。
基本整数 ISA は "I"（整数レジスタの幅に応じて RV32 または RV64 が前に付く）という名前で、整数計算命令、整数ロード、整数ストア、および制御フロー命令を含み、すべての RISC-V 実装で必須です。
標準の整数の乗算および除算の拡張は「M」と命名され、整数レジスタに保持された値を乗算および除算するための命令を追加します。
「A」で示される標準的な原子命令拡張は、プロセッサ間同期のためにメモリを原子的に読み取り、修正し、書き込む命令を追加する。
"F"で示される標準単精度浮動小数点拡張は、浮動小数点レジスタ、単精度計算命令、および単精度ロードおよびストアを追加します。
"D"で示される標準倍精度浮動小数点拡張は、浮動小数点レジスタを拡張し、倍精度の計算命令、ロード、およびストアを追加します。
整数ベースに加えて、これら 4 つの標準拡張（「IMAFD」）には、略語「G」が与えられ、汎用スカラ命令セットが提供される。
現在、RV32G と RV64G はコンパイラツールチェーンのデフォルトターゲットです。
後の章では、これらおよびその他の計画された標準的な RISC-V 拡張について説明します。
↑アトミックとアトミカリってどう訳すべきか。ARM AMBA AXI 規格に アトミック転送ってあったような。あれと同じ？

ベース整数 ISA および標準の拡張を超えて、新しい命令がすべてのアプリケーションに大きな利点をもたらすことはまれですが、特定のドメインにとっては非常に有益です。
エネルギー効率の懸念により、より専門化が強まっているので、ISA 仕様の必要部分を簡素化することが重要であると考えています。
他のアーキテクチャでは通常、ISA を単一のエンティティとして扱いますが、命令が時間の経過と共に追加されるにつれて新しいバージョンに変更されますが、RISC-V はベースと各標準拡張を時間の経過とともに一定に保ち、 その代わりにさらにオプションの拡張として新しい命令を階層化します。
たとえば、基本整数 ISAs は、後続の拡張機能に関係なく、完全に支持されている独立型 ISAs として引き続き使用さ続けます。

ユーザー ISA 仕様の 2.0 リリースでは、将来の開発のために、 "RV32IMAFD"と "RV64IMAFD"ベースと標準拡張（別名 "RV32G"と "RV64G"）を一定に保つ予定です。

1.2 命令長エンコーディング

ベース RISC-V ISA は、固定長の 32 ビット命令を備えています。これらの命令は、32 ビット境界で自然に整列する必要があります。
しかし、標準の RISC-V エンコーディング方式は、可変長命令を持つ ISA 拡張をサポートするように設計されています。各命令は任意の数の 16 ビット命令区切りになり、16 ビット境界で自然に整列されます。
第 12 章で説明した標準の圧縮 ISA 拡張命令は、圧縮された 16 ビット命令を提供することでコードサイズを縮小し、すべての命令（16 ビットと 32 ビット）を 16 ビット境界で整列させてコード密度を向上させるための整列制約を緩和します。

図 1.1 に、標準的な RISC-V 命令長エンコード規則を示します。
ベース ISA のすべての 32 ビット命令は、最下位の 2 ビットが 11 に設定されています。
オプションの圧縮された 16 ビット命令セット拡張は、00,01、または 10 に等しい最低 2 ビットを有しています。
32 ビット以上でエンコードされた標準命令セット拡張では、図 1.1 に示す 48 ビットおよび 64 ビットの長さ規則に従って、下位ビットが 1 に設定され追加されます。
80 ビットと 176 ビットの間の命令長は、最初の 5x16 ビットワードに加えて、16 ビットワードの数を与えるビット [14:12] の 3 ビットフィールドを使用して符号化されます。
ビット[14:12]が 111 にセットされた符号化は、将来のより長い命令符号化のために予約されています。

圧縮されたフォーマットのコードサイズと省エネルギーを考えると、これを補足として追加するのではなく、ISA 符号化方式の圧縮フォーマットをサポートしたいと考えましたが、より簡単な実装を可能にするために、必須です。
また、実験とより大きな命令セット拡張をサポートするために、より長い命令をオプションで許可したいと考えました。
私たちの符号化規約では、コア RISC-V ISA の符号化がより厳密に要求されていましたが、これにはいくつかの有益な効果があります。
標準 G ISA の実装では、命令キャッシュに最上位 30 ビットしか保持する必要がありません（6.25%の節約）。
命令キャッシュリフィルでは、不正な命令例外の動作を保持するためにキャッシュに格納する前に、ロービットクリアのいずれかの命令が不正 30bit 命令に記録される必要があります。
↑ここは何を言っているのかよくわからない。例外が起こった時には 30bit は保持してねということかな。？？？
おそらくもっと重要なのは、基本 ISA を 32 ビット命令語のサブセットに集約することによって、カスタム拡張で使用可能な空間が増えたことです。
第 21 章で説明したように、標準の圧縮命令拡張機能のサポートを必要としない実装では、≧32 ビット命令セット拡張を超える標準のサポートを維持しながら、3 つの 30 ビット命令スペースを 32 ビットの固定幅フォーマットにマップできます。さらに、実装はまた、長さが>32 ビットを超える命令を必要としない場合、それはさらに 4 つの主要なオペコードを回復することができます。

図 1.1 RISC-V 命令長符号化
↑要するに、命令コードは 16bit ずつの区切りで、下位ビットから見ていくと何ビットの命令かわかるよ、ってこと

すべてのゼロビットを含む任意の長さの命令が正当でないという特徴は、ゼロメモリ領域への誤ったジャンプを素早くトラップするためです。
同様に、プログラムされていない不揮発性メモリデバイス、切断されたメモリバス、または壊れたメモリデバイスで観察される他の一般的パターンを補足するために、すべてのものを含む命令符号化は不正命令として確保します。

ベース RISC-V ISA にはリトルエンディアンのメモリシステムがありますが、非標準のバリアントではビッグエンディアンまたはバイエンディアンのメモリシステムを提供できます。
命令はメモリに格納され、各 16 ビット区画は、実装の自然なエンディアンに従ってメモリハーフワードに格納されます。
1 つの命令を形成する区画は、命令仕様の最も低い番号のビットを保持する最下位の区画を有します、すなわちメモリシステムのエンディアンに関係なく小規模の区画配列に常に格納されます。
図 1.2 のコードシーケンスは、メモリシステムのエンディアンに関係なく、32 ビット命令をメモリに正しく格納します。

リトルエンディアンシステムが現在商業的に支配的しているため、我々は RISC-V メモリシステムのためにリトルエンディアンバイトオーダリングを選択しました（すべての x86 システム、iOS、Android、Windows for ARM）。
目立たない点は、我々はハードウェア設計者にとっては、リトルエンディアンのメモリシステムがより自然なものであることがわかったということです。
しかし、IP ネットワーキングなどの特定のアプリケーション分野は、ビッグエンディアンのデータ構造で動作するため、非標準のビッグエンディアンまたはバイエンディアンの可能性を残しています。
長さエンコーディングビットが必ず最初にハーフワードアドレス順に現れるようにするために、メモリシステムエンディアンとは独立して命令区画がメモリに格納される順序を修正する必要があります。
これにより、最初の 16 ビット命令区画の最初の数ビットのみを調べることによって、命令フェッチユニットによって可変長命令の長さを迅速に決定することが可能になります。
リトルエンディアンのメモリシステムと命令区画の順序付けを修正した後は、命令形式の LSB 位置に長さエンコードビットを配置し、オペコードフィールドの分割を避けることができるようになりました。

```
// x2 レジスタの 32 ビット命令を x3 が指し示す位置に格納します。
sh  x2, 0(x3)   // 命令の下位ビットを第 1 区画に格納します。
srli x2, x2, 16  // 上位ビットを下位ビットに移動し、x2 を上書きします。
sh  x2, 2(x3)   // 上位ビットを 2 番目の区画に格納します。
```

図 1.2： レジスタからメモリへの 32 ビット命令を格納する推奨コードシーケンス。
ビッグエンディアンとリトルエンディアンの両方のメモリシステムで正しく動作し、可変長命令セット拡張で使用される場合の誤整列アクセスを回避します。

1.3 例外、トラップ、および割り込み

現在の RISC-V スレッドの命令に関連付けられた実行時に発生する異常な状態を参照するために、例外という用語を使用します。
トラップという用語は、RISC-V スレッド内で発生する例外的条件によって引き起こされるトラップハンドラへの制御の同期転送を参照するために使用します。
トラップハンドラは通常より特権のある環境で実行されます。

現在の RISC-V スレッドとは非同期に発生する外部イベントを参照するために、割り込みという用語を使用します。
処理されなければならない割込みが発生すると、割込み例外を受け取るためにいくつかの命令が選択され、続いてトラップが発生します。

次章の命令説明では、実行中に例外が発生する条件について説明しています。
これらがトラップに変換されるかどうかは実行環境に依存しますが、例外が通知されたときにはほとんどの環境で正確なトラップが実行されることが予想されます (ただし、浮動小数点例外を除き、標準浮動小数点数の拡張は、トラップを引き起こしません)。

私たちの "例外"と "トラップ"の使用は、IEEE-754 浮動小数点標準と一致します。

-2018/04/14

1ページ空き。次ページへ。

第 2 章
RV32I ベース整数命令セット、
バージョン 2.0

この章では、RV32I 基本整数命令セットのバージョン 2.0 について説明します。
解説の多くは RV64I 変形にも当てはまります。

RV32I は、コンパイラターゲットを形成し、最新のオペレーティングシステム環境をサポートするのに十分なものになるように設計されています。
ISA は最小限の実装で必要なハードウェアを削減するように設計されています。
RV32I には 47 個のユニークな命令が含まれていますが、シンプルな実装では、8 個の SCALL / SBREAK / CSRR *命令を常に 1 つの SYSTEM ハードウェア命令でカバーすることができますが、FENCE および FENCE.I 命令を NOP として、 ハードウェア命令数を合計 38 に減らします。
↑いまいちよくわからんが、RV32I は 47 命令あるけど、おまとめすると 38 命令に減るよ。っていってるんだろうか
RV32I はほとんどの他の ISA 拡張をエミュレートすることができます（アトミック性を追加するハードウェアサポートが必要な A 拡張機能を除く）。

2.1 基本整数サブセットのプログラマモデル

図 2.1 に、基本整数サブセットのユーザーが表示できる状態を示します。
整数値を保持する 31 個の汎用レジスタ x1～x31 があります。
レジスタ x0 は定数 0 にハードワイヤードされています。
ハードワイヤードのサブルーチンリターンアドレスリンクレジスタはありませんが、標準ソフトウェア呼び出し規約ではレジスタ x1 を使用して呼び出し時にリターンアドレスを保持します。
RV32 では、x レジスタは 32 ビット幅で、RV64 では 64 ビット幅です。
このドキュメントでは、XLEN という用語を使用して、x レジスタの現在の幅（32 または 64 のいずれか）を参照します。

追加のユーザ可視レジスタが 1 つあります。：プログラムカウンタ pc は、現在の命令のアドレスを保持します。

-2018/04/17

使用可能なアーキテクチャレジスタの数は、コードサイズ、パフォーマンス、およびエネルギー消費に大きな影響を与える可能性があります。
コンパイルされたコードを実行する整数 ISA には 16 個のレジスタで十分ですが、3 アドレス形式を使用して 16 ビットの命令で 16 個のレジスタで完全な ISA をエンコードすることは不可能です。
2 アドレス形式も可能ですが、命令数が増えてそれと効率が低下します。
基本的なハードウェアの実装を簡素化するために中間命令サイズ（Xtensa の 24 ビット命令など）を避け、32 ビット命令サイズが採用されれば 32 の整数レジスタを維持するのは簡単でした。
整数レジスタの数が多いほど、ループ展開、ソフトウェアパイプライン、およびキャッシュタイリングを幅広く使用できる高性能コードのパフォーマンスも向上します。
↑キャッシュタイリングってなんだろ。キャッシュ領域を埋めるっていう意味かな？
これらの理由から、我々は基本 ISA 用のために 32 の整数レジスタを従来のサイズで選択しました。
動的レジスタの使用量は、頻繁にアクセスされるいくつかのレジスタによって支配される傾向があり、regfile の実装は頻繁にアクセスされるレジスタのアクセスエネルギーを削減するために最適化することができます[31]。
オプションで圧縮された 16 ビット命令フォーマットは主に 8 つのレジスタにしかアクセスせず、したがって高密度命令エンコーディングを提供することができ、追加の命令セット拡張は必要に応じて非常に大きなレジスタ空間（平坦かもしくは階層的に）をサポートすることができます。
資源に制約のある組み込みアプリケーションの場合、RV32E サブセットを定義しました。このサブセットには 16 個のレジスタしかありません（第 3 章）。

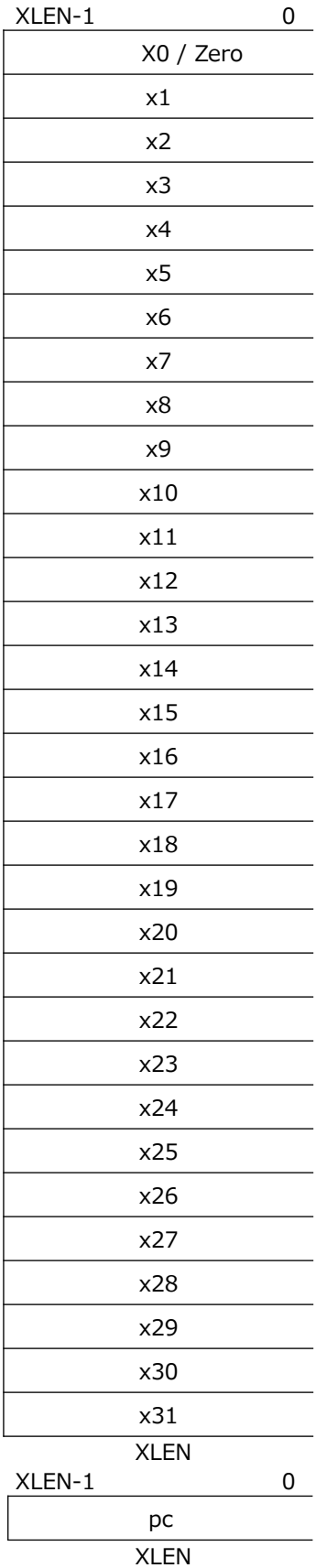


図 2.1：RISC-V ユーザーレベルの基本整数レジスタの状態

-2018/04/18

2.2 基本命令フォーマット

ベース ISA には、図 2.2 に示すように、4 つのコア命令フォーマット（R / I / S / U）があります。
すべて固定長 32 ビットであり、メモリ内の 4 バイト境界で整列しなければなりません。
もし対象アドレスが 4 バイト整列でない場合、命令不一致例外が、取得された分岐または無条件ジャンプで生成されます。
実行されない条件分岐に対して、命令フェッチ・不整列例外が生成されません。
↑この訳はなんか変。最初の No がどこにかかってくるか。

---

16 ビット長または 16 ビット長の他の奇数倍の命令拡張が追加されると、基本 ISA 命令のアラインメント制約が 2 バイト境界に緩和されます。

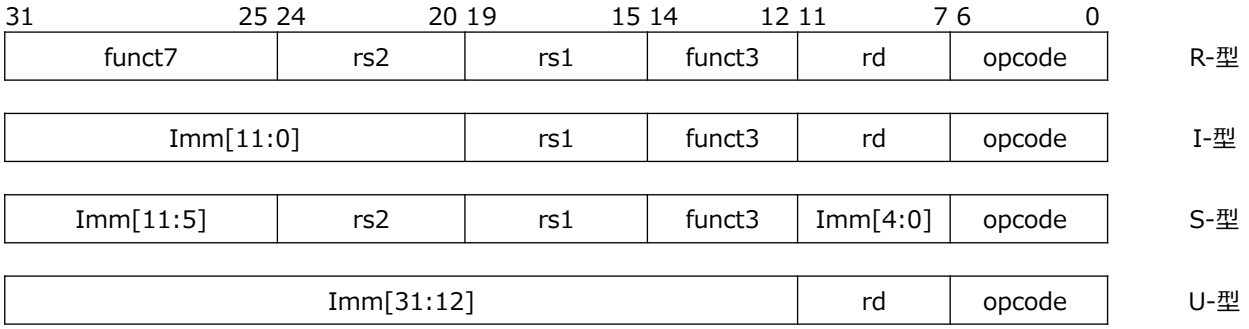


図 2.2 : RISC-V の基本命令フォーマット
各即値サブフィールドは、通常行われる命令の即値フィールド内のビット位置ではなく、生成される即値のビット位置（imm [x]）でラベル付けされます。

-2018/04/19

RISC-V ISA はソース（rs1 と rs2）とデスティネーション（rd）レジスタをすべてのフォーマットの同じ位置に保ち、デコードを簡単にします。
CSR 命令（セクション 2.8）で使用されている 5 ビット即値を除き、即値は常に符号拡張されており、一般に命令の最も左側の利用可能なビットに向かって詰め込まれ、ハードウェアの複雑さを軽減するために割り当てられています。
特に、すべての直の符号ビットは、符号拡張回路を高速化するため常に命令のビット 31 にあります。

レジスタ指定子のデコードは、通常、実装のクリティカルパス上にあります。したがって、すべてのレジスタ指定子をすべてのフォーマットの同じ位置に保持するように命令フォーマットが選択されました（RISC-IV と共有されるプロパティ 別名 SPUR [18]）。
実際には、ほとんどの即値は小さいか、すべての XLEN ビットが必要です(必要とします)。
我々は、通常の命令で利用可能なオペコード空間を増加させるために、非対称即時分割（通常の命令では 12 ビットと 20 ビットの特別なロード上の即値命令）を選択した。
↑(規則正しい命令 12bit に加え 20bit の特別なロード上位即値命令) なのかな
即値は、MIPS ISA のようにいくつかの即値に対してゼロ拡張を使用する利点を観察せず、ISA をできるだけ単純なものに保ちたいという理由で、符号拡張されています。
↑値はゼロ拡張はあまり良くなかったし、単純にしたかったので符号拡張にしたよ。

2.3 即時符号化変形
!即値のエンコード種類 の方がいいかな

図 2.3 に示すように、即値の処理に基づいた命令フォーマット（B / J）のさらに 2 つの変形があります。

S フォーマットと B フォーマットとの唯一の違いは、12 ビットの即値フィールドを使用して、B フォーマットの 2 の倍数で分岐オフセットを符号化するために使用されることです。
命令符号化された命令内のすべてのビットを、従来のようにハードウェアで 1 つ左にシフトする代わりに、中間ビット（imm [10：1]）および符号ビットは固定位置に留まり、S フォーマットの最下位ビット（inst [7]）は、B フォーマットの上位ビットを符号化します。

同様に、U フォーマットと J フォーマットとの唯一の違いは、20 ビット即値が 12 ビット左にシフトして U 即値を形成し、1 ビットが J 即値を形成することです。
U および J フォーマット即値における命令ビットの位置は、他のフォーマットと互いに重なり(共通部分)を最大にするように選択されます。

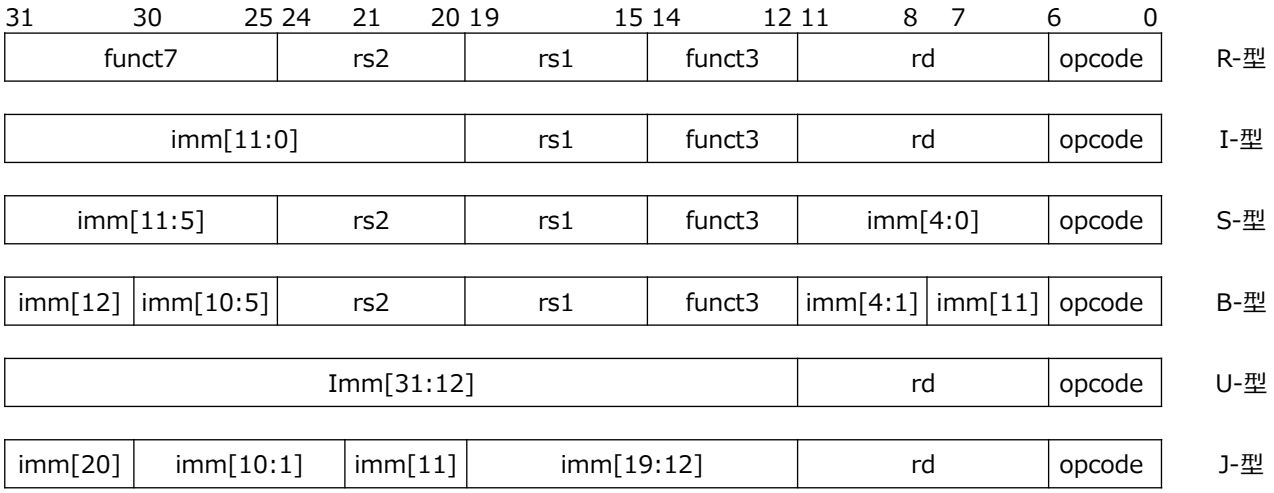


図 2.3: 即値変形を示す RISC V の基本命令形式。
↑variants は種類とかに訳した方がいいかな。即値の種類。しかし J-type とかなんちゅう並びだ。

図 2.4 は、各基本命令フォーマットによって生成された即値を示し、どの命令ビット（inst [y]）が即値の各ビットを生成するかを示すためにラベル付けされています。

-2018/04/20

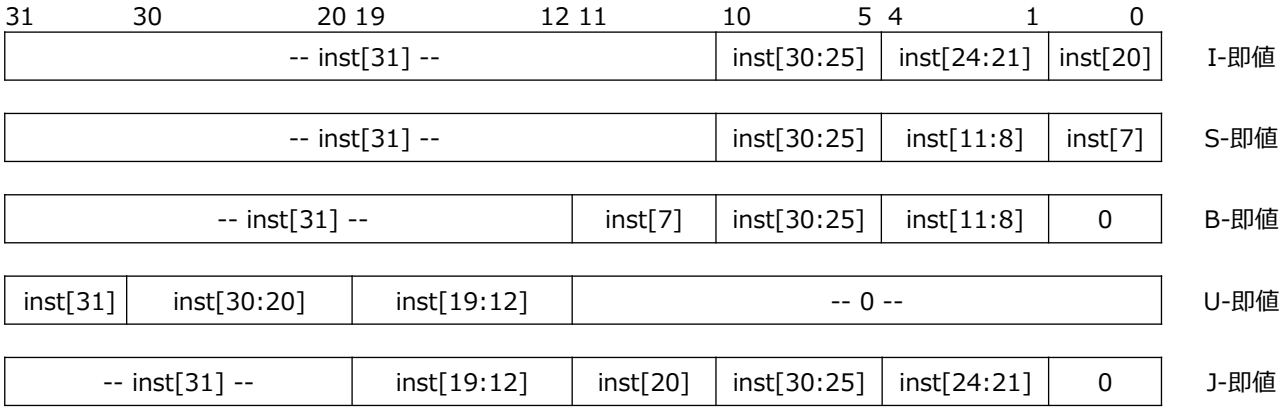


図 2.4：RISC-V 命令で生成される即値の型
フィールドは、値を構成するために使用される命令ビットでラベル付けされています。
符号拡張は常に inst [31]を使用します。

符号拡張は、即値（特に RV64I）で最も重要な操作の 1 つであり、RISC-V では、すべての即値の符号ビットが常に命令のビット 31 に保持され、符号拡張が命令デコードと並行して進められます。
より複雑な実装では、分岐とジャンプの計算に別々の加算器が存在する可能性があるため、即値ビットの位置を命令のタイプ間で一定に保つことで恩恵を受けることはありませんが、我々は最も簡単な実装のハードウェアコストを削減する必要がありました。
動的ハードウェアマルチプレクサを使用する代わりに B および J 即値の命令エンコーディングでビットを回転(ローテート)することにより、即値に 2 を乗算することにより、命令信号ファンアウトおよび即時マルチプレクサコストを約 2 倍に削減します。
スクランブルされた即時エンコードでは、静的または事前コンパイル時に無視できる時間が追加されます。
命令を動的に生成するためには、若干の付加的なオーバーヘッドがありますが、最も一般的な短い前方分岐は簡単な即時エンコーディングを持っています。

- 2018/04/22

2.4 整数計算命令

ほとんどの整数計算命令は、整数レジスタファイルに保持された値の XLEN ビットで動作します。
整数計算命令は、I 型フォーマットを使用するレジスタ即値演算として、または R 型フォーマットを使用するレジスタ-レジスタ演算として符号化されます。
宛先はレジスタ即値命令とレジスタ-レジスタ命令の両方に対するレジスタ「rd」です。
整数演算命令では算術例外は発生しません。
↑destination は”rd”、算術例外なし

RISC-V 分岐を使用して多くのオーバーフローチェックを安価に実装できるようにするため、基本命令セットの整数算術演算のオーバーフローチェックに特別な命令セットサポートは含まれていませんでした。
符号なし加算のオーバーフローチェックでは、加算後に 1 つの追加の分岐命令しか必要としません。add t0、t1、t2; bltu t0、t1、オーバフロー。
符号付き加算では、1 つのオペランドの符号が分かっている場合、加算後に 1 つの分岐のみが必要になります。addi t0、t1、+imm; blt t0、t1、オーバフロー。
これは、即値オペランドを用いた加算の一般的なケースをカバーしています。
一般的な符号付き加算については、加算後に 3 つの追加命令が必要であり、他方のオペランドが負である場合にのみ、その和がオペランドの 1 つよりも小さくなければならないという考えを利用します。

```
add t0, t1, t2
slti t3, t2, 0
slt t4, t0, t1
bne t3, t4, overflow
```

RV64 では、オペランドの ADD と ADDW の結果を比較することで、32 ビット符号付き加算のチェックをさらに最適化できます。

整数レジスタ – 即時命令

31 20	19 15	14 12	11 7	6 0
Imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-即値[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-即値[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

ADDI は、符号拡張された 12 ビットの即値をレジスタ rs1 に加算します。
算術オーバーフローは無視され、結果は単に結果の低い XLEN ビットになります。
ADDR rd、rs1,0 は、MV rd、rs1 アセンブラ疑似命令の実装に使用されます。

レジスタ rs1 が符号拡張された即値よりも小さい場合、両方が符号付き数値として扱われる場合、SLTI（即時より小さい設定）はレジスタ rd に値 1 を置きます。そうでない場合は rd に 0 が書き込まれます。
SLTIU は類似していますが、その値を符号なし数として比較します（すなわち、即値は最初に符号拡張されて XLEN ビットに変換され、次に符号なし数として扱われます）。
注：SLTIU rd、rs1,1 は、rs1 がゼロの場合は rd を 1 に設定し、そうでない場合は rd を 0 に設定します（アセンブラ疑似命令 SEQZ rd、rs）。

ANDI、ORI、XORI は、レジスタ rs1 と符号拡張 12 ビットの即値をビット単位で AND、OR、XOR し、その結果を rd に格納する論理演算です。
注：XORI rd、rs1、-1 は、レジスタ rs1 のビット単位の論理反転を実行します（アセンブラ疑似命令 NOT rd、rs）。

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

定数によるシフトは、I 型形式の特殊化としてエンコードされます。
シフトされるオペランドは rs1 であり、シフト量は I-即値フィールドの下位 5 ビットにエンコードされます。
右シフト型は、I 即値の上位ビットで符号化されます。
SLLI は論理左シフトです（ゼロは下位ビットにシフトされます）。 SRLI は論理右シフトです（0 は上位ビットにシフトされます）。 SRAI は算術右シフトです（元の符号ビットは空いている上位ビットにコピーされます）。

31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-即値[31:12]	dest	LUI
U-即値[31:12]	dest	AUIPC

LUI（上位即値ロード）は、32 ビット定数を作成するために使用され、U 型形式を使用します。
LUI は U-即値を転送先レジスタ rd の上位 20 ビットに配置し、下位 12 ビットをゼロで埋めます。

AUIPC（PC に上位即値を追加）は、PC 相対アドレスを作成するために使用され、U 型形式を使用します。
AUIPC は、20 ビットの U-即値から 32 ビットのオフセットを作成し、最下位の 12 ビットを 0 で埋め込み、このオフセットを pc に加え、結果をレジスタ rd に配置します。

AUIPC 命令は、制御フロー転送とデータアクセスの両方に対して PC からの任意のオフセットにアクセスするための 2 命令シーケンスをサポートしています。
AUIPC と JALR の 12 ビットイミディエイトを組み合わせることで、任意の 32 ビット PC 相対アドレスに制御を移すことができ、AUIPC に加えて通常のロード命令またはストア命令の 12 ビット即値オフセットは 32 ビット PC 相対データアドレス にアクセスできます。
現在の PC は、U-即値を 0 に設定することで取得できます。
PC を取得するために JAL +4 命令を使用することもできますが、より単純なマイクロアーキテクチャではパイプラインが切断されるか、より複雑なマイクロアーキテクチャでは BTB 構造が汚染される可能性があります。

整数レジスタ – レジスタ演算

RV32I は、いくつかの算術 R タイプ演算を定義します。 すべてのオペレーションは、rs1 と rs2 レジスタをソースオペランドとして読み込み、結果をレジスタ rd に書き込みます。
funct7 フィールドと funct3 フィールドは、操作の種類を選択します。

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

ADD と SUB はそれぞれ加算と減算を行います。
オーバーフローは無視され、結果の低 XLEN ビットが出力先に書き込まれます。
SLT と SLTU は、符号付きと符号なしの比較をそれぞれ実行し、rs1 <rs2 の場合は rd に 1 を、それ以外の場合は 0 を書き込みます。
SLTU rd、x0、rs2 は、rs2 がゼロでない場合には rd を 1 に設定し、そうでない場合は rd をゼロに設定します（アセンブラ疑似演算 SNEZ rd、rs）。
AND、OR、および XOR はビット単位の論理演算を実行します。

NOP 命令

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	ADDI	0	OP-IMM

NOP 命令は、PC を進めることを除いて、ユーザーが見ることのできる状態を変更しません。
NOP は ADDI x0、x0、0 としてエンコードされます。

NOPs を使用すると、コードセグメントをマイクロアーキテクチャ上重要なアドレス境界に合わせたり、インラインコードを変更するための領域を確保したりすることができます。
NOP をエンコードするには多くの方法がありますが、わかりやすい逆アセンブリ出力と同様に、マイクロアーキテクチャの最適化を可能にする標準 NOP エンコーディングを定義しています。

-- 5 月連休明けから常駐のお仕事が決まったためバタバタしてこの 2 週間翻訳に手を付けられませんでした。(x_x)
-- 2018/05/03

2.5 コントロール転送命令

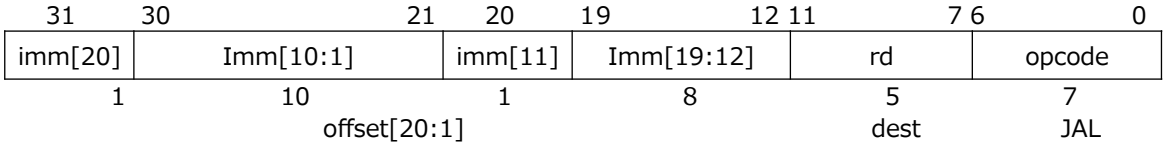
RV32I には、無条件ジャンプと条件分岐の 2 種類の制御転送命令があります。
RV32I の制御転送命令には、アーキテクチャ上見える遅延スロットはありません。

無条件ジャンプ

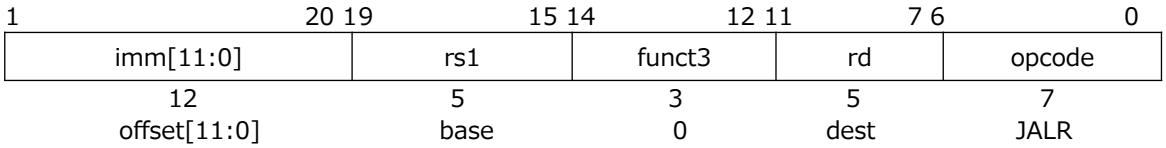
ジャンプおよびリンク（JAL）命令は J 型形式を使用し、J 即時は 2 バイトの倍数で符号付きオフセットを符号化します。
オフセットは符号拡張され、ジャンプターゲットアドレスを形成するために PC に追加されます。
したがって、ジャンプは ±1 MiB 範囲を対象にすることができます。
JAL は、ジャンプ（pc + 4）に続く命令のアドレスをレジスタ rd に格納します。
標準ソフトウェア呼び出し規約では、x1 をリターンアドレスレジスタとして使用し、x5 を代替リンクレジスタとして使用します。

代替リンクレジスタは、通常のリターンアドレスレジスタを保持しながら、ミリコードルーチンを呼び出し（例えば、圧縮コードでレジスタを保存し復元する場合など）をサポートします。
レジスタ x5 は、標準の呼び出し規約で一時的にマップされるように代替リンクレジスタとして選ばれ、通常のリンクレジスタとは異なる 1 ビットのエンコーディングを持っています。

プレーン無条件ジャンプ（アセンブラ疑似オペレーション J）は、rd = x0 の JAL としてエンコードされます。



間接ジャンプ命令 JALR（ジャンプおよびリンクレジスタ）は、I 型エンコーディングを使用します。
ターゲットアドレスは、レジスタ rs1 に 12 ビット符号付き I 即値を加算し、結果の最下位ビットをゼロに設定することによって得られます。
ジャンプ後の命令のアドレス（pc + 4）がレジスタ rd に書き込まれます。
結果が必要ない場合は、レジスタ x0 を宛先として使用できます。



無条件ジャンプ命令はすべて、PC 相対アドレッシングを使用して独立した位置コードをサポートします。
JALR 命令は、2 命令シーケンスが 32 ビット絶対アドレス範囲のどこにでもジャンプできるように定義されています。
LUI 命令は、まずターゲットアドレスの上位 20 ビットで rs1 をロードし、次に JALR は下位ビットを加算することができます。
同様に、AUIPC そして JALR は、32 ビットの PC 相対アドレス範囲のどこにでもジャンプすることができます。
JALR 命令は、条件付き分岐命令とは異なり、12 ビットの即値を 2 バイトの倍数として扱わないことに注意してください。
これは、ハードウェアで 1 つのより即値フォーマットを避けます。
↑これはハードウェアでもう 1 つの即値フォーマットを避けます。ハードウェアで区別できるようになってるってことか？
実際には、JALR のほとんどの用途は即時にゼロになるか、または LUI または AUIPC とペアになるため、範囲のわずかな縮小は重要ではありません。
JALR 命令は、計算されたターゲットアドレスの最下位ビットを無視します。
これは、ハードウェアをわずかに簡略化し、関数ポインタの下位ビットを使用して補助情報を格納することを可能にします。
この場合、エラーチェックのわずかな損失が発生する可能性がありますが、実際には誤った命令アドレスにジャンプすると、通常はすぐに例外が発生します
ベース rs1 = x0 で使用すると、JALR を使用して、アドレス空間のどこからでも最低 2KiB または最高 2KiB のアドレス領域への単一命令サブルーチン呼び出しを実装できます。これは、小さなランタイムライブラリへの高速呼び出しを実装するために使用できます。

-- 2018/05/03

JAL 命令と JALR 命令は、ターゲットアドレスが 4 バイトの境界に揃っていないと、命令フェッチ例外が誤って生成されます。

命令フェッチのミスアライン例外は、圧縮命令セット拡張 C のような 16 ビット整列命令を持つ拡張をサポートするマシンでは不可能です。

リターンアドレス予測スタックは、高性能命令フェッチユニットの共通の機能ですが、プロシージャコールおよびリターンに使用される命令を正確に検出する必要があります。
RISC-V の場合、命令の使用に関するヒントは、使用されるレジスタ番号を介して暗黙的に符号化されます。
JAL 命令は、rd = x1 / x5 の場合にのみリターンアドレスをリターンアドレススタック（RAS）にプッシュする必要があります。
JALR 命令は、表 2.1 に示すように RAS をプッシュ/ポップする必要があります。

rd	rs1	rs1=rd	RAS action
!link	!link	-	none
!link	link	-	pop
link	!link	-	push
link	link	0	push and pop
link	link	1	push

表 2.1：命令で使用されるレジスタ指定子にエンコードされたリターンアドレススタック予測ヒント。
上記では、レジスタが x1 または x5 の場合にリンクが真になります。

他のいくつかの ISA はリターンアドレススタック操作を導くために間接ジャンプ命令に明示的なヒントビットを追加しました。
これらのヒントに使用されるエンコード領域を減らすために、レジスタ番号と呼び出し規約に結びついている暗黙のヒントを使用します。
2 つの異なるリンクレジスタ（x1 と x5）が rs1 と rd として与えられると、RAS はコルーチンをサポートするために両方プッシュされ、ポップされます。
rs1 と rd が同じリンクレジスタ（x1 または x5 のいずれか）である場合、RAS はプッシュされ、シーケンスのマクロオペレーションの融合が可能になります。
lui ra、imm20; jalr ra、ra、imm12 および auipc ra、imm20; jalr ra、ra、imm12

条件付き分岐

すべての分岐命令は、B 型命令形式を使用します。
12 ビットの B-即値は符号付きオフセットを 2 の倍数で符号化し、現在の pc に加算してターゲットアドレスを与えます。
条件分岐範囲は±4 KiB です。

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	immm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12,10:5]		src2	src1	BEQ/BNE	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BLT[U]	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BGE[U]	offset[11,4:1]		BRANCH

分岐命令は 2 つのレジスタを比較します。
BEQ と BNE は、レジスタ rs1 と rs2 がそれぞれ等しいか等しくなければ、分岐を取ります。
BLT と BLTU は、符号付きと符号なしの比較をそれぞれ使用して、rs1 が rs2 より小さい場合に分岐を取ります。
BGE と BGEU は、符号付きと符号なしの比較をそれぞれ使用して、rs1 が rs2 以上の場合は分岐を取ります。
なお、BGT、BGTU、BLE、および BLEU は、それぞれオペランドを BLT、BLTU、BGE、および BGEU に逆にすることで合成できます。
↑take は取ります だけど、 分岐します と訳してもよいかもしれない。

符号付きの配列境界は、単一の BLTU 命令でチェックできますが、いずれかの負のインデックスは非負の境界よりも大きく比較されるからです。

ソフトウェアは、シーケンシャルコードパスが最も一般的なパスであり、頻度の低いコードパスが行の外に配置されるように、最適化する必要があります。
ソフトウェアは、少なくとも最初にそれらが遭遇したとき、後方ブランチが取られると予測し、前方ブランチが取られないと予測することも仮定しなければならない。
動的予測は、予測可能な分岐動作をすばやく学習する必要があります。

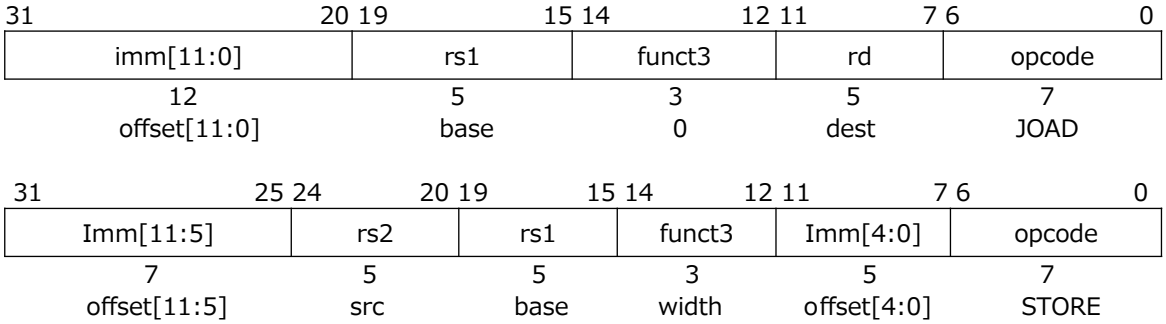
他のいくつかのアーキテクチャとは異なり、RISC-V ジャンプ（rd = x0 の JAL）命令は、常に真の条件の条件付き分岐命令の代わりに常に無条件分岐に対して使用する必要があります。
RISC-V ジャンプは PC 相対でもあり、分岐よりもずっと広いオフセット範囲をサポートし、条件分岐予測テーブルを圧迫することはありません。

条件分岐は、2 つのレジスタ間の算術比較演算を含むように設計されており（PA-RISC と Xtensa ISA でも同様です）むしろ、条件コード（x86、ARM、SPARC、PowerPC）、または 1 つのレジスタをゼロ（Alpha、MIPS）と比較するだけ、または等価のみの 2 つのレジスタ（MIPS）を使用します。
この設計は、比較と分岐命令を組み合わせたものが通常のパイプラインに適合し、追加の条件コード状態または一時レジスタの使用を回避し、静的コード・サイズおよび動的命令フェッチ・トラフィックを削減するという観察によって動機付けられました。
もう 1 つのポイントは、ゼロとの比較では、（特に高度なプロセスでは静的ロジックに移行した後に）些細ではない回路の遅延必要なため、算術規模の比較と同じくらい高価です。
↑ゼロとの比較の方が簡単なような気がするが、ここではコストかかるよ～みたいになっている。ハテ？
融合された比較および分岐命令の別の利点は、分岐がフロントエンド命令ストリームの早期に観察され、より早期に予測できることが可能です。
同じ条件コードに基づいて複数の分岐を取ることができる場合には、条件コードを含む設計には多分利点がありますが、このケースは比較的まれであると考えられます。
我々は、命令符号化に静的分岐ヒントが含まれているか考慮したが含まれていませんでした。
これにより、動的予測への負担を軽減することができますが、最適な結果を得るためには、より多くの命令エンコード領域とソフトウェアプロファイリングが必要になり、実動実行がプロファイリング実行と一致しないとパフォーマンスが低下する可能性があります。↑分岐予測ミスるとパフォーマンス低下するよ
我々は、条件付き移動または予測命令が含まれないか考慮しました、予期しない短い前方分岐を効果的に置き換えることができました。
条件付き移動は 2 つの方が簡単ですが、例外（メモリアクセスと浮動小数点演算）を引き起こす条件付きコードでは使用が困難です。
条件付き実行制御は、システムに追加のフラグ状態、フラグの設定とクリアのための追加命令、およびすべての命令の追加の符号化オーバーヘッドを追加します。
条件付き移動命令と述語命令の両方が、述語が偽であれば、デスティネーションアーキテクチャレジスタの元の値を名前変更されたデスティネーション物理レジスタにコピーする必要があるため、暗黙的な第 3 のソースオペランドを追加して、順序外のマイクロアーキテクチャに複雑さを追加します。↑predicate は述語だけどどう訳すとそれらしくなるか？
また、分岐の代わりに述語を使用する静的なコンパイル時の決定は、特に予期しない分岐がまれであることを考えると、コンパイラのトレーニングセットに含まれない入力でパフォーマンスが低下し、分岐予測技術が向上するにつれて希少になります。
我々は、予測不能な短い順方向分岐を内部予測されたコードに動的に変換して、分岐予測ミス予測時にパイプラインをフラッシュするコストを回避するため商用プロセッサに実装されている[27]様々なマイクロアーキテクチャ技術が存在することを認識している[13,17,16]。
最も簡単な手法は、フェッチパイプライン全体ではなくブランチシャドウ内の命令をフラッシュするだけで、またはワイド命令フェッチまたはアイドル命令フェッチスロットを使用して両側から命令をフェッチすることによって、誤予測された短い前方分岐から回復するペナルティを軽減するだけです。
アウトオブオーダーコアのより複雑な技法は、分岐述部によって内部述部値が書き込まれた状態で、分岐シャドウの命令に内部述部を追加し、分岐および後続命令を他のコードに対して推論的かつ順不同で実行できるようにします[27]。

2.6 ロードとストア命令

RV32I は、ロードストア命令のみがメモリにアクセスし、算術命令が CPU レジスタ上でのみ動作するロードストアアーキテクチャです。
RV32I は、バイトアドレスとリトルエンディアンの 32 ビットのユーザーアドレス空間を提供します。
実行環境は、アドレス空間のどの部分がアクセスに正当であるかを定義する。
x0 の宛先を持つロードは、ロード値が破棄されても、例外を発生させ、他の副作用を引き起こす必要があります。

-- 2018/05/05



ロード命令とストア命令は、レジスタとメモリの間に値を転送します。
ロードはI型形式でエンコードされ、ストアはSタイプです。
有効なバイトアドレスは、レジスタrs1を符号拡張された12ビットオフセットに加算することによって得られる。
ロードは、メモリからレジスタrdに値をコピーします。
ストアはレジスタrs2の値をメモリにコピーします。

LW命令は、メモリからrdに32ビット値をロードします。
LHはメモリから16ビットの値をロードし、次にrdに格納する前に32ビットに符号拡張します。
LHUはメモリから16ビットの値をロードしますが、rdに格納する前にゼロを32ビットに拡張します。
LBおよびLBUは、8ビット値についても同様に定義されます。
SW、SH、およびSB命令は、レジスタrs2の下位ビットから32ビット、16ビット、および8ビットの値をメモリに格納します。

最高のパフォーマンスを得るには、すべてのロードとストアの実効アドレスを各データ型（つまり、32ビットアクセスの場合は4バイト境界、16ビットアクセスの場合は2バイト境界）に合わせて自然に配置する必要があります。
ベースISAは、整列していないアクセスをサポートしますが、実装によっては非常に遅く実行される可能性があります。
さらに、自然に整列されたロードとストアはアトミックに実行されることが保証されますが、整列がずれたロードとストアはアトミック性を保証するために追加の同期を必要とされます。
↑atomically、atomicity アトミック、アトミック性ってどういうことなんだろうか

従来のコードを移植するときに、ずれたアクセスが必要になることがあり、多くのアプリケーションでパフォーマンスを向上させるには、任意の形式のパック SIMD 拡張を使用することが不可欠です。
通常のロードおよびストア命令による不整列アクセスをサポートするための我々の理論的根拠は、整列していないハードウェアサポートの追加を簡素化することである。
1つの選択肢は、ベースISAの不整列されたアクセスを禁止し、不整列されたアクセスを処理するための特別な命令、または不整列されたアクセスのための新しいハードウェアアドレッシングモードを提供することです。
特別な命令は、使用するのが難しく、ISAを複雑にし、しばしば新しいプロセッサ状態（例えば、SPARC VIS整列アドレスオフセットレジスタ）を追加するか、既存のプロセッサ状態へのアクセスを複雑にします（MIPS LWL / LWR部分レジスタ書き込み）。
さらに、ループ指向のパックドSIMDコードでは、オペランドの位置がずれている余分なオーバーヘッドが生じるため、ソフトウェアがオペランドの配置に応じて複数の形式のループを提供されるようになり、コード生成が複雑になり、ループ起動オーバーヘッドが増加します。
新しい不整列ハードウェアアドレス指定モードでは、命令符号化においてかなりのスペースを取るか、または非常に単純化されたアドレッシングモード（例えば、レジスタ間接のみ）を必要とします。
我々は、不整列されたアクセスに対してアトミック性を要求するわけではないので、簡単な実装で、機械トラップとソフトウェアハンドラを使用して、一部またはすべての不整列されたアクセスを処理できます。
ハードウェアの不整列サポートが提供されている場合、ソフトウェアは通常のロードおよびストア命令を使用するだけでこれを利用できます。
ハードウェアは、ランタイムアドレスが整列されているかどうかによってアクセスを自動的に最適化することができます。

2.7 メモリモデル

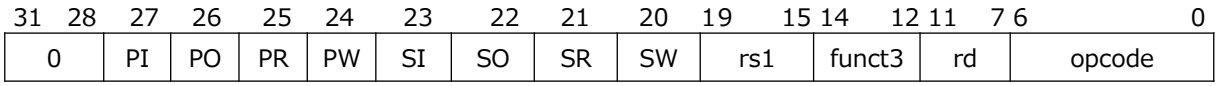
このセクションは、現在のプログラミング言語のメモリモデルを効率的にサポートできるように、RISC-V メモリモデルが現在改訂中であるため、古くなっています。
修正された基本メモリモデルには、少なくとも同じハートからの同じアドレスへのロードを並べ替えることができず、命令間の構文データ依存性が尊重されることを含む、さらなる順序制約が含まれます。
↑ hart ハート ってどこを指しているのか？

ベース RISC-V ISA は、単一のユーザアドレス空間内で複数の同時実行スレッドをサポートします。
各 RISC-V ハードウェアスレッドまたはハートは、独自のユーザーレジスタステートとプログラムカウンタを持ち、独立したシーケンシャルな命令ストリームを実行します。
実行環境は、RISC-V ハートの作成および管理方法を定義します。
RISC-V ハーツは、各実行環境の仕様で個別に文書化されている実行環境への呼び出しを介して、または共有メモリシステムを介して直接、他のハートと通信して同期することができます。
RISC-V ハーツは I / O デバイスとやりとりすることも、I / O に割り当てられたアドレス空間の一部にロードやストアを介して間接的に相互に対話することもできます。
↑ だから、hart harts ってなんなのよ

ハートという用語は、ソフトウェア管理のスレッドコンテキストとは対照的に、ハードウェアスレッドを明確かつ簡潔に記述するために使用します。
↑ hart やっと出てきた。先に言わんかーい （＾＾；）。ハードウェアの実行単位とか並列実行が何個あるかとかのようだな。

ベース RISC-V ISA では、各 RISC-V ハートは、プログラム順に順次実行されるかのように、それ自身のメモリ操作を観察します。
RISC-V はハート間の緩やかなメモリモデルを持ち、異なる RISC-V ハーツからのメモリ操作の順序を保証するために明示的な FENCE 命令を必要とします。
第 7 章では、追加の同期操作を提供するオプションのアトミックメモリ命令拡張 "A"について説明します。

--2018/05/08

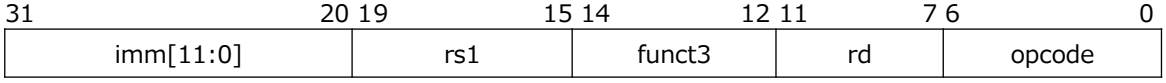


フェンス命令は、他の RISC-V ハートや外部デバイスやコプロセッサで見られるように、デバイス I / O とメモリアクセスを順序付けるために使用されます。
デバイス入力（I）、デバイス出力（O）、メモリ読み出し（R）、およびメモリ書き込み（W）の任意の組み合わせは、それらの任意の組み合わせに関して順序付けられます。
正式には、他の RISC-V ハートまたは外部装置は、フェンスの前に設定された先行のオペレーションの前に、フェンスに続いて後続セットのオペレーションを観察することはできません。
↑非公式に、他の RISC-V ハートまたは外部デバイスは、フェンスの前に設定された前任者の任意の操作がフェンスに続いて、後続のセットで任意の操作を観察することができます。 どこで切るかによって真逆の訳になるような？
実行環境は、どの I/O 操作が可能であるか、特にどのロード命令およびストア命令が、メモリ読み取りおよび書き込みではなく、それぞれデバイス入力およびデバイス出力操作として処理および順序付けされるかを定義します。
例えば、メモリマップされた I/O デバイスは通常、R および W ビットではなく I/O ビットを使用して順序付けされたキャッシュされていないロードおよびストアによってアクセスされます。
命令セット拡張はまた、フェンス内の I および O ビットを使用して順序付けられる新しいコプロセッサ I/O 命令を記述することもできます。

フェンス命令の imm [11：8]、rs1、および rd の未使用フィールドは、将来の拡張でより細かい微量なフェンス用に予約されています。
前方互換性のために、基本実装はこれらのフィールドを無視し、標準ソフトウェアはこれらのフィールドをゼロにしなければいけません。

--2018/05/09

私たちは、単純なマシン実装から高性能を可能にするリラックスメモリモデルを選択しましたが、しかし、完全にリラックスしたメモリモデルはプログラミング言語メモリモデルをサポートするには弱すぎるため、メモリモデルが強化されています。
リラックスしたメモリモデルは、将来のコプロセッサまたはアクセラレータの拡張機能と最も互換性があります。
I / O 命令をメモリ R / W 命令から分離して、デバイスドライバハート内の不要なシリアル化を回避し、追加のコプロセッサまたはI / Oデバイスを制御する代わりの非メモリパスをサポートします。
単純な実装では、先行フィールドと後続フィールドを無視して、すべての操作で常に保守フェンスを実行することができます。



31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	FENCE.I	0	MISC-MEM

FENCE.I 命令は、命令ストリームとデータストリームの同期に使用されます。
RISC-V は、命令メモリへのストアが、FENCE.I 命令が実行されるまで、同じ RISC-V ハートの命令フェッチでは表示されることを保証しません。
FENCE.I 命令では、RISC-V ハートの後続の命令フェッチで、同じ RISC-V ハートにすでに表示されている以前のデータストアがあることを保証するだけです。
FENCE.I は、他の RISC-V ハーツの命令フェッチがマルチプロセッサシステム内のローカルハートストアを監視することを保証していません。
すべての RISC-V ハートで命令メモリへのストアを表示させるには、すべてのリモート RISC-V ハートが FENCE.I を実行するように要求する前に、書き込みハートがデータ FENCE を実行する必要があります。

FENCE.I 命令の imm [1：0]、rs1、および rd の未使用フィールドは、将来の拡張でより細かいフェンス用に予約されています。
前方互換性のために、基本実装はこれらのフィールドを無視し、標準ソフトウェアはこれらのフィールドをゼロにしなければなりません。

FENCE.I 命令は、さまざまな実装をサポートするように設計されています。
簡単な実装では、FENCE.I が実行されるときにローカル命令キャッシュと命令パイプラインをフラッシュすることができます。
より複雑な実装では、すべてのデータ（命令）キャッシュミスで命令（データ）キャッシュを詮索するか、またはローカルストア命令によって書き込まれているときにプライマリ命令キャッシュからのラインを無効にするために包括的な統一プライベート L2 キャッシュを使用することがあります。
命令キャッシュとデータキャッシュがこのようにコヒーレントに保たれている場合は、パイプラインのみを FENCE.I でフラッシュする必要があります

私たちは(MAJC [30]のように)、「ストア命令語」命令は考慮しましたが、含めませんでした。
JIT コンパイラは、単一の FENCE.I の前に命令の大きなトレースを生成し、I キャッシュに存在しないことが分かっているメモリ領域に変換された命令を書き込むことによって、命令キャッシュの詮索/無効化オーバーヘッドを償却することができます。

2.8 コントロールおよびステータスレジスタの命令

システム命令は、特権アクセスを必要とし、I 型命令フォーマットを使用して、符号化されるシステム機能にアクセスするために使用されます。
これらは 2 つの主要なクラスに分けることができます：
これらは、アトミックにリード・モディファイ・ライト・コントロールおよびステータス・レジスタ（CSR）と、その他すべての潜在的に特権のある命令です。
CSR 命令は、次のセクションで説明する 2 つのユーザーレベルの SYSTEM 命令を使用して、このセクションでは説明されます。

SYSTEM 命令は、より単純な実装が常に単一のソフトウェアトラップハンドラにトラップできるように定義されています。
より洗練された実装は、ハードウェア内の各システム命令のより多くを実行することができるかもしれません。

CSR 命令

標準的なユーザレベルのベース ISA では、ほんの一握りの読取り専用カウンタ CSR にアクセス可能ですが、ここでは CSR の完全なセットを定義します。

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

CSRRW（アトミック 読み取り/書き込み CSR）命令は、CSR および整数レジスタの値をアトミックに交換します。
CSRRW は CSR の古い値を読み取り、その値を XLEN ビットにゼロ拡張した後、整数レジスタ rd に書き込みます。
rs1 の初期値は CSR に書き込まれます。
rd = x0 の場合、命令は CSR を読み取らず、CSR の読み込みで発生する可能性のある副作用を引き起こさないものとします。

CSRRS（アトミック リードおよびセット ビット CSR）命令は、CSR の値を読み取り、その値をゼロ拡張して XLEN ビットにし、それを整数レジスタ rd に書き込みます。
整数レジスタ rs1 の初期値は、CSR に設定するビット位置を指定するビットマスクとして扱われます。
CSR ビットが書き込み可能である場合、rs1 の上位ビットは CSR に対応するビットを設定します。
CSR の他のビットは影響を受けません（ただし、CSR は書かれたことによる副作用があるかもしれません）。

CSRRC（CSR のアトミックリードとクリアビット）命令は CSR の値を読み取り、値をゼロ拡張して XLEN ビットにし、それを整数レジスタ rd に書き込みます。
整数レジスタ rs1 の初期値は、CSR でクリアされるビット位置を指定するビットマスクとして扱われます。
CSR ビットが書き込み可能である場合、rs1 の上位ビットは CSR 内の対応するビットをクリアします。
CSR の他のビットは影響を受けません。

CSRRS と CSRRC の両方について、rs1 = x0 ならば、命令は CSR にまったく書き込まない。
したがって、読み取り専用 CSR へのアクセスで違法命令の例外を発生させるなど、CSR 書き込みで発生する可能性のある副作用を引き起こさないものとします。
rs1 が x0 以外のゼロの値を保持するレジスタを指定する場合、命令は変更されていない値を CSR に書き戻そうと試み、付随する副作用を引き起こすことに注意してください。

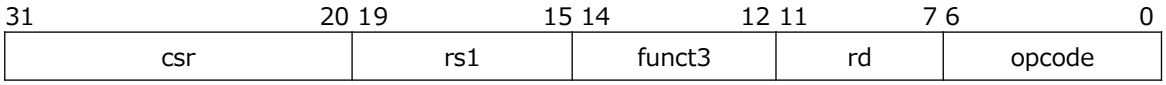
CSRRWI、CSRRSI、および CSRRCI バリアントは、CSRRW、CSRRS、および CSRRC にそれぞれ類似していますが、ただし、整数レジスタの値ではなく rs1 フィールドにエンコードされた 5 ビットの符号なし即値(uimm[4:0]) フィールドをゼロ拡張した XLEN ビット値を使用して CSR を更新する点が異なります。
CSRRSI と CSRRCI の場合、uimm [4 : 0]フィールドがゼロの場合、これらの命令は CSR に書き込まれず、CSR 書き込みで発生する可能性のある副作用を引き起こしません。
CSRRWI の場合、rd = x0 の場合、命令は CSR を読み取らず、CSR の読み込みで発生する可能性のある副作用を引き起こしません。

廃止されたカウンタ、instret などのいくつかの CSR は、命令実行の副作用として変更されることがあります。
↑retired counter って 退役とか引退とかいう意味だと思うが、引退したカウンタって何？
このような場合、CSR アクセス命令が CSR を読み取ると、CSR の命令はその命令の実行前に値を読み取ります。
CSR アクセス命令が CSR を書き込む場合、更新は命令の実行後に行われます。
特に、1 つの命令によって instret に書き込まれた値は、次の命令によって読み取られる値になります。
（すなわち、最初の命令のリタイアに起因する instret の増分は、新しい値の書き込みの前に行われます。）。

CSR CSRR rd、csr を読み取るためのアセンブラ疑似命令は、CSRRS rd、csr、x0 としてエンコードされます。
CSRWI csr、rs1 を書くためのアセンブラ疑似命令は CSRRW x0、csr、rs1 としてエンコードされ、一方、CSRWI csr、uimm は CSRRWI x0、csr、uimm としてエンコードされます。

古い値が必要でない場合、CSR のビットをセットしてクリアするために、さらにアセンブラ疑似命令が定義されています。：
CSRS / CSRC csr、rs1; CSRSI / CSRCI csr、uimm。

タイマとカウンタ



12	5	3	5	7
RDCYCLE[H]	0	CSRRS	dest	SYSTEM
RDTIME[H]	0	CSRRS	dest	SYSTEM
RDINSTRET[H]	0	CSRRS	dest	SYSTEM

RV32I には、12 ビットの CSR アドレス空間にマップされ、CSRRS 命令を使用して 32 ビット単位でアクセスされる、64 ビットの読み取り専用ユーザーレベルカウンタが多数用意されています。
↑ a number of を 多数のと訳すか、いくつかの と訳すか ？

RDCYCLE 擬似命令は、ハートが過去の任意の開始時刻から実行されており、プロセッサコアが実行するクロックサイクル数のカウントを保持し、サイクル CSR の低 XLEN ビットを読み込みます。
！低 XLEN ビットとは 64bit の下位 32bit の事
RDCYCLEH は、同じサイクルカウンタのビット 63-32 を読み取る RV32I 専用の命令です。
基礎となる 64 ビットカウンタは実際にはオーバーフローしません。
サイクルカウンタが進む速度は、実装および動作環境によって異なります。
実行環境は、サイクルカウンタがインクリメントする現在のレート（サイクル/秒）を決定する手段を提供する必要があります。

RDTIME 擬似命令は、過去の任意の開始時刻から経過したウォールクロックリアルタイムをカウントする time CSR の低 XLEN ビットを読み取ります。
RDTIMEH は、同じリアルタイムカウンタのビット 63-32 を読み取る RV32I 専用の命令です。
基礎となる 64 ビットカウンタは実際にはオーバーフローしません。
実行環境は、リアルタイムカウンタの周期（秒/ティック）を決定する手段を提供する必要があります。
期間は一定でなければなりません。
単一のユーザアプリケーション内のすべてのハーツのリアルタイムクロックは、リアルタイムクロックの 1 チック以内に同期される必要があります。
環境は、クロックの精度を決定する手段を提供する必要があります。

RDINSTRET 擬似命令は、instret CSR の下位 XLEN ビットを読み取ります。これは、過去の任意の開始点からこのハートによって廃止された命令の数をカウントします。
RDINSTRETH は、同じ命令カウンタのビット 63-32 を読み取る RV32I 専用の命令です。
実際にオーバーフローすることのない根本的な 64 ビットのカウンタです。

次のコードシーケンスは、カウンタが上と下半分を読み取る間にオーバーフローしても、有効な 64 ビットサイクルカウンタ値を x3：x2 に読み込みます。

```
again:
   rdcycleh  x3
   rdcycle   x2
   rdcycleh  x4
   bne       x3, x4, again
```

図 2.5：RV32 の 64 ビット・サイクル・カウンタを読み取るためのサンプル・コード

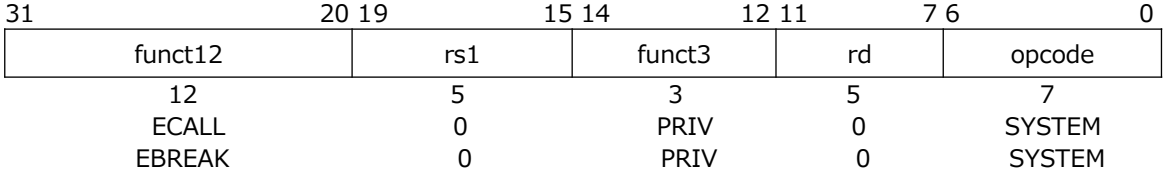
これらの基本カウンタは、基本的なパフォーマンス分析、適応的かつ動的な最適化、およびアプリケーションがリアルタイムストリームで動作するために不可欠であるため、すべての実装で提供することを義務づけています。
パフォーマンスの問題を診断するために追加のカウンタを用意する必要があり、これらのカウンタは、ユーザーレベルのアプリケーションコードから低オーバーヘッドでアクセスできるようにする必要があります。

我々はカウンターは RV32 でも 64 ビット幅であることを必要とし、それ以外の場合は、値がオーバーフローしたかどうかをソフトウェアが判断することは非常に困難です。
ローエンドの実装では、各カウンタの上位 32 ビットは、下位 32 ビットのオーバーフローによってトリガされるトラップハンドラによってインクリメントされるソフトウェアカウンタを使用して実装できます。
上で説明したサンプルコードは、個々の 32 ビット命令を使用して完全な 64 ビット幅の値を安全に読み取る方法を示しています。

アプリケーションによっては、同じ瞬間に複数のカウンタを読み取ることができることが重要です。
マルチタスク環境で実行すると、カウンタを読み取ろうとしている間にユーザースレッドがコンテキスト切り替えを受ける可能性があります。
1 つの解決策は、ユーザスレッドが他のカウンタを読み取る前後にリアルタイムカウンタを読み取って、シーケンスの途中でコンテキストスイッチが発生したかどうかを判断することです。この場合、読み取りを再試行できます。
ユーザスレッドがカウンタ値をアトミックにスナップショットできるように出力ラッチを追加することを検討しましたが、特に、より豊富なカウンタセットを使用する実装では、ユーザコンテキストのサイズが大きくなります。

-- 2018/05/13

2.9 環境呼び出しとブレークポイント



ECALL 命令は、通常はオペレーティングシステムであるサポート実行環境への要求を行うために使用されます。
システムの ABI は、環境要求のパラメータがどのように渡されるかを定義しますが、通常、これらは整数レジスタファイルの定義された場所にあります。

EBREAK 命令はデバッガによってコントロールがデバッグ環境に戻されるように使用されます。

ECALL と EBREAK は、以前は SCALL と SBREAK という名前でした。
命令は同じ機能とエンコーディングを持ちますが、スーパバイザレベルのオペレーティングシステムまたはデバッガを呼び出すより一般的に使用できることを反映するために名前が変更されました。

-- 2018/05/13

-- 空ページを置いて次ページへ。

第 3 章

RV32E ベース整数命令セット、バージョン 1.9

この章では、組み込みシステム用に設計された RV32I の縮小版である RV32E 基本整数命令セットについて説明します。
主な変更点は、整数レジスタの数を 16 に減らし、RV32I で必須のカウンタを削除することです。
この章では、RV32E と RV32I の違いについてのみ説明しますので、第 2 章の後にお読みください。

RV32E は、組み込みマイコン用にさらに小型の基本コアを提供するように設計されています。
このドキュメントのバージョン 2.0 でこの可能性について言及しましたが、最初はこのサブセットの定義に抵抗しました。
しかし、可能な限り最小限の 32 ビットマイクロコントローラの需要があり、この領域で多様化を先取りするために、
RV32I、RV64I、および RV128I に加えて、RV32E を 4 番目の標準 ISA として定義しました。
E バリアントは、32 ビットのアドレス空間幅に対してのみ標準化されています。

3.1 RV32E プログラマのモデル

RV32E は、整数レジスタのカウントを 16 個の汎用レジスタ（x0～x15）に縮小します.x0 は専用ゼロレジスタです。

我々は、小さな RV32I コア設計では、上位 16 個のレジスタがメモリを除くコアの総面積の約 4 分の 1 を消費することを発見したため、コアの消費電力が約 25%削減され、対応するコア電力が削減されます。

この変更には、異なる呼び出し規約と ABI が必要です。
特に、RV32E はソフトフロート呼び出し規約でのみ使用されます。
ハードウェア浮動小数点を持つシステムでは、I ベースを使用する必要があります。

3.2 RV32E 命令セット

RV32E は RV32I と同じ命令セットエンコーディングを使用しますが、命令でレジスタ指定子 x16～x31 を使用すると不正な命令例外が発生します。

将来の任意の標準拡張は、縮小されたレジスタ指定フィールドによって解放された命令ビットを使用しないので、これらは標準以外の拡張で利用可能です。

さらに単純化すると、カウンタ命令（rdcycle [h]、rdtime [h]、rdinstret [h]）はもはや必須ではありません。

必須カウンタは追加のレジスタとロジックを必要とし、より多くのアプリケーション固有の機能に置き換えることができます。

3.3 RV32E 拡張

RV32E は M、A、と C のユーザーレベルの標準拡張で拡張できます。

RV32E サブセットでハードウェア浮動小数点をサポートするつもりはありません。
ハードウェア浮動小数点ユニットでは、レジスタ数の削減による節約は無視できるほど小さくなり、ABIs の拡散を減らしたいと考えています。
↑ABI って(application binary interface)だよね

RV32E システムの特権アーキテクチャには、ユーザモードとマシンモード、および第 II 巻で説明されている物理メモリ保護スキームが含むことができます。

私たちは、RV32E サブセットを備えた完全な Unix スタイルのオペレーティングシステムをサポートするつもりはありません。
OS の可能性のあるコアのコンテキストでは、レジスタ数の削減による節約は無視できなくなり、私たちは OS の断片化を避けたいと考えています。

-- 2018/05/15

第４章

RV64I ベース整数命令セット、
バージョン 2.0

この章では、RV64I 基本整数命令セットについて説明します。これは、第２章で説明した RV32I 変形を基にしています。
この章では、RV32I との相違点のみを示していますので、前の章と併せて読んでください。

4.1 レジスタ状態

RV64I は、整数レジスタとサポートされるユーザアドレス空間を 64 ビットに広げます（図 2.1 の XLEN = 64）。

4.2 整数計算命令

RV64I の 32 ビット値を操作する追加の命令の変形が用意されています。これはオペコードの接尾辞「W」で示されます。
これらの「* W」命令は、その入力の上位 32 ビットを無視し、常に 32 ビットの符号付き値を生成する。すなわち、ビット XLEN-1～31 が等しくなります。
！XLEN＝64 なので、63:31 が同じになる。符号拡張されているということ
RV32I で不正な命令例外が発生します。

コンパイラと呼び出し規約では、すべての 32 ビット値が 64 ビットレジスタの符号拡張形式で保持されるという不変量を維持しています。
32 ビットの符号なし整数でも、ビット 31 をビット 63 から 32 に拡張する。
その結果、符号付き 32 ビット整数から符号付き 64 ビット整数への変換と同様に、符号なし 32 ビット整数と符号付き 32 ビット整数の間の変換は、演算なしです。
既存の 64 ビット幅の SLTU および符号なし分岐比較は、この不変量の下で符号なし 32 ビット整数に対しても正しく動作します。
同様に、32 ビット符号拡張整数の既存の 64 ビット幅論理演算は、符号拡張プロパティを保持します。
32 ビット値の妥当な性能を保証するために、追加およびシフトにはいくつかの新しい命令（ADD [I] W / SUBW / SxxW）が必要です。

整数レジスタ - 即時命令

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-即値[11:0]	src	ADDIW	dest	OP-IMM-32

ADDIW は、符号拡張された 12 ビットの即値をレジスタ rs1 に加算し、rd の 32 ビット結果の適切な符号拡張を生成する RV64I のみの命令です。
オーバーフローは無視され、結果は 64 ビットに符号拡張された結果の下位 32 ビットになります。
注、ADDIW rd,rs1,0 は、レジスタ rs1 の下位 32 ビットの符号拡張をレジスタ rd に書き込みます（アセンブラ疑似演算 SEXT.W）。

31 26	25	24 20	19 15	14 12	11 7	6 0
imm[11:6]	imm[5]	imm[4:0]	rs1	funct3	rd	opcode
6	1	5	5	3	5	7
000000	shamt[5]	shamt[4:0]	src	SLLI	dest	OP-IMM
000000	shamt[5]	shamt[4:0]	src	SRLI	dest	OP-IMM
010000	shamt[5]	shamt[4:0]	src	SRAI	dest	OP-IMM
000000	0	shamt[4:0]	src	SLLIW	dest	OP-IMM-32
000000	0	shamt[4:0]	src	SRLIW	dest	OP-IMM-32
010000	0	shamt[4:0]	src	SRAIW	dest	OP-IMM-32

定数によるシフトは、RV32I と同じ命令オペコードを使用して I 型形式の特殊化としてエンコードされます。
シフトされるオペランドは rs1 であり、シフト量は RV64I の I 即値フィールドの下位 6 ビットにエンコードされます。
右のシフトタイプはビット 30 で符号化されます。
SLLI は論理的な左シフトです（ゼロは下位ビットにシフトされます）。 SRLI は論理右シフトです（0 は上位ビットにシフトされます）。 SRAI は算術右シフトです（元の符号ビットは空いている上位ビットにコピーされます）。
RV32I の場合、imm [5] 6 = 0 の場合、SLLI、SRLI、および SRAI は不正な命令例外を生成します。

SLLIW、SRLIW、および SRAIW は、同様に定義されていますが、32 ビット値で動作し、符号付き 32 ビット結果を生成する RV64I 専用の命令です。
SLLIW、SRLIW、および SRAIW は imm [5] ≠ 0 の場合、不正な命令例外を生成します。

31 12	11 7	6 0
Imm[31:12]	rd	opcode
20	5	7
U-即値[31:12]	dest	LUI
U-即値[31:12]	dest	AUIPC

LUI（ロードアッパー即時）は、RV32I と同じオペコードを使用します。
LUI は、20 ビットの U-即値をレジスタ rd のビット 31-12 に配置し、最下位の 12 ビットにゼロを配置します。
32 ビットの結果は 64 ビットに符号拡張されます。

AUIPC（PC に即値の上位を追加）は、RV32I と同じオペコードを使用します。
AUIPC（PC に即値の上位を追加）は、PC 相対アドレスを作成するために使用され、U タイプ形式を使用します。
AUIPC は、20 ビットの U−即値に 12 の下位ゼロビットを付加し、その結果を 64 ビットに符号拡張し、それを pc に加え、結果をレジスタ rd に格納します。

整数レジスタ - レジスタ演算

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SRA	dest	OP
0000000	src2	src1	ADDW	dest	OP-32
0000000	src2	src1	SLLW/SRLW	dest	OP-32
0100000	src2	src1	SUBW/SRAW	dest	OP-32

ADDW および SUBW は、ADD および SUB と同様に定義されますが、32 ビット値で動作し、符号付き 32 ビット結果を生成する RV64I 専用の命令です。
オーバーフローは無視され、結果の下位 32 ビットは 64 ビットに符号拡張され、デスティネーション・レジスタに書き込まれます。

SLLW、SRLW、および SRAW は、同様に定義されていますが、32 ビット値で動作し、符号付き 32 ビット結果を生成する RV64I 専用の命令です。
シフト量は rs2 [4 : 0]で与えられます。

4.3 ロードとストア命令

RV64I はアドレス空間を 64 ビットに拡張します。
実行環境は、アドレス空間のどの部分にアクセスするのが正当であるかを定義します。

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

LD 命令は、メモリから RV64I のレジスタ rd に 64 ビットの値をロードします。

LW 命令はメモリから 32 ビット値をロードし、これを 64 ビットに符号拡張してから RV64I のレジスタ rd に格納します。
一方、LWU 命令は、RV64I のメモリから 32 ビット値をゼロ拡張します。
LH および LHU は、8 ビット値の LB および LBU と同様に、16 ビット値に対しても同様に定義されます。
SD、SW、SH、および SB 命令は、レジスタ rs2 の下位ビットから 64 ビット、32 ビット、16 ビット、および 8 ビットの値をそれぞれメモリに格納します。

4.4 システム命令

RV64I では、CSR 命令が 64 ビット CSRs を操作できます。
特に、RDCYCLE、RDTIME、および RDINSTRET の疑似命令は、サイクル、時間、および開始カウンタの全 64 ビットを読み取ります。
したがって、RDCYCLEH、RDTIMEH、RDINSTRETH 命令は不要であり、RV64I では不正です。

第 5 章

RV128I 基本整数命令セット、
バージョン 1.7

「メモリアドレッシングとメモリ管理に十分なアドレスビットを持たないことから回復することが難しいコンピュータ設計では、間違いが 1 つしかありません。」
ベル と ストレッカー、ISCA-3、1976。

この章では、128 ビットアドレス空間をサポートする RISC-V ISA の変形である RV128I について説明します。
この変形は、既存の RV32I および RV64I 設計の直接的な外挿(推定)です。

整数レジスタの幅を拡張する主な理由は、より大きなアドレス空間をサポートするためです。
64 ビット以上の平なアドレス空間が必要な場合は明確ではありません。
執筆時点では、Top500 ベンチマークで測定された世界最速のスーパーコンピュータは 1PB 以上の DRAM を搭載しており、すべての DRAM が単一のアドレス空間に存在する場合は 50 ビット以上のアドレス空間が必要になります。
いくつかの倉庫規模のコンピュータの中には、既に大量の DRAM が搭載されており、新しい高密度個体不揮発性メモリや高速相互接続技術によって、さらに大きなメモリ空間の需要を牽引が求められる場合があります。
エクサスケール システム研究は、57 ビットのアドレス空間を占める 100PB のメモリシステムを対象としています。
過去の成長率では、2030 年までに 64 ビット以上のアドレス空間が必要になる可能性があります。

歴史は、64 ビット以上のアドレス空間が必要となることが明らかになったときには、セグメンテーション、96 ビットアドレス空間、ソフトウェア回避策などのアドレス空間を拡張する選択肢についての集中的な議論を繰り返すことを示唆している回避策は、、最終的には 128 ビット アドレス空間は最もシンプルで最良の解決策として採用されます。

現時点では 128 ビットアドレス空間の実際の使用に基づいて設計を進化させる必要があるため、RV128 仕様を凍結していません。
↑ まだ変わることもあるかもよってこと

RV128I は、整数レジスタを 128 ビットに拡張し、RV32I の上に RV64I を構築するのと同様に、RV64I を構築したものです（すなわち、XLEN = 128）。
ほとんどの整数計算命令は、XLEN ビットで動作するように定義されているので変更されません。
レジスタの下位ビットの 32 ビット値で動作する RV64I "* W"整数命令が保持され、そして、128 ビット整数レジスタの下位ビットに保持されている 64 ビット値で動作する新しい「* D」整数命令セットが追加されます。
「* D」命令は、標準の 32 ビットエンコーディングで 2 つの主要なオペコード（OP-IMM-64 および OP-64）を消費します。

即値（SLLI / SRLI / SRAI）によるシフトは、現在 I 即値の下位 7 ビットを使用して符号化され、可変シフト（SLL / SRL / SRA）はシフト量ソースレジスタの下位 7 ビットを使用します。

LDU（ロード ダブル 符号なし）命令は、既存の LOAD メジャーオペコードと、新しい LQ 命令と SQ 命令を使用してクワッドワード値をロードして格納する命令を一緒に追加します。
SQ は STORE メジャーオペコードに追加され、LQ は MISC-MEM メジャーオペコードに追加されます。

浮動小数点命令セットは変更されていませんが、128 ビット Q 浮動小数点拡張は FMV.X.Q および FMV.Q.X 命令と、T（128 ビット）整数フォーマットとの追加の FCVT 命令をサポートできるようになりました。

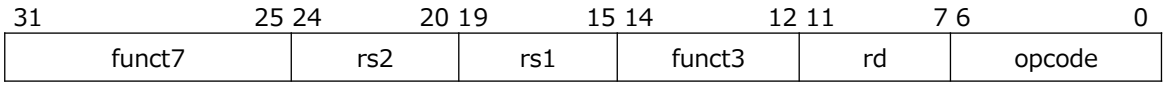
--2018/05/19

第 6 章
整数乗算および除算のための "M"標準拡張、バージョン 2.0

この章では、 "M"という名前の標準整数乗除算命令拡張について説明し、2 つの整数レジスタに保持された値を乗算または除算する命令を含んでいます。

ローエンドの実装を単純化するために、または整数の乗除算演算が接続されたアクセラレータではあまり頻繁ではない、またはより適切に処理されるアプリケーションのために、整数の乗算と除算をベースから分離します。
↑基本命令セットには乗除算は組み込まずに別にした。そりゃ乗除算は面積食うからね。

6.1 乗算演算

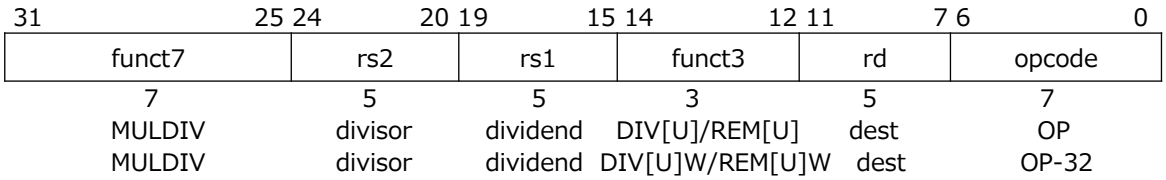


7	5	5	3	5	7
MULDIV	multiplier	multiplicand	MUL/MULH[[SU]	dest	OP
MULDIV	multiplier	multiplicand	MULW	dest	OP-32

MUL は XLEN ビット* XLEN ビットの乗算を行い、下位の XLEN ビットを送り先レジスタに配置します。
MULH、MULHU、および MULHSU は同じ乗算を行いますが、符号付き*符号付き、符号なし*符号付き、および符号付き*符号なし乗算の場合、フル 2 * XLEN ビット積の上位 XLEN ビットを返します。
同じ製品の上位ビットと下位ビットの両方が必要な場合、推奨コードシーケンスは次のとおりです。 : MULH [[S] U] rdh, rs1, rs2; MUL rdl, rs1, rs2（送り元レジスタ指定子は同じ順序でなければならず、rdh は rs1 または rs2 と同じにすることはできません）。
マイクロアーキテクチャは、2 つの別々の乗算を実行する代わりに、これらを 1 つの乗算演算に融合することができます。

MULW は RV64 でのみ有効で、送り元レジスタの下位 32 ビットを乗算し、結果の下位 32 ビットの符号拡張を送り先レジスタに配置します。
MUL を使用して 64 ビット積の上位 32 ビットを得ることができますが、符号付き引数は適切な 32 ビット符号付き値でなければなりませんし、符号なし引数は上位 32 ビットをクリアする必要があります。

6.2 除算操作



DIV と DIVU は XLEN ビットで XLEN ビットの符号付きおよび符号なし整数除算を実行します。
REM と REMU は、対応する除算演算の残りの部分を提供します。
商と剰余の両方が同じ除算から必要とされる場合、推奨されるコードシーケンスは次のとおりです。DIV [U] rdq, rs1,rs2; REM [U] rdr, rs1, rs2（rdq は rs1 または rs2 と同じにすることはできません）。
マイクロアーキテクチャは、2 つの別個の除算を実行する代わりに、これらを 1 つの除算演算に融合することができます。

ゼロ除算と除算オーバーフローの意味を表 6.1 に要約します。
0 による除算の商は、すべてのビットがセットされています、すなわち、符号なし除算の場合は 2^{XLEN}-1、符号付き除算の場合は-1 です。
ゼロによる除算の残りは被除数と等しいです。
符号付き除算のオーバーフローは、負の整数-2^{XLEN-1} を-1 で除算した場合にのみ発生します。
符号付き除算オーバーフローの商は被除数に等しく、余りはゼロです。
符号なし除算のオーバーフローは発生しません。

状態	被除数	除数		DIVU	REMU	DIV	REM
ゼロで除算	x	0		2 XLEN-1	x	-1	x
オーバーフロー(符号付きのみ)	-2^{XLEN-1}	-1		-	-	-2^{XLEN-1}	0

表 6.1：ゼロ除算および除算オーバーフローのセマンティクス

私たちは、整数ゼロ除算の例外を発生させることを考慮しました。これらの例外は、ほとんどの実行環境でトラップを引き起こします。
しかし、これは標準 ISA（浮動小数点例外フラグをセットし、デフォルト値を書き込むが、トラップは発生しません）の唯一の算術トラップであり、この場合、実行環境のトラップハンドラと対話する言語実装が必要となります。
さらに、ゼロ除算例外が即座の制御フロー変更を引き起こさなければならない言語規格(標準)では、各分岐演算に 1 つの分岐命令のみを追加する必要があり、この分岐命令は分割後に挿入することができ、 非常に予想通りに実行されず、ランタイムオーバーヘッドを少し追加します。
↑この文章はセンテンスが長いので係り受けがいまいちあやしいかも

除算回路を簡略化するために、符号なしと符号付きの両方の除算値をゼロで除算すると設定されたすべてのビットの値が返されます。
すべての 1s の値は、最大の符号なし数値を表す符号なし除算に返される自然数と、単純な符号なし除算器実装の自然な結果の両方です。
符号付き除算は符号なし除算回路を使用して実装されることが多く、同じオーバーフロー結果を指定するとハードウェアが単純化されます。

DIVW 命令と DIVUW 命令は RV64 でのみ有効で、rs1 の下位 32 ビットを rs2 の下位 32 ビットで除算し、それぞれ符号付き整数と符号なし整数として扱い、32 ビット商を rd に配置し、64 ビットに符号拡張します 。
REMW 命令と REMUW 命令は RV64 でのみ有効で、対応する符号付きおよび符号なしの剰余演算をそれぞれ提供します。

REMW と REMUW はともに、常に 32 ビットの結果を 64 ビットに符号拡張します（ゼロ除算を含む）。

--2018/05/19

-- 1 ページ空けて次ページへ

第７章

アトミック命令のための "A" 標準拡張、バージョン 2.0

！アトミックって何？ がここで明かされるのか？

このセクションは、現在のプログラミング言語のメモリモデルを効率的にサポートできるように、RISC-V メモリモデルが現在改訂中であるため、多少古くなっています。
修正された基本メモリモデルには、少なくとも同じハートからの同じアドレスへのロードを並べ替えることができず、および命令間の構文データ依存性が尊重されることを含む、さらなる順序制約が含まれます。

標準的なアトミック命令拡張は、命令サブセット名 ”A” によって示され、同じメモリ空間で動作する複数の RISC-V ハート間の同期をサポートするために原子的に読み取り－変更－書き込み メモリの命令を含んでいます。
提供される 2 つの形式のアトミック命令は、ロードー予約/ストアー条件付き命令およびアトミック フェッチーアンドーオペレーション・メモリ命令です。
どちらの種類のアトミック命令は、順序付けなし、取得、解放、および逐次的に一貫性のあるセマンティクスを含む、様々なメモリ整合性順序をサポートします。
これらの命令は、RISC-V が RCsc メモリ一貫性モデル[10]をサポートすることを可能にすることができます。

多くの議論の末、言語コミュニティとアーキテクチャコミュニティは、標準的なメモリ一貫性モデルとしてリリースの一貫性を解決したように見えるため、RISC-V アトミックサポートはこのモデルを中心に構築されています。

7.1 アトミック命令の順序付けの指定

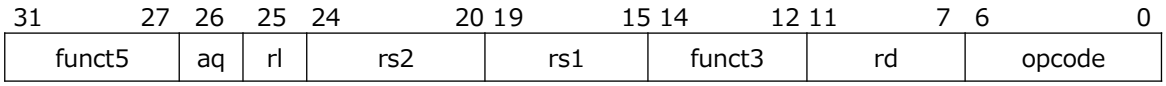
基本 RISC-V ISA には緩やかなメモリモデルがあり、FENCE 命令は追加の順序制約を課すために使用されます。
アドレス空間は、実行環境によってメモリ領域と I / O 領域に分割され、FENCE 命令は、これら 2 つのアドレス領域の一方または両方へのアクセスを順序付けるためのオプションを提供します。

リリース一貫性[10]のより効率的なサポートを提供するために、各アトミック命令は 2 つのビット aq と rl を持ち、他の RISC-V ハートで見られる追加のメモリ順序制約を指定します。
ビット順は、原子命令がどのアドレス・ドメインにアクセスしているかに応じて、2 つのアドレス・ドメイン、メモリまたは I / O のうちの 1 つにアクセスします。
順序付け制約は他のドメインへのアクセスには暗示されず、FENCE 命令を使用して両方のドメイン間で順序付けする必要があります。

両方のビットがクリアされている場合、アトミックメモリ動作に追加の順序制約条件が課せられません。
aq ビットのみがセットされている場合、アトミック・メモリ操作は獲得アクセスとして扱われ、
すなわち、この RISC-V ハートに対する後続のメモリ操作は、メモリ取得動作の前に行われることは観察できません。
rl ビットだけがセットされている場合、アトミックメモリ操作は解放アクセスとして扱われ、
すなわち、解放メモリ操作は、この RISC-V ハート上の以前のメモリ操作の前に行われることは観察できません。
aq ビットと rl ビットの両方がセットされている場合、アトミックメモリ動作は逐次一致し、同じ RISC-V ハート内で、以前のメモリ動作の前またはその後のメモリ動作の後に発生することは観測できず、 同一のアドレス領域に対するすべての逐次的に一貫した原子メモリ操作の同じグローバル順序で他のハートによって観察されることができる。

理論的には、aq ビットと rl ビットの定義は、グローバルストアのアトミック性を持たない実装を可能にします。
しかし、aq ビットと rl ビットの両方がセットされている場合、アトミック操作のための完全な逐次一貫性が必要です。これは、取得と解放の両方のセマンティクスに加えてグローバルストアの原子性を意味します。
実際には、ハードウェアシステムは、通常、シングルライタのキャッシュコヒーレンスプロトコルとともにローカルプロセッサの順序付け規則に組み込まれたグローバルストアのアトミック性で実装されています。

7.2 ロード予約/ストア条件付き命令



5	1	1	5	5	3	5	7
LR	ordering		0	addr	width	dest	AMO
SC	ordering		src	addr	width	dest	AMO

単一のメモリワードに対する複雑なアトミックメモリ操作は、ロード予約（LR）およびストア条件付き（SC）命令で実行されます。
LR は rs1 のアドレスからワードをロードし、符号拡張された値を rd に配置し、メモリアドレスに予約を登録します。
SC は、そのアドレスに有効な予約がまだ存在する場合、rs1 のアドレスに rs2 のワードを書き込みます。
SC は、成功すると rd に 0 を、失敗した場合は 0 以外のコードを書き込みます。

比較とスワップ（CAS）と LR / SC の両方は、ロック-フリーのデータ構造を構築するのに使用できます。
広範な議論の後、私たちはいくつかの理由から LR / SC を選択しました：
1）CAS は、データ値の変化をチェックするのではなく、アドレスへのすべてのアクセスを監視するため、LR / SC が回避する ABA の問題を抱えています。
2）CAS はまた、3 つのソースオペランド（アドレス、比較値、スワップ値）、および異なるメモリシステムメッセージフォーマットをサポートするために新しい整数命令フォーマットを必要とし、これはマイクロアーキテクチャを複雑にします。
3）さらに、ABA 問題を回避するために、他のシステムでは、カウンタをデータワードとともにテストしインクリメントできるように、ダブル幅の CAS（DW-CAS）を提供しています。
これには、5 つのレジスタを読み込んで 1 つの命令に 2 つの命令を書き込む必要があります。かつ、新しい大規模なメモリシステムのメッセージタイプであり、実装をさらに複雑にします。
4）LR / SC は、CAS を使用する 2 つのロードではなく、1 つのロードしか必要としないため、多くのプリミティブのより効率的な実装を提供します。
（投機的計算のための値を得るために CAS 命令の前に 1 回ロードし、次に CAS 命令の一部として 2 回目のロードを行い、値が更新前に変更されていないかどうかをチェックします）

CAS に対する LR / SC の主な欠点はライブロックであり、これは、以下で説明するように、最終的なフォワード進捗のアーキテクチャ保証を避けています。
もう一つの懸念は、現在の x86 アーキテクチャが DW-CAS に与える影響が、DW-CAS が基本的なマシンプリミティブであることを前提とした同期ライブラリやその他のソフトウェアの移植を複雑にするかどうかです。
考えられる緩和要因は、最近の x86 へのトランザクショナルメモリ命令の追加であり、DW-CAS からの移行を引き起こす可能性があります。

値 1 の障害コードは、不特定の障害をエンコードするために予約されています。
現時点では他の障害コードが予約されており、可搬ソフトウェアは障害コードがゼロではないと仮定する必要があります。
LR および SC は、メモリ内の 64 ビット（RV64 のみ）または 32 ビットの自然なアラインメントで動作します。
ミスアラインアドレスは、ミスアラインアドレス例外を生成します。

SLT / SLTU 命令に必要な既存のマルチプレクサを使用して簡単な実装がこの値を返すことができるように、”未指定” を意味する 1 の不合格コードを予約します。
より具体的な障害コードは、ISA の将来のバージョンまたは拡張で定義される可能性があります。

標準 A 拡張では、特定の制約付き LR / SC シーケンスが最終的に成功することが保証されています。
LR / SC シーケンスの静的コードと、障害の発生時にシーケンスを再試行するコードには、最大 16 個の整数命令を順次メモリに配置する必要があります。
シーケンスが最終的に成功することが保証されるために、LR 命令と SC 命令との間で実行される動的コードは、ロード、ストア、逆方向ジャンプまたは取り去られた逆方向分岐、FENCE、FENCEI、および SYSTEM 命令を除いて、ベース ”I” サブセットからの他の命令のみを含むことができる。
失敗した LR / SC シーケンスを再試行するコードは、後方ジャンプおよび/または分岐を含み、LR / SC シーケンスを繰り返すことができるが、そうでなければ同じ制約を有します。
SC は、実行された最新の LR と同じアドレスになければなりません。
これらの制約を満たさない LR / SC シーケンスは、いくつかの実装でいくつかの試みを完了するかもしれませんが、最終的な成功を保証するものではありません。

CAS の利点の 1 つは、LR / SC 原子配列が一部のシステムで無期限にライブロックできるのに対し、最終的にいくつかのハートが進行することを保証することです。
この懸念を避けるために、我々は、LR / SC 原子配列に前方進行の構造的保証を追加しました。
LR / SC シーケンス内容の制限により、実装では、LR 上のキャッシュラインをキャプチャし、短時間のリモートキャッシュ介入を保留することによって LR / SC シーケンスを完了することができます。
割込みおよび TLB ミスにより、予約が失われる可能性がありますが、最終的にはアトミックシーケンスが完了することがあります。
命令キャッシュと TLB のサイズと結合性に関する過度の制限を避けるため、LR / SC シーケンスの長さを基本 ISA の 64 個の連続した命令バイトに収めるように制限しました。
同様に、シーケンス内に他のロードやストアを許可しないことで、データキャッシュの結合性の制限が回避されました。
分岐とジャンプの制限により、シーケンスで費やせる時間が制限されます。
浮動小数点演算と整数乗算/除算は、適切なハードウェアサポートがない実装でこれらの命令のオペレーティングシステムのエミュレーションを単純化するために許可されませんでした。

実装は、各 LR 上のメモリ空間の任意のサブセットを予約することができ、複数の LR 予約は、単一のハートのために同時にアクティブになることができます。
アドレスを予約するために、このハートの SC と最後の LR との間に他のハートからアドレスへのアクセスが発生していないことが確認できれば、SC は成功することができます。
注：この LR はアドレス引数が異なるかもしれませんが、メモリサブセットの一部として SC のアドレスを予約していることに注意してください。
このモデルに続いて、メモリ変換を伴うシステムでは、以前の LR が別の仮想アドレスを持つエイリアスを使用して同じロケーションを予約していた場合、SC は成功することができます。しかし、仮想アドレスが異なる場合には失敗することが許されます。
他のハートからアドレスへのメモリアクセスが観察可能である場合、またはこのハートに介在するコンテキストスイッチがある場合、またはその間にハートが特権例外復帰命令を実行した場合、SC は失敗する必要があります。

この仕様では、制約付きシーケンスに対するアトミック性の保証を無効にしない限り、より強力な実装をサポートすることを明示的に許可しています。

LR / SC は、ロックフリーのデータ構造を構築するために使用できます。
比較およびスワップ機能を実装するための LR / SC の使用例を図 7.1 に示します。
インライン化されている場合、比較とスワップの機能は 3 つの命令しか必要としません。

```
        # a0 メモリロケーションのアドレスを保持
        # a1 期待値を保持
        # a2 望ましい値を保持
        # a0 戻り値、成功した場合は 0、そうでない場合は !0 を保持
    cas:
        lr.w t0, (a0)       #元の値をロードします。
        bne t0, a1, fail    #が一致しないので失敗します。
        sc.w a0, a2, (a0)   #更新を試みます。
        jr ra               #リターン。
    fail:
        li a0, 1            #リターンを失敗に設定します。
        jr ra               #リターン。
```

図 7.1 : LR / SC を使用した比較およびスワップ機能のサンプルコード

-- 2018/05/25

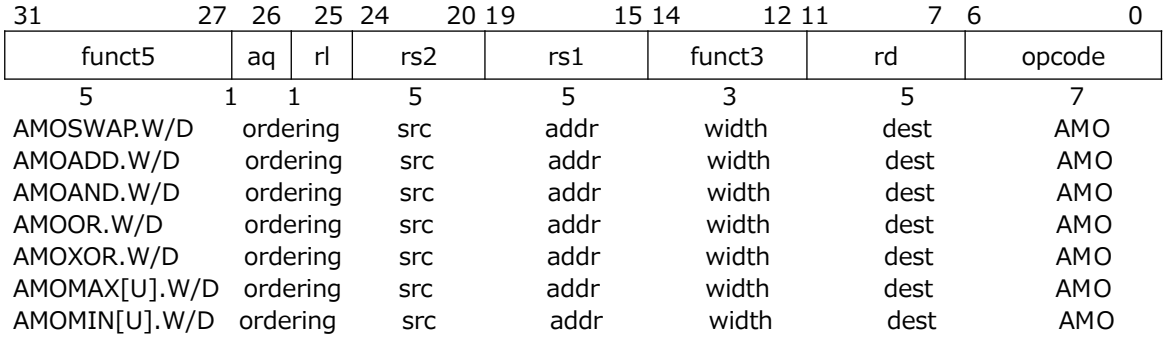
SC 命令は、直前の LR の前に別の RISC-V ハートによって観察されることは決してできません。
LR / SC シーケンスの原子的性質のために、LR と成功した SC との間には何らかのハートからのメモリ操作は発生していないことが確認できます。
LR / SC シーケンスは、SC 命令の aq ビットをセットすることによって、セマンティクスを獲得することができます。
LR / SC シーケンスは、LR 命令の rl ビットをセットすることにより、解除セマンティクスを与えることができます。
LR 命令の aq ビットと rl ビットの両方をセットし、SC 命令の aq ビットをセットすることにより、LR / SC シーケンスは、他の逐次的に一貫したアトミック操作に関して順次整合します。

どちらのビットも LR と SC の両方に設定されていない場合、同じ RISC-V ハートから周囲のメモリ操作の前後に LR / SC シーケンスが発生することが観察されます。
これは、LR / SC シーケンスを使用して並列リダクション操作を実装する場合に適しています。

どちらのビットも LR と SC の両方に設定されていない場合、同じ RISC-V ハートから周囲のメモリ操作の前後に LR / SC シーケンスが発生することが観察されます。
これは、LR / SC シーケンスを使用して並列リダクション操作を実装する場合に適しています。

一般に、マルチワードの原子プリミティブが望ましいが、これがどのような形で取るべきかについてはまだ議論があり、進歩を保証することはシステムに複雑さを加えます。
我々の現在の考えは、任意の標準的な拡張子 "T" として、元のトランザクショナルメモリ提案のラインに沿った限られた容量の小さなトランザクションメモリバッファを含むことです。

7.3 原子メモリ操作



アトミックメモリ操作（AMO）命令は、マルチプロセッサ同期のための読み出し - 変更 - 書き込み動作を実行し、R タイプの命令フ形式で符号化されます。
これらの AMO 命令は、rs1 のアドレスからデータ値を原子にロードし、その値をレジスタ rd に配置し、ロードされた値と rs2 の元の値に二項演算子を適用し、結果を rs1 のアドレスに戻します。
AMOs は、メモリ内の 64 ビット（RV64 のみ）または 32 ビットワードで動作することができます。
RV64 では、32 ビットの AMOs は常に rd に配置された値を符号拡張します。
rs1 に保持されるアドレスは、オペランドのサイズに自然に整列されなければなりません（すなわち、64 ビットワードの場合は 8 バイト整列、32 ビットワードの場合は 4 バイト整列）。
アドレスが自然に整列されていない場合、不整合アドレス例外が生成されます。

サポートされている操作は、スワップ、整数加算、論理 AND、論理 OR、論理 XOR、符号付きおよび符号なし整数の最大値と最小値です。
順序制約がない場合、これらの AMOs を使用して通常、戻り値は x0 に書き込むことによって破棄される、並列リダクション演算を実装することができます。

我々は、LR / SC または CAS よりも高度に並列なシステムに拡張するので、フェッチ・アンド・オペレーション・スタイルの原子プリミティブを提供しました。
単純なマイクロアーキテクチャでは、LR / SC プリミティブを使用して AMOs を実装できます。
さらに複雑な実装では、メモリコントローラに AMOs を実装し、宛先が x0 のときに元の値を取り出すことを最適化できます。

AMOs のセットは、C11 / C ++ 11 アトミックメモリ操作を効率的にサポートし、また、メモリの並列削減をサポートするために選択されました。
AMOs の別の使用法は、I / O 空間内のメモリマップされたデバイスレジスタ（e..g、設定、クリア、またはトグルビット）にアトミック更新を提供することです。

マルチプロセッサ同期の実装を支援するために、AMO はオプションでリリース一貫性セマンティクスを提供します。
aq ビットがセットされていれば、この RISC-V ハートの後のメモリ操作は、AMO の前に行われることはありません。
逆に、rl ビットが設定されている場合、他の RISC-V ハーツは、この RISC-V ハートの AMO に先行するメモリアクセスの前に AMO を観察しません。

AMOs は、C11 および C ++ 11 メモリモデルを効率的に実装するように設計されました。
FENCE R、RW 命令は取得操作とフェンス RW を実装するのに、W はリリースを実装するのに十分ですが、対応する aq または rl ビットが設定された AMOs と比較して不必要な順序付けが必要です。

テスト-アンド-セット スピンロックによって保護されるクリティカルセクションのコードシーケンスの例を図 7.2 に示します。
注：最初の AMO は aq とマークされ、クリティカルセクションの前にロック取得を注文し、2 番目の AMO は rl とマークされて、ロックの解放前にクリティカルセクションを注文することに注意してください。

```
    li          t0, 1       # Initialize swap value.
again:
    amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
    bnez        t0, again   # Retry if held.
    # ...
    # Critical section.
    # ...
    amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
```

図 7.2：相互排除のためのサンプルコード。 a0 にはロックのアドレスが格納されます。

投機的ロック省略の実装を簡略化するために、上記の AMO スワップ・イディオムをロック取得と解放の両方に使用することを推奨します[25]。

アトミック操作の実装を複雑にする危険性がある場合、ロック値がスワップ値と一致すると、マイクロアーキテクチャーは取得スワップ内のストアを削除して、共有または排他クリーン状態に保持されているキャッシュラインを汚染しないようにすることができます。
この効果は、テスト-アンド-セット ロックと似ていますが、コード・パスは短くなります。

“A” 拡張の命令は、一貫性のあるロードおよびストアを連続して提供するためにも使用できます。
連続的に一貫した負荷は、aq と rl の両方を設定した LR として実装できます。
連続的に一貫したストアは、古い値を x0 に書き込み、aq と rl の両方を設定する AMOSWAP として実装できます。

! ちょっと調べてみたけど 原子的、アトミック って (分割不可分) ということらしい
! ここの章では主にメモリについてのアクセスが記してあったように思える。
! あるハーツ(アクセス)での転送を細切れにせず、そのアクセスの間ははバスを確保するようなことなのかな
! sequentially とかとかあるしね

! で、ウィキペディアに不可分操作ってあって、アトミック操作とも書いてある
! https://ja.wikipedia.org/wiki/%E4%B8%8D%E5%8F%AF%E5%88%86%E6%93%8D%E4%BD%9C
! 以下、ウィキペディアより引用
! 不可分操作は、以下の 2 つの条件を満たさなければならない。
! １．全操作が完了するまで、他のプロセスはその途中の状態を観測できない。
! ２．一部操作が失敗したら組合せ全体が失敗し、システムの状態は不可分操作を行う前の状態に戻る。
! システムの他の部分から見て、操作の組合せが一度に成功したか失敗したように見える。
! 途中の状態にアクセスすることはできない。このため不可分操作あるいはアトミック操作（= 原子操作）と呼ぶのである。

! 原子ってこれ以上分割できないっていう意味で使ってるんですな。

! 例えば A と B の 2 つのハーツ(タスクとかスレッド)が走ってて、A が 0～99 番地まで書き込みを行っていて 20 番地まで
! 書いたときに、B が割り込んで 50～59 番地まで書いて、また A に戻って 21～99 番地まで書いたとすると、B が書いた
! 50～59 は書きつぶされてしまう。
!
! この場合は A の書き込みが終わるまでロックかけてバスを占有しないといけない。
! こういう転送を原子(アトミック)転送 ってことで良いかな。
!

第 8 章

単精度浮動小数点の "F"標準拡張、バージョン 2.0

この章では、 "F"という名前の単精度浮動小数点の標準命令セット拡張について説明し、IEEE 754-2008 算術標準[14]に準拠した単精度浮動小数点計算命令を追加します。

8.1 F レジスタの状態

F 拡張は、浮動小数点ユニットの動作モードと例外ステータスを含む 32 個のそれぞれ 32 ビット幅の浮動小数点レジスタ f0～f31、と浮動小数点制御およびステータスレジスタ fcsr を追加します。
この追加の状態を図 8.1 に示します。
RISC-V ISA では浮動小数点レジスタの幅を記述するために FLEN という用語を使用し、F 単精度浮動小数点の拡張では FLEN = 32 を使用します。
ほとんどの浮動小数点命令は浮動小数点レジスタファイルを操作します。
浮動小数点ロード命令とストア命令は、浮動小数点値をレジスタとメモリ間で転送します。
また、整数レジスタファイルとの間で値を転送するための命令も提供されます。

整数と浮動小数点値の統一されたレジスタファイルを考えました。これは、ソフトウェアレジスタの割り当てと呼び出し規約を単純化し、ユーザーの全状態を減らすためです。
ただし、分割組織では、指定された命令幅でアクセス可能なレジスタの総数が増加し、広いスーパスカラ問題に対応する十分な regfile ポートの準備が簡素化され、分離浮動小数点ユニットアーキテクチャがサポートされ、内部浮動小数点エンコーディングテクニックの使用が簡素化されます。
分割レジスタファイルアーキテクチャのコンパイラサポートと呼び出し規約は十分に理解されており、浮動小数点レジスタファイル状態でダーティビットを使用すると、コンテキストスイッチのオーバーヘッドを減らすことができます。

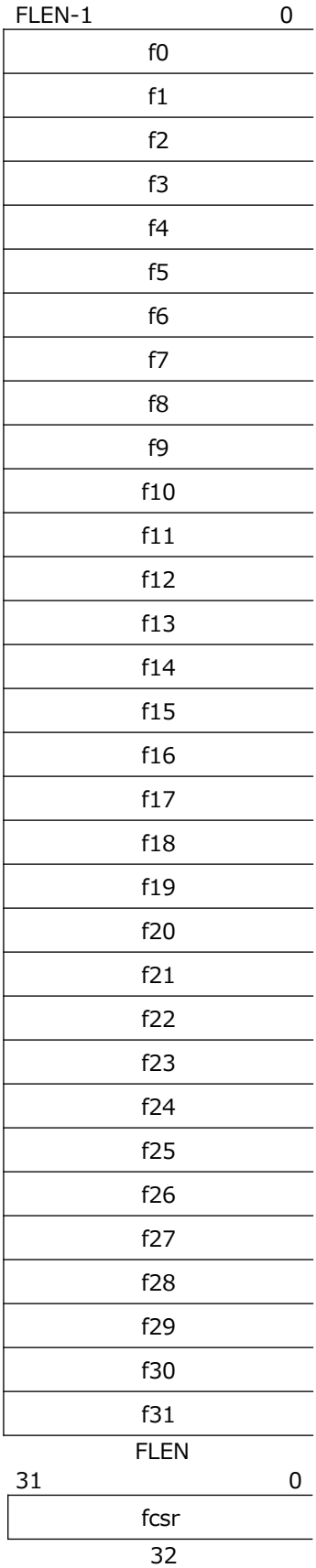


図 8.1：RISC-V 標準 F 拡張単精度浮動小数点状態

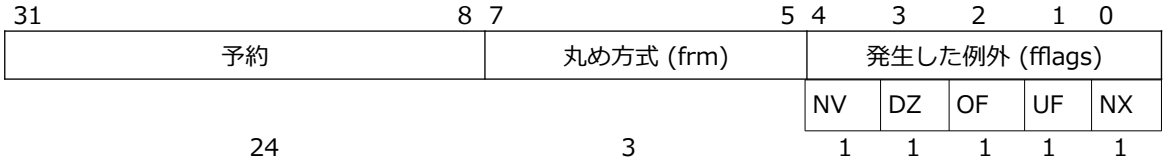


図 8.2 : 浮動小数点制御およびステータスレジスタ。

8.2 浮動小数点制御および状態レジスタ

浮動小数点制御および状態レジスタ fcsr は、RISC-V 制御および状態レジスタ（CSR）です。
図 8.2 に示すように、浮動小数点算術演算の動的丸めモードを選択し、発生した例外フラグを保持する 32 ビットの読取り/書込みレジスタです。

fcsr レジスタは、基になる CSR アクセス命令で構築されたアセンブラ疑似操作である FRCSR 命令と FSCSR 命令で読み書きできます。
FRCSR は fcsr を整数レジスタ rd にコピーすることによってそれを読み取ります。
FSCSR は、元の値を整数レジスタ rd にコピーし、次に整数レジスタ rs1 から取得した新しい値を fcsr に書き込むことによって、fcsr の値を入れ替えます。

fcsr 内のフィールドは、異なる CSR アドレスを介して個別にアクセスすることもでき、これらのアクセスに対して別々のアセンブラ擬似操作が定義されます。
FRRM 命令は、丸めモードフィールド frm を読み取り、それを整数レジスタ rd の最下位 3 ビットにコピーします。他のすべてのビットはゼロです。
FSRM は、元の値を整数レジスタ rd にコピーし、整数レジスタ rs1 の最下位 3 ビットから得られた新しい値を frm に書き込むことによって、frm の値を交換します。
FRFLAGS と FSFLAGS は、未払い例外フラグフィールド fflags と同様に定義されています。
↑Accrued って Google 翻訳とかだと 未払いとか未収ってなるけど、Webilo で辞書ひくと accure 生じる、発生する となる。
↑発生した例外フィールド くらいでいいかな
追加の疑似命令 FSRMI と FSFLAGSI は、レジスタ rs1 の代わりに即値を使用して値を交換します。

fcsr のビット 31-8 は、10 進浮動小数点の "L"標準拡張を含む他の標準拡張のために予約されています。
これらの拡張が存在しない場合、実装はこれらのビットへの書き込みを無視し、読み込み時にゼロの値を供給しなければならない。
標準ソフトウェアはこれらのビットの内容を保持する必要があります。

浮動小数点演算は、命令でエンコードされた静的な丸めモードまたは frm に保持された動的な丸めモードのいづれかを使用します。
丸めモードは、表 8.1 に示すようにエンコードされます。
命令の rm フィールドの値が 111 の場合、frm に保持されている動的丸めモードが選択されます。
frm が無効な値（101-111）に設定されている場合、その後の動的丸めモードで浮動小数点演算を実行しようとすると、不正な命令トラップが発生します。
rm フィールドを持ついくつかの命令は、丸めモードの影響をそれにも関わらず受けません(受けないのもあります)。 rm フィールドを RNE（000）に設定する必要があります。

C99 言語規格は、動的丸めモードレジスタの提供を効果的に義務づけています。

--2018/05/26

丸めモード	ニーモニック	意味
000	RNE	最も近いものへの丸め、等しいにくくる
001	RTZ	ゼロに向かって丸め
010	RDN	切り捨て (−∞方向)
011	RUP	切り上げ (+∞方向)
100	RMM	最も近いものへの丸め、最大の大きさにくくる
101		無効、将来使用するために予約
110		無効、将来使用するために予約
111		命令の rm フィールド、動的丸めモード選択 丸めモードレジスタ、無効

表 8.1：丸めモードのエンコーディング

発生した例外フラグは、表 8.2 に示すように、フィールドがソフトウェアによって最後にリセットされた後の浮動小数点算術命令で発生した例外条件を示します。

フラグ ニーモニック	フラグの意味
NV	無効な操作
DZ	ゼロ除算
OF	オーバーフロー
UF	アンダーフロー
NX	不正確

表 8.2：発生した例外フラグの符号化。

標準で許可されているように、我々はベース ISA の浮動小数点例外のトラップはサポートしていませんが、ソフトウェアのフラグを明示的にチェックする必要があります。
浮動小数点が発生した例外フラグの内容によって直接制御される分岐を追加することを検討しましたが、最終的に ISA を単純に保つためにこれらの命令を省略することを選択しました。

8.3 NaN の生成と伝播

↑NaN って 確か不定状態のことだったよな

特に明記されている場合を除いて、浮動小数点演算の結果が NaN の場合、正規 NaN になります。
正規の NaN は正の符号を持ち、MSB 以外のすべての符号ビットをクリアします。つまり、静かなビットです。
単精度浮動小数点の場合、これはパターン 0x7fc00000 に対応します。

FMIN と FMAX の場合、少なくとも 1 つの入力がシグナリング NaN である場合、または両方の入力が静かな NaN の場合、結果は正規の NaN になります。
1 つのオペランドが静かな NaN であり、もう 1 つが NaN でない場合、結果は非 NaN オペランドになります。

サイン インジェクション命令（FSGNJ、FSGNJN、FSGNJX）は NaN を正規化しません。 基本ビットパターンを直接操作します。

標準で推奨されているように、私たちは NaN ペイロードを伝播すると考えましたが、この決定はハードウェアコストを増加させるでしょう。
さらに、この機能は標準ではオプションなので、移植可能なコードでは使用できません。

実装者は、非標準動作モードで有効にされた非標準拡張として NaN ペイロード伝播スキームを自由に提供できます。
ただし、上記の正規の NaN スキームは常にサポートされていなければならず、既定のモードでなければなりません。

私たちは例外レベルの場合には、標準レベルのデフォルト値を返す実装が必要です。これは、ユーザレベルのソフトウェアの部分に何ら介入しなくても可能です。（アルファ ISA 浮動小数点トラップバリアとは異なります）。
私たちは例外的なケースの完全なハードウェア処理がより一般的になると考えており、他のアプローチを最適化するためにユーザーレベルの ISA を複雑にしないようにしたいと考えています。
実装は、例外的なデフォルト値を提供するために、常にマシンモードのソフトウェアハンドラにトラップすることができます。

8.4 非正常な算術

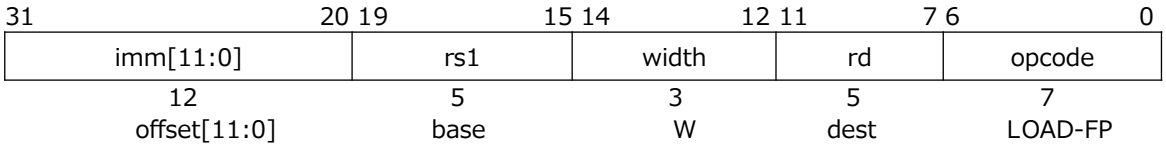
非正規数の演算は、IEEE 754-2008 規格に従って処理されます。

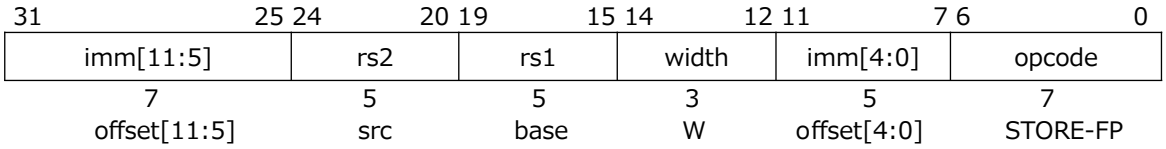
IEEE 標準の言い回しでは、丸めた後に微小が検出されます。
↑ tininess って とても小さいという意味みたいだが、どう訳すのがいいかな

丸めた後に微小を検出すると、スプリアス アンダー フロー信号が少なくなります。

8.5 単精度ロード命令とストア命令

浮動小数点ロードとストアは、整数ベース ISA と同じベース+オフセットアドレッシングモードを使用し、レジスタ rs1 のベースアドレスと 12 ビット符号付きバイトオフセットを使用します。
FLW 命令は、メモリから浮動小数点レジスタ rd に単精度浮動小数点値をロードします。
FSW は、浮動小数点レジスタ rs2 から単精度値をメモリに格納します。





FLW と FSW は、実効アドレスが自然に整列されている場合にのみ原子的に実行されることが保証されています。

8.6 単精度浮動小数点計算命令

1 つまたは 2 つのソースオペランドを使用する浮動小数点算術命令は、OP-FP の主要なオペコードで R 型形式を使用します。
FADD.S、FSUB.S、FMUL.S、および FDIV.S は、それぞれ rs1 と rs2 の単精度浮動小数点加算、減算、乗算、および除算を実行し、結果を rd に書き込みます。
FMIN.S および FMAX.S は、それぞれ rs1 および rs2 のうちの小さい方または大きい方をそれぞれ rd に書き込みます。
FSQRT.S は rs1 の平方根を計算し、結果を rd に書き込みます。

2 ビット浮動小数点フォーマットフィールド fmt は、表 8.3 に示すようにエンコードされます。
F 拡張のすべての命令について S（00）に設定されます。

2 ビット浮動小数点フォーマットフィールド fmt は、表 8.3 に示すようにエンコードされます。
F 拡張のすべての命令について S（00）に設定されます。

Fmt フィールド	ニーモニック	意味
00	S	32bit 単精度
01	D	64bit 倍精度
10	-	予約
11	Q	128bit 4 倍精度

表 8.3：フォーマットフィールドのエンコード

丸めを実行するすべての浮動小数点演算は、表 8.1 に示すエンコーディングの rm フィールドを使用して丸めモードを選択できます。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	S	src2	src1	RM	dest	OP-FP
FMUL/FDIV	S	src2	src1	RM	dest	OP-FP
FMIN-MAX	S	src2	src1	MIN/MAX	dest	OP-FP
FSQRT	S	0	src	RM	dest	OP-FP

浮動小数点の融合乗加算命令は、新しい標準命令フォーマットを必要とします。
↑融合乗加算って 乗算と加算が一緒になったっていう意味だよね。つまり積和演算のことね。昔は DSP ならではの命令だった
R4 型命令は、3 つのソースレジスタ（rs1、rs2、rs3）とデスティネーションレジスタ（rd）を指定します。
この形式は、浮動小数点の融合乗加算命令でのみ使用されます。
融合乗加算命令は、rs1 と rs2 の値を乗算し、オプションで積を否定し、rs3 の値を加算または減算して最終結果を rd に書き込みます。
FMADD.S は、rs1×rs2+rs3 を計算します。 FMSUB.S は、rs1×rs2-rs3 を計算します。 FNMSUB.S は、-rs1×rs2+rs3 を計算します。 FNMADD.S は-rs1×rs2-rs3 を計算します。

加算和が静かな NaN であっても、被乗数が∞とゼロの場合、融合乗加算命令は無効演算例外を発生させる必要があります。

IEEE 754-2008 規格では、操作∞×0+qNaN に対して無効な例外を発生させることはできますが、必要はありません。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
src3	S	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

8.7 単精度浮動小数点変換および移動命令

浮動小数点から整数への変換命令および整数から浮動小数点への変換命令は、OP-FP メジャーオペコード空間で符号化されています。
FCVT.W.S または FCVT.L.S は、浮動小数点レジスタ rs1 の浮動小数点数を、整数レジスタ rd の符号付き 32 ビットまたは 64 ビット整数にそれぞれ変換します。
FCVT.S.W または FCVT.S.L は、整数レジスタ rs1 の 32 ビットまたは 64 ビットの符号付き整数をそれぞれ浮動小数点レジスタ rd の浮動小数点数に変換します。
FCVT.WU.S、FCVT.LU.S、FCVT.S.WU、および FCVT.S.LU バリアントは、符号なし整数値に変換されます。
FCVT.L [U] .S と FCVT.S.L [U]は RV32 では不正です。
丸められた結果が宛先フォーマットで表現できない場合、それは最も近い値にクリッピングされ、無効フラグがセットされます。
表 8.4 に、FCVT.int.S の有効な入力の範囲と無効な入力の動作を示します。

	FCVT.W.S	FCVT.WU.S	FCVT.L.S	FCVT.LU.S
最小有効入力（丸め後）	-2^{31}	0	-2^{63}	0
最大有効入力（丸め後）	$2^{31}-1$	$2^{32}-1$	$2^{63}-1$	$2^{64}-1$
範囲外の負入力時の出力	-2^{31}	0	-263	0
-∞時の出力	-2^{31}	0	-263	0
範囲外の正入力時の出力	$2^{31}-1$	$2^{32}-1$	$2^{63}-1$	$2^{64}-1$
+∞もしくは NaN 時の出力	$2^{31}-1$	$2^{32}-1$	$2^{63}-1$	$2^{64}-1$

表 8.4：浮動小数点から整数への変換のドメインと無効な入力の動作

すべての浮動小数点から整数への整数と浮動小数点への変換命令は、rm フィールドに従って丸めます。
浮動小数点レジスタは、例外を発生させない FCVT.S.W rd、x0 を使用して、浮動小数点の正のゼロに初期化することができます。

--2018/05/29

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.int.fmt	S	W[U]/L[U]	src1	RM	dest	OP-FP
FCVT.fmt.int	S	W[U]/L[U]	src1	RM	dest	OP-FP

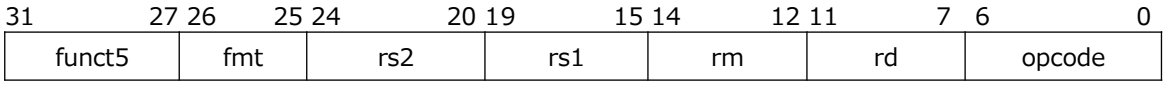
浮動小数点から浮動小数点への符号注入命令 FSGNJ.S、FSGNJN.S、および FSGNJX.S は、rs1 からの符号ビットを除くすべてのビットをとる結果を生成します。
FSGNJ の場合、結果の符号ビットは rs2 の符号ビットです。 FSGNJN の場合、結果の符号ビットは rs2 の符号ビットの反対です。
FSGNJX の場合、符号ビットは rs1 と rs2 の符号ビットの排他的論理和です。
サインイン命令では、浮動小数点例外フラグは設定されません。
注：FSGNJ.S rx,ry,ry は ry を rx に移動します（アセンブラ疑似オペレーション FMV.S rx,ry）。 FSGNJN.S rx,ry,ry は、ry の否定を rx に移動します（アセンブラ疑似演算 FNEG.S rx,ry）。FSGNJX.S rx,ry,ry は ry の絶対値を rx に移動します（アセンブラ疑似演算 FABS.S rx、ry）。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	S	src2	src1	J[N]/JX	dest	OP-FP

符号注入命令は、浮動小数点 MV、ABS、NEG を提供するだけでなく、超越数学関数ライブラリでの IEEE copySign 演算と符号操作を含むいくつかの他の演算をサポートします。
MV、ABS、および NEG は単一のレジスタ・オペランドだけを必要としますが、FSGNJ 命令は 2 つ必要ですが、ほとんどのマイクロアーキテクチャでは、これらの比較的まれな命令のためのレジスタ読取り回数の減少の恩恵を受けるために最適化を追加することはほとんどありません。
この場合であっても、マイクロアーキテクチャは、両方のソースレジスタが FSGNJ 命令で同じであることを単に検出し、単一のコピーのみを読み取ることができます。

浮動小数点レジスタと整数レジスタの間でビットパターンを移動するための命令が提供されています。
FMV.X.W は、IEEE 754-2008 エンコーディングで表される浮動小数点レジスタ rs1 の単精度値を整数レジスタ rd の下位 32 ビットに移動します。
RV64 の場合、デスティネーションレジスタの上位 32 ビットは浮動小数点数の符号ビットのコピーで埋められます。
FMV.W.X は、IEEE 754-2008 標準エンコーディングでエンコードされた単精度値を、整数レジスタ rs1 の下位 32 ビットから浮動小数点レジスタ rd に移動します。
ビットは転送で変更されません。特に、非標準 NaN のペイロードは保持されます。

FMV.W.X および FMV.X.W 命令は、以前は FMV.S.X および FMV.X.S.と呼ばれていました。
W の使用は、それらの解釈を行わずに 32 ビットを動かす命令として、そのセマンティクスとより一貫しています。
NaN-ボクシングを定義した後、これはより明確になりました。
既存のコードを妨害しないように、W と S の両方のバージョンがツールでサポートされます。



31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FMV.X.W	S	0	src	000	dest	OP-FP
FMV.W.X	S	0	src	000	dest	OP-FP

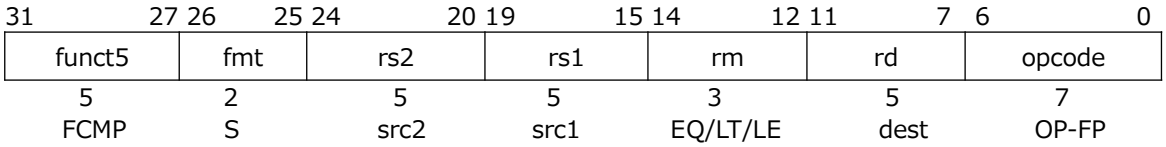
基本浮動小数点 ISA は、実装が浮動小数点フォーマットの内部再コーディングをレジスタに採用して、異常値の処理を簡素化し、場合によっては機能ユニットの待ち時間を短縮できるように定義されています。
この目的のために、ベース ISA は、整数レジスタファイルを直接読み書きする変換および比較演算を定義することによって、浮動小数点レジスタにおける整数値の表現を避けます。
これにより、整数と浮動小数点レジスタ間の明示的な移動が必要な一般的なケースの多くが削除され、一般的な混合フォーマットコードシーケンスの命令数とクリティカルパスが削減されます。

--2018/05/30

8.8 単精度浮動小数点比較命令

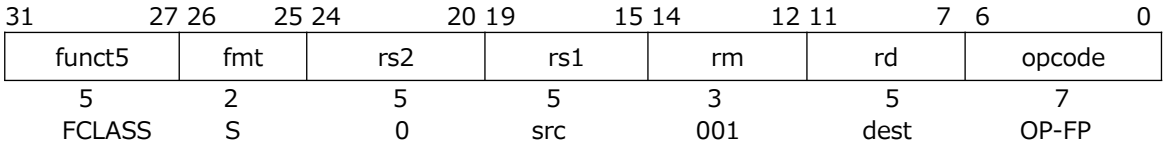
浮動小数点比較命令は、浮動小数点レジスタ rs1 と rs2 の間で指定された比較（等しい、小さい、または等しい）を実行し、ブール結果を整数レジスタ rd に記録します。

FLT.S および FLE.S は、IEEE 754-2008 標準がシグナリング比較と呼ぶものを実行します。つまり、どちらかの入力が NaN の場合、無効演算例外が発生します。
FEQ.S は静かな比較を実行します：シグナリング NaN 入力だけが無効な操作例外を引き起こします。
3 つの命令すべてについて、どちらかのオペランドが NaN の場合、結果は 0 になります。



8.9 単精度浮動小数点分類命令

FCLASS.S 命令は、浮動小数点レジスタ rs1 の値を調べ、浮動小数点数のクラスを示す 10 ビットのマスクを整数レジスタ rd に書き込みます。
マスクのフォーマットについては、表 8.5 を参照してください。
プロパティが真の場合は rd の対応するビットが設定され、それ以外の場合はクリアされます。
rd の他のビットはすべてクリアされます。
rd のちょうど 1 ビットが設定されることに注意してください。



rd bit	意味
0	rs1 は −∞
1	rs1 は 負の正規数
2	rs1 は 負の非正規数
3	rs1 は −0
4	rs1 は +0
5	rs1 は 正の非正規数
6	rs1 は 正の正規数
7	rs1 は +∞
8	rs1 は シグナリング NaN
9	rs1 は 静かな NaN

表 8.5：FCLASS 命令の結果のフォーマット

ー 2018/06/03

！浮動小数点まで来た。この後は倍精度とか 4 倍精度のだね
！ちょっと別件が入ったので、RISC-V 翻訳はペースダウンするのだ。
！やめるわけではないので、気を長〜くしてまってておくれ。
！

－1 ページ置いて次へ

第 9 章

倍精度浮動小数点の「D」標準拡張、バージョン 2.0

この章では、「D」という名前の標準倍精度浮動小数点命令セット拡張について説明し、IEEE 754-2008 算術標準に準拠した倍精度浮動小数点計算命令を追加します。
D 拡張は、基本単精度命令サブセット F に依存します。

9.1 D レジスタの状態

D 拡張は、32 の浮動小数点レジスタ f0～f31 を 64 ビットに広げます（図 8.1 の FLEN = 64）。
f レジスタは、9.2 節で後述するように、32 ビットまたは 64 ビットの浮動小数点値を保持できるようになりました。

FLEN は、F、D、および Q 拡張のどれがサポートされているかに応じて、32,64、または 128 にすることができます。
H、F、D、および Q を含む 4 つまでの異なる浮動小数点精度がサポートされています。
半精度 H スカラー値は、V ベクトル拡張がサポートされている場合にのみサポートされます。

より狭い値の 9.2 NaN のボクシング

複数の浮動小数点精度がサポートされている場合、NaN-ボクシングと呼ばれる処理では、n <FLEN の狭い n ビット型の有効な値が FLEN ビットの NaN 値の下位 n ビットに表されます。
有効な NaN ボックス値の上位ビットはすべて 1s でなければなりません。
したがって、有効な NaN ボックス n ビット値は、より広い m ビット値（n <m≦FLEN）と見なされるとき、負の静穏 NaN（qNaN）として表示されます。

ソフトウェアは、浮動小数点レジスタに格納されている現在のデータタイプを知らないかもしれないが、レジスタ値を保存して復元することができなければならないため、より幅の広い演算を使用してより狭い値を転送した結果を定義する必要があります。
一般的なケースは、呼び出し先保存レジスタですが、バリアック、ユーザーレベルのスレッドライブラリ、仮想マシンの移行、およびデバッグなどの機能には、標準的な規則も望ましいものです。

浮動小数点 n ビット転送演算は、IEEE 標準フォーマットで保持されている外部値を f レジスタとの間で移動し、浮動小数点ロードおよびストア（FLn / FSn）および浮動小数点移動命令（FMV.n.X / FMV.X.n）を含みます。
n <FLEN の f レジスタへのより狭い n ビット転送は、宛先 f レジスタのすべての上位 FLEN-n ビットを 1 に設定することにより、有効な NaN ボックス値を作成します。
浮動小数点レジスタからのより狭い n ビット転送は、上位 FLEN-n ビットを無視してレジスタの下位 n ビットを転送します。

浮動小数点演算および符号インジェクション演算は、f レジスタに保持された FLEN ビット値に基づいて結果を計算します。
n <FLEN である狭い n ビット演算は、入力オペランドが NaN ボックスで正しくチェックされる、すなわち、すべての上位 FLEN-n ビットが 1 であることをチェックします。
そうであれば、入力の n 最下位ビットが入力値として使用され、それ以外の場合、入力値は n ビット正準 NaN として扱われます。
n ビットの浮動小数点結果は、デスティネーション f レジスタの最下位 n ビットに書き込まれ、すべて 1 が最上位の FLEN-n ビットに書き込まれ、正当な NaN ボックス値が生成されます。

整数から浮動小数点への変換（例えば、FCVT.S.X）は、FLEN ビットのデスティネーションレジスタを満たすために FLEN より狭い任意の結果を NaN ボックスに入れます。
より狭い n ビット浮動小数点値から整数（例えば、FCVT.X.S）への変換は正当な NaN ボクシングをチェックし、正当な n ビット値でない場合には入力を n ビット正準 NaN として扱います。

このドキュメントの以前のバージョンでは、幅の狭いオペランドの結果をより狭いオペランドの値を保存することを要求する以外は、より狭いオペランドまたはより広いオペランドの結果をオペレーションに渡す動作は定義されていませんでした。
新しい定義では、この実装固有の動作が削除されますが、浮動小数点ユニットのレコーディングされていない実装とレコード化された実装の両方に対応します。
新しい定義は、値が正しく使用されない場合に NaN を伝播することによってソフトウェアエラーを捕捉するのにも役立ちます。

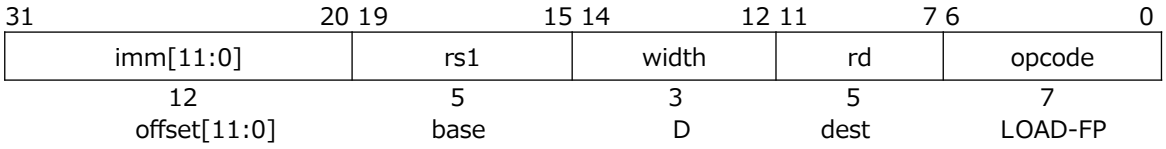
コード化されていない実装では、すべての浮動小数点演算の入力と出力にオペランドを IEEE 標準形式に展開してパックします。
非再構成の実装に対する NaN-ボクシングのコストは、主に、より狭い演算の上位ビットが正当な NaN-ボックス化された値を表しているかどうかをチェックし、結果の上位ビットにすべて 1 を書き込むことにあります。

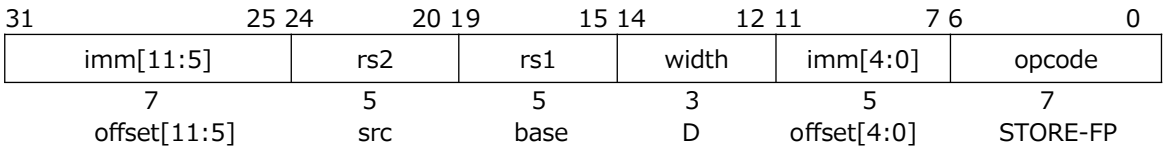
レコーディングされた実装では、浮動小数点値を表現するためのより便利な内部形式を使用し、すべての値を正規化して保持できるように指数ビットが追加されています。
レコード化された実装のコストは主に内部型と符号ビットを追跡するために必要な余分なタグ付けですが、これは指数部の内部で NaN をコード化することによって新しい状態ビットを追加することなく行うことができます。
記録フォーマットの内外で値を転送するために使用されるパイプラインには小さな変更が必要ですが、データパスとレイテンシのコストは最小限です。
レコーディング処理では、いずれの場合もワイドオペランドの入力正規化値のシフトを処理しなければならず、NaN ボックス値の抽出は、先頭の 0 ビットをスキップせずに先頭の 1 ビットをスキップする以外は正規化と同様の処理です。データパス多重化を共有することができます。

9.3 倍精度ロード命令とストア命令

FLD 命令は、メモリから浮動小数点レジスタ rd に倍精度浮動小数点値をロードします。
FSD は浮動小数点レジスタの倍精度値をメモリに格納します。

倍精度値は、NaN ボックス化された単精度値です。

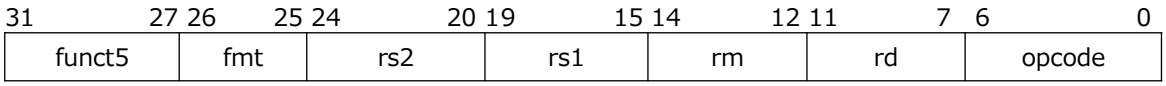




FLD と FSD は、実効アドレスが自然にアライメントされ、XLEN≧64 の場合にのみ原子的に実行されることが保証されています。

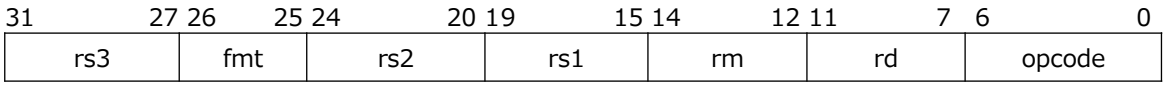
9.4 倍精度浮動小数点計算命令

倍精度浮動小数点計算命令は、その単精度対応と同様に定義されますが、倍精度オペランドで動作し、倍精度結果を生成します。



5	2	5	5	3	5	7
FADD/FSUB	D	src2	src1	RM	dest	OP-FP
FMUL/FDIV	D	src2	src1	RM	dest	OP-FP
FMIN-MAX	D	src2	src1	MIN/MAX	dest	OP-FP
FSQRT	D	0	src	RM	dest	OP-FP

↑単精度浮動小数点とほぼいっしょ D のところだけ違う



5	2	5	5	3	5	7
src3	D	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

9.5 倍精度浮動小数点変換および移動命令

浮動小数点から整数への変換命令および整数から浮動小数点への変換命令は、OP-FP メジャーオペコード空間で符号化されます。
FCVT.W.D または FCVT.L.D は、浮動小数点レジスタ rs1 の倍精度浮動小数点数を、整数レジスタ rd の符号付き 32 ビットまたは 64 ビット整数にそれぞれ変換します。
FCVT.D.W または FCVT.D.L は、整数レジスタ rs1 の 32 ビットまたは 64 ビットの符号付き整数をそれぞれ浮動小数点レジスタ rd の倍精度浮動小数点数に変換します。

FCVT.WU.D、FCVT.LU.D、FCVT.D.WU、およびFCVT.D.LUバリアントは、符号なし整数値に変換されます。
FCVT.L [U] .Dと FCVT.D.L [U]は RV32では不正です。
FCVT.int.Dの有効な入力の範囲と無効な入力の動作は、FCVT.int.Sの場合と同じです。

すべての浮動小数点から整数への整数と浮動小数点への変換命令は、rmフィールドに従って丸めます。
注 FCVT.D.W [U]は常に正確な結果を生成し、丸めモードの影響を受けません。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.int.D	D	W[U]/L[U]	src	RM	dest	OP-FP
FCVT.D.int	D	W[U]/L[U]	src	RM	dest	OP-FP

倍精度から単精度および単精度から倍精度への変換命令 FCVT.S.D および FCVT.D.S は、OP-FP メジャーオペコード空間でエンコードされ、ソースおよびデスティネーションの両方が浮動小数点レジスタです。
rs2 フィールドはソースのデータ型をエンコードし、fmt フィールドはデスティネーションのデータ型をエンコードします。
FCVT.S.D は RM フィールドに従って丸めます。 FCVT.D.S は決してラウンドしません。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.S.D	S	D	src	RM	dest	OP-FP
FCVT.D.S	D	S	src	RM	dest	OP-FP

浮動小数点から浮動小数点への符号インジェクション命令 FSGNJ.D、FSGNJN.D、および FSGNJX.D は、単精度符号インジェクション命令と同様に定義されています。

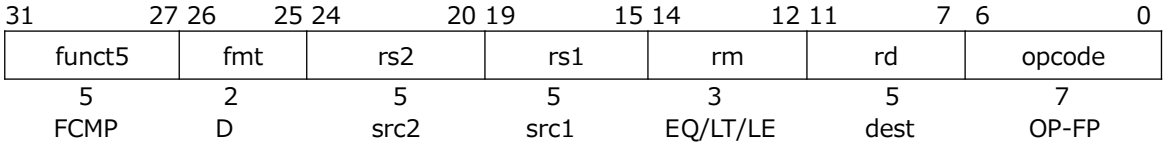
31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	D	src2	src1	J[N]/JX	dest	OP-FP

RV64の場合のみ、浮動小数点レジスタと整数レジスタの間でビットパターンを移動するための命令が提供されます。
FMV.X.D は、浮動小数点レジスタ rs1 の倍精度値を整数レジスタ rd の IEEE 754-2008 標準エンコーディングの表現に移動します。
FMV.D.X は、IEEE 754-2008 標準エンコーディングでエンコードされた倍精度値を整数レジスタ rs1 から浮動小数点レジスタ rd に移動します。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FMV.X.D	D	0	src	000	dest	OP-FP
FMV.D.X	D	0	src	000	dest	OP-FP

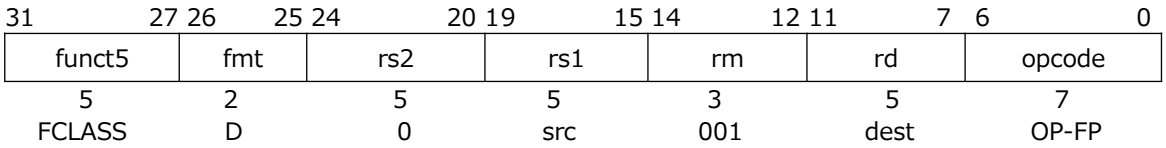
9.6 倍精度浮動小数点比較命令

倍精度浮動小数点比較命令は、単精度対応と同様に定義されますが、倍精度オペランドで動作します。



9.7 倍精度浮動小数点分類命令

倍精度浮動小数点分類命令 FCLASS.D は、単精度対応と同様に定義されますが、倍精度オペランドで動作します。



-- 2018/07/08

-- 1 ページ空き、次ページへ

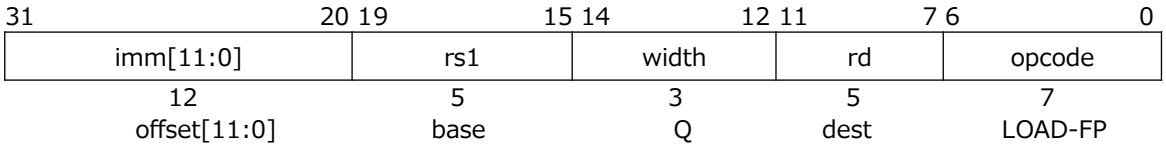
第 10 章

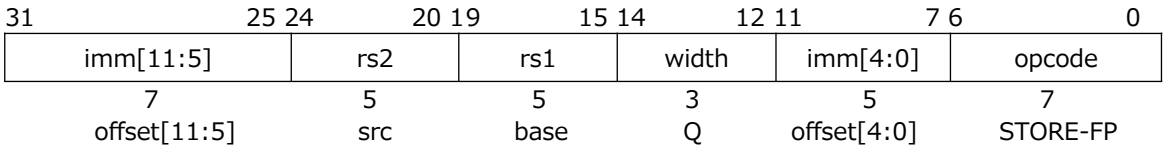
クワッド精密浮動小数点の「Q」標準拡張、バージョン 2.0

この章では、IEEE 754-2008 算術標準に準拠した 128 ビットバイナリ浮動小数点命令の Q 標準拡張について説明します。
128 ビットまたは 4 倍精度の 2 進浮動小数点命令サブセットの名前は "Q"で、RV64IFD が必要です。
浮動小数点レジスタは、単精度、倍精度、または 4 倍精度浮動小数点値（FLEN = 128）のいずれかを保持するように拡張されました。
セクション 9.2 で説明した NaN-boxing スキームが再帰的に拡張され、倍精度値内で単精度値が NaN ボックス化されるようになりました。倍精度値自体は 4 倍精度値の NaN ボックス化されています。

10.1 四倍精度ロード命令とストア命令

新しい 128 ビットの LOAD-FP 命令と STORE-FP 命令が追加され、funct3 幅フィールドの新しい値がエンコードされます。





FLQ と FSQ は、実効アドレスが自然にアライメントされ、XLEN = 128 の場合にのみアトミックに実行されることが保証されています。

10.2 クワッド精密計算命令

表 10.1 に示すように、サポートされている新しいフォーマットがほとんどの命令のフォーマットフィールドに追加されます。

Fmt フィールド	ニーモニック	意味
00	S	32bit 単精度
01	D	64bit 倍精度
10	-	予約
11	Q	128bit 4 倍精度

表 10.1：フォーマットフィールドのエンコード

4 倍精度浮動小数点計算命令は、その倍精度対応部と同様に定義されますが、4 倍精度オペランドで動作し、4 倍精度結果を生成します。

31 27 26 25 24 20 19 15 14 12 11 7 6 0

funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	Q	src2	src1	RM	dest	OP-FP
FMUL/FDIV	Q	src2	src1	RM	dest	OP-FP
FMIN-MAX	Q	src2	src1	MIN/MAX	dest	OP-FP
FSQRT	Q	0	src	RM	dest	OP-FP

↑単精度浮動小数点、倍精度浮動小数点とほぼいっしょ Q のところだけ違う

31 27 26 25 24 20 19 15 14 12 11 7 6 0

rs3	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
src3	D	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

10.3 四倍精度変換および移動命令

31 27 26 25 24 20 19 15 14 12 11 7 6 0

funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.S.Q	Q	W[U]/L[U]	src	RM	dest	OP-FP
FCVT.Q.S	Q	W[U]/L[U]	src	RM	dest	OP-FP

新しい浮動小数点から浮動小数点への変換命令 FCVT.S.Q、FCVT.Q.S、FCVT.D.Q、FCVT.Q.D が追加されました。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.S.Q	S	Q	src	RM	dest	OP-FP
FCVT.Q.S	Q	S	src	RM	dest	OP-FP
FCVT.D.Q	D	Q	src	RM	dest	OP-FP
FCVT.Q.D	Q	D	src	RM	dest	OP-FP

浮動小数点から浮動小数点への符号インジェクション命令、FSGNJ.Q、FSGNJN.Q、および FSGNJX.Q は、倍精度符号インジェクション命令と同様に定義されています。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	Q	src2	src1	J[N]/JX	dest	OP-FP

FMV.X.Q および FMV.Q.X 命令は提供されていないため、4 倍精度のビットパターンをメモリ経由で整数レジスタに移動する必要があります。

RV128 は Q 拡張で FMV.X.Q と FMV.Q.X をサポートしています。

10.4 4 倍精度浮動小数点比較命令

浮動小数点比較命令は、浮動小数点レジスタ rs1 と rs2 の間で指定された比較（等しい、小さい、または等しい）を実行し、ブール結果を整数レジスタ rd に記録します。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCMP	Q	src2	src1	EQ/LT/LE	dest	OP-FP

10.5 四倍精度浮動小数点分類命令

4 倍精度浮動小数点分類命令 FCLASS.Q は、その倍精度対応と同様に定義されますが、4 倍精度オペランドで動作します。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCLASS	Q	0	src	001	dest	OP-FP

-- 1 ページ空き、次ページへ

第 11 章

小数点浮動小数点バージョン「L」の標準拡張、バージョン 0.0

この章は、IEEE 754-2008 規格で定義されているように 10 進浮動小数点演算をサポートするように設計された "L"という標準拡張の仕様のプレースホルダーです。

11.1 10 進浮動小数点レジスタ

既存の浮動小数点レジスタは 64 ビットおよび 128 ビットの 10 進浮動小数点値を保持するために使用され、既存の浮動小数点ロードおよびストア命令はメモリ間で値を移動するために使用されます。

融合乗加算命令で要求されるオペコード空間が大きいため、10 進浮動小数点命令拡張では、30 ビットの符号化空間に 5 つの 25 ビットメジャーオペコードが必要になります。

-- 1 ページ空き、次ページへ

第 12 章

圧縮された命令のための "C"標準拡張、バージョン 2.0

この章では、一般的な操作に短い 16 ビットの命令エンコーディングを追加することにより、静的および動的コードサイズを削減する、RISC-V 標準圧縮命令セット拡張「C」の現在のドラフト提案について説明します。
C 拡張は、基本 ISA（RV32、RV64、RV128）のいずれかに追加することができ、これらのいずれかをカバーするために総称「RVC」を使用します。
通常、プログラム内の RISC-V 命令の 50%～60%を RVC 命令に置き換えることができるため、コードサイズを 25%～30%削減できます。

12.1 概要

RVC は以下の場合に、一般的な 32 ビット RISC-V 命令のより短い 16 ビットバージョンを提供する簡単な圧縮方式を使用します。
・即時オフセットまたはアドレス指定が小さいか、または
・レジスタの 1 つがゼロレジスタ（x0）、ABI リンクレジスタ（x1）、または ABI スタックポインタ（x2）、または
・デスティネーションレジスタと第 1 のソースレジスタが同一であるか、または
・使用されるレジスタは 8 つの最も一般的なレジスタです。

C 拡張は、他のすべての標準命令拡張と互換性があります。
C 拡張は、16 ビット命令を 32 ビット命令と自由に混在させることができ、後者は任意の 16 ビット境界で開始することができます。
C 拡張を追加すると、JAL 命令と JALR 命令は命令のミスアライン例外を発生しなくなります。

オリジナルの 32 ビット命令で 32 ビット境界整列制約を削除すると、コード密度が大幅に向上します。

圧縮された命令エンコーディングは RV32C、RV64C、RV128C で共通ですが、表 12.3 に示すように、いくつかのオペコードはベース ISA の幅に応じて異なる目的で使用されます。
たとえば、RV32C は同じオペコードを使用して単精度浮動小数点値のロードと格納を圧縮するのに対して、64 ビット整数値のロードと格納を圧縮するには、より広いアドレス空間の RV64C と RV128C バリアントが追加のオペコードを必要とします。
同様に、RV128C では、128 ビット整数値のロードと格納をキャプチャするために追加のオペコードが必要ですが、RV32C と RV64C の倍精度浮動小数点値のロードと格納には同じオペコードが使用されます。
C 拡張が実装されている場合は、関連する標準浮動小数点拡張（F および/または D）も実装されているときは常に、適切な圧縮浮動小数点ロードおよびストア命令を提供する必要があります。
さらに、RV32C には、RV64C および RV128C の ADDIW を圧縮するために同じオペコードが使用される、短距離サブルーチンコールを圧縮する圧縮ジャンプおよびリンク命令が含まれています。

倍精度のロードとストアは、静的命令と動的命令のかなりの部分であり、RV32C と RV64C のエンコーディングにそれらを含めることが動機となります。

現在サポートされている ABI 用にコンパイルされたベンチマークでは、単精度ロードおよびストアは静的または動的圧縮の重要なソースではありませんが、ハードウェア単精度浮動小数点ユニットのみを提供し、単精度のロードとストアは、少なくとも倍精度ロードと同じくらい頻繁に使用され、測定されたベンチマークに格納されます。
したがって、RV32C での圧縮サポートを提供する動機です。

短距離サブルーチンコールは、マイクロコントローラの小さなバイナリで可能性が高いため、これらを RV32C に含める動機付けがあります。

さまざまなベースレジスタ幅に対して異なる目的でオペコードを再利用するとドキュメンテーションが複雑になりますが、複数のベース ISA レジスタ幅をサポートするデザインであっても実装の複雑さへの影響は小さいです。
圧縮された浮動小数点ロードおよびストア・バリアントは、より広い整数ロードおよびストアと同じレジスタ指定子を有する同じ命令フォーマットを使用します。

RVC は、各 RVC 命令がベース ISA（RV32I / E、RV64I、または RV128I）または存在する場合は F および D 標準拡張のいずれかで単一の 32 ビット命令に展開されるという制約の下で設計されました。
この制約を採用すると、2 つの主な利点があります。

- ハードウェア設計では、デコード中に RVC 命令を拡張するだけで、検証を簡素化し、既存のマイクロアーキテクチャの変更を最小限に抑えることができます。
- コンパイラは RVC 拡張を認識せずに、アセンブラとリンカにコード圧縮を残すことができますが、圧縮認識コンパイラは一般的により良い結果を生み出すことができます。

私たちは、C と基本 IFD 命令との間の単純な 1 対 1 マッピングの複数の複雑さの削減が、追加の命令は C 拡張、または 1 つの C 命令で複数の IFD 命令のエンコードを許可した命令でのみサポートされてた、わずかに高密度なエンコーディングの潜在的な利益をはるかに上回ることを感じました

C 拡張はスタンドアロン ISA として設計されておらず、ベース ISA と一緒に使用されることに注意することが重要です。

コード長を改善するために、可変長命令セットが長く使用されてきました。
たとえば、1950 年代後半に開発された IBM Stretch [6]には、32 ビットおよび 64 ビット命令を含む ISA がありました。ここでは、32 ビット命令の一部は完全な 64 ビット命令の圧縮バージョンでした。
Stretch は、インデックスレジスタの 1 つのみを参照できる短い分岐命令を使用して、より短い命令フォーマットのいくつかでアドレス可能なレジスタのセットを制限するという概念も採用しました。
後の IBM 360 アーキテクチャ[3]は、16 ビット、32 ビット、または 48 ビットの命令フォーマットを使用した単純な可変長命令エンコーディングをサポートしていました。

1963 年に CDC は RISC アーキテクチャの前身である Cray 設計の CDC 6600 [28]を導入しました。このアーキテクチャでは、2 つの長さ、15 ビットと 30 ビットの命令を持つレジスタ豊富なロードストアアーキテクチャが導入されました。
後の Cray-1 デザインは、16 ビットと 32 ビットの命令長を持つ非常に似た命令フォーマットを使用していました。

1980 年代の初期の RISC ISA は、ワークステーション環境では合理的で、コードサイズよりも性能が優れていましたがしかし組込みシステムでは合理的でありませんでした。
したがって、ARM と MIPS は、その後、標準の 32 ビット幅の命令の代わりに代替の 16 ビット幅の命令セットを提供することにより、より小さなコードサイズを提供する ISA のバージョンを作成しました。
圧縮された RISC ISA は、開始ポイントに対して約 25～30%のコードサイズを縮小し、80x86 よりも大幅に小さいコードを生成しました。
この結果は、可変長 CISC ISA が 16 ビットおよび 32 ビットフォーマットのみを提供していた RISC ISAs よりも小さくなければならないということだったため、いくつかのことを驚かせました。

元の RISC ISAs は、これらの計画外圧縮命令を含むために十分なオペコード空間を自由に残さなかったので、完全な新しい ISA として開発されました。
これは、コンパイラが独立した圧縮 ISA 用に異なるコードジェネレータを必要とすることを意味していました。
最初の圧縮 RISC ISA 拡張（例えば、ARM Thumb および MIPS16）は固定の 16 ビット命令サイズのみを使用しており、静的なコードサイズは大幅に削減されましたが、動的な命令数が増加をもたらし、元の固定幅の 32 ビット命令サイズと比較してパフォーマンスが低下しました。
これにより、パフォーマンスは純粋な 32 ビット命令に似ていましたが、(たとえば、ARM Thumb2、microMIPS、PowerPC VLE など)、16 ビットと 32 ビットの混在した命令長の圧縮 RISC ISA デザインの第 2 世代が開発され大幅なコードサイズが削減されました。
残念なことに、圧縮された ISA のこれらの異なる世代は、相互に互換性がなく、元の圧縮されていない ISA と互換性がなく、ドキュメンテーション、実装、およびソフトウェアツールのサポートが大幅に複雑になります。

一般的に使用されている 64 ビット ISA のうち、現在 PowerPC と microMIPS のみが圧縮命令フォーマットをサポートしています。
スタティックコードサイズと動的命令フェッチ帯域幅が重要なメトリックであるため、モバイルプラットフォーム（ARM v8）で最も一般的な 64 ビット ISA に圧縮命令フォーマットが含まれていないことは驚くべきことです。
大規模システムでは静的コードサイズは大きな問題ではありませんが、商用ワークロードを実行しているサーバーでは命令フェッチ帯域幅が大きなボトルネックになりがちです。

25 年間の見通しから知見を得て、RISC-V は、最初から圧縮された命令をサポートするように設計されており、RVC がベース ISA の上に簡単に拡張されるように（多くの他の拡張機能と共に）追加されています。
RVC の哲学は、組み込みアプリケーションのコードサイズを削減し、命令キャッシュのミスが少なくなるため、すべてのアプリケーションのパフォーマンスとエネルギー効率を向上させることです。
Waterman は、RVC が命令ビットを 25%～30%削減し、命令キャッシュミスを 20%～25%削減すること、または命令キャッシュサイズを 2 倍にするのとほぼ同じパフォーマンスに影響することを示しています[33]。

12.2 圧縮された命令フォーマット

表 12.1 に 8 つの圧縮命令フォーマットを示します。
CR、CI、CSS は 32 個の RVI レジスタを使用できますが、CIW、CL、CS、CB はわずか 8 個に制限されています。
表 12.2 にレジスタ x8～x15 に対応する一般的なレジスタを示します。
スタックポインタをベースアドレスレジスタとして使用するロード命令とストア命令は別々のバージョンが存在することに注意してください。スタックへの保存と復元は非常に一般的であるため、CI および CSS 形式を使用して 32 のすべてのデータレジスタにアクセスできるようにします。
CIW は、ADDI4SPN 命令のために 8 ビットの即値を供給します。

RISC-V ABI は頻繁に使用されるレジスタをレジスタ x8～x15 にマップするように変更されました。
これにより、連続して自然にアライメントされたレジスタ番号のセットを有することによって解凍デコーダが簡略化され、16 個の整数レジスタしか持たない RV32E サブセットベース仕様とも互換性があります。

圧縮されたレジスタベースの浮動小数点ロードおよびストアでは、CL および CS フォーマットもそれぞれ使用され、8 つのレジスタが f8 から f15 にマッピングされます。

標準の RISC-V 呼び出し規約では、頻繁に使用される浮動小数点レジスタをレジスタ f8～f15 にマップします。これにより、整数レジスタ番号と同じレジスタ圧縮解除デコードが可能になります。

このフォーマットは、すべての命令の同じ場所に 2 つのレジスタソース指定子のビットを保持するように設計されていて、宛先レジスタフィールドは移動することができます。
完全な 5 ビットデスティネーションレジスタ指定子が存在する場合、それは 32 ビット RISC-V エンコーディングと同じ場所にあります。
イミディエイトが符号拡張されている場合、符号拡張は常にビット 12 から始まります。
イミディエイトフィールドは、ベース仕様のようにスクランブルされており、必要な即時マルチプレクサの数を減らしています。

即値フィールドは、順次命令ではなく命令フォーマットでスクランブルされるので、可能な限り多くのビットがすべての命令の同じ位置にあり、それにより実装が単純化される。
例えば、即値ビット 17-10 は、常に同じ命令ビット位置から供給される。
5 つの他の即値ビット（5,4,3,1 および 0）は、わずか 2 つのソース命令ビットを有し、4 つ（9,7,6 および 2）は 3 つのソースを有し、1 つ（8）は 4 つのソースを有します。

多くの RVC 命令では、ゼロ値のイミディエイトは禁止され、x0 は有効な 5 ビットのレジスタ指定子ではありません。
これらの制約は、より少ないオペランドビットを必要とする他の命令のための符号化空間を解放します。

– 2018/07/15

Format	Meaning	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	Register	funct4		rd/rs1		rs2		op
CI	Immediate	funct3	imm	rd/rs1		imm		op
CSS	Stack-relative Store	funct3	imm			rs2		op
CIW	Wide Immediate	funct3	imm				rd’	op
CL	Load	funct3	imm		rs1’	imm	rd’	op
CS	Store	funct3	imm		rs1’	imm	rs2’	op
CB	Branch	funct3	offset		rs1’	offset		op
CJ	Jump	funct3	Jump target					op

表 12.1：圧縮された 16 ビット RVC 命令フォーマット。

RVC Register Number	0000	001	010	011	100	101	110	111
Integer Register Number	x8	x9	x10	x11	x12	x13	x14	x15
Integer Register ABI Name	s0	s1	a0	a1	a2	a3	a4	a5
Floating-Point Register Number	f8	f9	f10	f11	f12	f13	f14	f15
Floating-Point Register ABI Name	fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

表 12.2 CIW、CL、CS、および CB フォーマットの 3 ビット rs1 '、rs2'、および rd 'フィールドで指定されるレジスタ。

12.3 ロードとストア命令

16 ビット命令のリーチを増加させるために、データ転送命令はバイト単位のデータサイズでスケーリングされたゼロ拡張命令を使用します。ワードは x4、ダブルワードは x8、クワッドワードは x16 です。

RVC には、ロードとストアの 2 種類があります。
ABI スタック・ポインタ x2 をベース・アドレスとして使用し、任意のデータ・レジスタをターゲットにすることができます。
もう 1 つは 8 つのベースアドレスレジスタの 1 つと 8 つのデータレジスタの 1 つを参照できます。

スタックポインタベースのロードとストア

15 13	12	11 7	6 2	1 0
funct3	imm	rd	imm	op
3	1	5	5	2
C.LWSP	offset[5]	dest≠0	offset[4:2][7:6]	C2
C.LDSP	offset[5]	dest≠0	offset[4:3][8:6]	C2
C.LQSP	offset[5]	dest≠0	offset[4][9:6]	C2
C.FLWSP	offset[5]	dest	offset[4:2][7:6]	C2
C.FLDSP	offset[5]	dest	offset[4:3][8:6]	C2

これらの命令は CI 形式を使用します。

C.LWSP はメモリからレジスタ rd に 32 ビット値をロードします。
スタックポインタ x2 に 4 でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。
lw rd, offset [7:2]（x2）に展開されます。

C.LDSP は、メモリからレジスタ rd に 64 ビット値をロードする RV64C / RV128C 専用命令です。
スタックポインタ x2 に 8 でスケーリングされたゼロ拡張オフセットを加算することによって、実効アドレスを計算します。
ld rd, offset [8:3]（x2）に展開されます。

C.LQSP は、メモリからレジスタ rd に 128 ビットの値をロードする RV128C 専用の命令です。
スタックポインタ x2 に 16 でスケーリングされたゼロで拡張されたオフセットを加算することによって、実効アドレスを計算します。
lq rd、offset [9:4]（x2）に展開されます。

C.FLWSP は、メモリから浮動小数点レジスタ rd に単精度浮動小数点値をロードする RV32FC 専用命令です。
スタックポインタ x2 に 4 でスケーリングしたゼロ拡張オフセットを加算することにより、実効アドレスを計算します。
それは flw rd, offset [7:2]（x2）に展開されます。

C.FLDSP は、メモリから浮動小数点レジスタ rd に倍精度浮動小数点値をロードする RV32DC / RV64DC 専用命令です。
スタックポインタ x2 に 8 でスケーリングされたゼロ拡張オフセットを加算することによって、実効アドレスを計算します。
fld rd, offset [8:3]（x2）に展開されます。

15 13	12 7	6 2	1 0
funct3	imm	rs2	op
3	6	5	2
C.SWSP	offset[5:2][7:6]	src	C2
C.SDSP	offset[5:3][8:6]	src	C2
C.SQSP	offset[5:4][9:6]	src	C2
C.FSWSP	offset[5:2][7:6]	src	C2
C.FSDSP	offset[5:3][8:6]	src	C2

これらの手順では、CSS 形式を使用します。

C.SWSP はレジスタ rs2 の 32 ビット値をメモリに格納します。
スタックポインタ x2 に 4 でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。
sw rs2, offset [7:2]（x2）に展開されます。

C.SDSP は、レジスタ rs2 の 64 ビット値をメモリに格納する RV64C / RV128C 専用の命令です。
これは、ゼロで拡張されたオフセットを 8 でスケーリングしたものをスタックポインタ x2 に加算することによって実効アドレスを計算します。
sd rs2、offset [8 : 3]（x2）に展開されます。

C.SQSP は、レジスタ rs2 の 128 ビット値をメモリに格納する RV128C 専用の命令です。
これは、スタックポインター x2 にゼロで拡張されたオフセットを 16 でスケーリングしたものを加算することによって実効アドレスを計算します。
sq rs2, offset [9:4]（x2）に展開されます。

C.FSWSP は浮動小数点レジスタ rs2 の単精度浮動小数点値をメモリに格納する RV32FC 専用命令です。
スタックポインタ x2 に 4 でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。
fsw rs2, offset [7:2]（x2）に展開されます。

C.FSDSP は、浮動小数点レジスタ rs2 の倍精度浮動小数点値をメモリに格納する RV32DC / RV64DC 専用の命令です。
これは、ゼロで拡張されたオフセットを 8 でスケーリングしたものをスタックポインタ x2 に加算することによって実効アドレスを計算します。
fsd rs2, offset [8:3]（x2）に展開されます。

– 2018/07/16

関数の入口/出口でのレジスタ保存/復元コードは、静的コードサイズのかなりの部分を表します。
RVC のスタックポインタベースの圧縮ロードとストアは、動的命令帯域幅を削減してパフォーマンスを向上させながら、セーブ/リストアスタティックコードサイズを 2 倍に減らすのに有効です。

他の ISA で保存/復元コードのサイズをさらに減らすために使用される一般的なメカニズムは、ロード・マルチプルおよびストア・マルチプル・インストラクションです。
RISC-V にこれらを採用することを検討しましたが、これらの手順には次のような欠点がありました。：

・これらの命令はプロセッサの実装を複雑にします。
・仮想メモリシステムでは、一部のデータアクセスが物理メモリに常駐する可能性があり、実行できないものもあります。これは部分的に実行される命令に対して新しい再起動メカニズムを必要とします。
・残りの RVC 命令とは異なり、複数ロードと複数ストアに相当する IFD はありません。
・RVC 命令の残りの部分とは異なり、コンパイラは、命令を生成し、それらを保存して格納する可能性を最大にするためにレジスタを割り当てるために、これらの命令を認識しなければならず、それらは保存され、順番に復元されるからです。
・シンプルなマイクロアーキテクチャの実装は、負荷の周りに他の命令をスケジューリングして複数の命令を格納する方法を制限し、潜在的なパフォーマンスの低下につながります。
・シーケンシャルなレジスタ割り付けの要望は、CIW、CL、CS、および CB フォーマット用に選択された機能レジスタと競合する可能性があります。

さらに、プロローグとエピローグコードをサブルーチンコールで共通のプロローグとエピローグコードに置き換えることで、多くの利益をソフトウェアで実現することができます。これは[34]のセクション 5.6 に記載されています。

合理的な設計者は異なる結論に至るかもしれませんが、私たちはロードとストアを省略し、代わりに最大のコードサイズの縮小を達成するために保存/復元ミリコードルーチンを呼び出すソフトウェアのみのアプローチを使用することにしました。

レジスタベースのロードおよびストア

15 13	12 10	9 7	6 5	4 2	1 0
funct3	imm	rs1′	imm	rd′	op
3	3	3	2	3	2
C.LW	offset[5:3]	base	offset[2|6]	dest	C0
C.LD	offset[5:3]	base	offset[7:6]	dest	C0
C.LQ	offset[5|4|8]	base	offset[7:6]	dest	C0
C.FLW	offset[5:3]	base	offset[2|6]	dest	C0
C.FLD	offset[5:3]	base	offset[7:6]	dest	C0

これらの命令は、CL 形式を使用します。

C.LW はメモリからレジスタ rd'に 32 ビット値をロードします。
レジスタ rs1'のベースアドレスに 4 でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。
lw rd'、offset [6：2]（rs1'）に展開されます。

C.LD は、メモリからレジスタ rd'に 64 ビット値をロードする RV64C / RV128C 専用命令です。
レジスタ rs1'のベースアドレスに 8 でスケーリングしたゼロ拡張オフセットを加算して実効アドレスを計算します。
ld rd'、offset [7：3]（rs1'）に展開されます。

C.LQ は、メモリからレジスタ rd'に 128 ビットの値をロードする RV128C 専用命令です。
レジスタ rs1'のベースアドレスに 16 でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。
lq rd'、offset [8：4]（rs1'）に展開されます。

C.FLW は、単精度浮動小数点値をメモリから浮動小数点レジスタ rd'にロードする RV32FC 専用命令です。
レジスタ rs1'のベースアドレスに 4 でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。
fld rd'、offset [6：2]（rs1'）に展開されます。

C.FLD は、メモリから浮動小数点レジスタ rd'に倍精度浮動小数点値をロードする RV32DC / RV64DC 専用命令です。
レジスタ rs1'のベースアドレスに 8 でスケーリングしたゼロ拡張オフセットを加算して実効アドレスを計算します。
fld rd'、offset [7：3]（rs1'）に展開されます。

15 13	12 10	9 7	6 5	4 2	1 0
funct3	imm	rs1′	imm	rs2′	op
3	3	3	2	3	2
C.SW	offset[5:3]	base	offset[2|6]	src	C0
C.SD	offset[5:3]	base	offset[7:6]	src	C0
C.SQ	offset[5|4|8]	base	offset[7:6]	src	C0
C.FSW	offset[5:3]	base	offset[2|6]	src	C0
C.FSD	offset[5:3]	base	offset[7:6]	src	C0

これらの命令は、CS フォーマットを使用します。

C.SW はレジスタ rs2'の 32 ビット値をメモリに格納します。
レジスタ rs1'のベースアドレスに 4 でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。
sw rs2'、offset [6：2]（rs1'）に展開されます。

C.SD は、レジスタ rs2'の 64 ビット値をメモリに格納する RV64C / RV128C 専用の命令です。
レジスタ rs1'のベースアドレスに 8 でスケーリングしたゼロ拡張オフセットを加算して実効アドレスを計算します。
sd rs2'、offset [7：3]（rs1'）に展開されます。

C.SQ はレジスタ rs2'の 128 ビット値をメモリに格納する RV128C 専用命令です。
レジスタ rs1'のベースアドレスに 16 でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。
sq rs2'、offset [8：4]（rs1'）に展開されます。

C.FSW は浮動小数点レジスタ rs2'の単精度浮動小数点値をメモリに格納する RV32FC 専用命令です。
レジスタ rs1'のベースアドレスに 4 でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。
fsw rs2'、offset [6：2]（rs1'）に展開されます。

C.FSD は、浮動小数点レジスタ rs2'の倍精度浮動小数点値をメモリに格納する RV32DC / RV64DC 専用の命令です。
レジスタ rs1'のベースアドレスに 8 でスケーリングしたゼロ拡張オフセットを加算して実効アドレスを計算します。
fsd rs2'、offset [7：3]（rs1'）に展開されます。

--
C.LW ↔ C.SW のように ロードとストアで対応した命令になっている。

12.4 制御転送命令

RVC は無条件ジャンプ命令と条件付き分岐命令を提供します。
基本 RVI 命令の場合と同様に、すべての RVC 制御転送命令のオフセットは 2 バイトの倍数になります。

15	13 12	2 1	0
funct3	imm	op	
3	11	2	
C.J	offset[11|4|9:8|10|6|7|3:1|5]	C1	
C.JAL	offset[11|4|9:8|10|6|7|3:1|5]	C1	

これらの命令はCJ形式を使用します。

C.J は無条件制御転送を行います。
オフセットは符号拡張され、ジャンプターゲットアドレスを形成するために PC に追加されます。
従って、C.J は±2KiB の範囲を対象とすることができます。
C.J は jal x0、offset [11：1]に展開されます。

C.JAL は C.J と同じ操作を実行するが、リンクレジスタ x1 にジャンプ（pc + 2）に続く命令のアドレスを追記する RV32C 専用命令です。
C.JAL は jal x1、offset [11：1]に展開されます。

15	12 11	7 6	2 1	0
funct4	rs1	rs2	op	
4	5	5	2	
C.JR	src≠0	0	C2	
C.JALR	src≠0	0	C2	

These instructions use the CR format.

C.JR (jump register) performs an unconditional control transfer to the address in register rs1.
C.JR expands to jalr x0, rs1, 0.

C.JALR (jump and link register) performs the same operation as C.JR, but additionally writes the address of the instruction following the jump (pc+2) to the link register, x1.
C.JALR expands to jalr x1, rs1, 0.

厳密に言えば、C.JALR は基本 RVI 命令に正確には拡張されません。これは、リンクアドレスを形成するために PC に追加される値が、基本 ISA の場合のように 4 ではなく 2 であるため、2 バイトと 4 バイトの両方のオフセットをサポートすることは、ベースのマイクロアーキテクチャーにはごくわずかな変更です。

15	13 12	10 9	7 6	2 1	0
funct3	imm	rs1′	imm	op	
3	3	3	5	2	
C.BEQZ	offset[8|4:3]	src	offset[7:6|2:1|5]	C1	
C.BNEZ	offset[8|4:3]	src	offset[7:6|2:1|5]	C1	

これらの命令はCB形式を使用します。

C.BEQZ は、条件付き制御転送を実行する。
オフセットは符号拡張され、PC に追加されて分岐先アドレスを形成します。
したがって、±256B の範囲をターゲットにすることができます。
C.BEQZ は、レジスタ rs1 'の値がゼロの場合に分岐を取ります。
beq rs1 '、x0、offset [8：1]に展開されます。

C.BNEZ も同様に定義されますが、rs1 'にゼロ以外の値が含まれている場合は分岐が必要です。
bn rs1 '、x0、offset [8：1]に展開されます。

12.5 整数計算命令

RVC は、整数演算および定数生成のためのいくつかの命令を提供します。

整数定数生成命令

2 つの定数生成命令は両方とも CI 命令フォーマットを使用し、任意の整数レジスタを対象とすることができます。

15 13	12	11 7	6 2	1 0
funct3	imm[5]	rd	Imm[4:0]	op
3	1	5	5	2
C.LI	imm[5]	dest≠0	imm[4:0]	C1
C.LUI	nzuimm[17]	dest≠{0,2}	nzuimm[16:12]	C1

C.LI は、符号拡張された 6 ビットの即値 imm をレジスタ rd にロードします。
C.LI は、rd≠x0 のときのみ有効です。
C.I.I は、追加、x0、imm [5：0]に展開されます。

C.LUI は、デスティネーションレジスタのビット 17-12 に非ゼロの 6 ビット即値フィールドをロードし、下位 12 ビットをクリアし、ビット 17 をデスティネーションのすべての上位ビットに符号拡張します。
C.LUI は、rd≠{x0、x2}のときとイミディエートがゼロに等しくないときにのみ有効です。
C.LUI は lui rd、nzuimm [17:12]に拡張されます。

整数レジスタ - 即値演算

これらの整数レジスタ即値操作は、CI 形式でエンコードされ、非 x0 整数レジスタおよび 6 ビット即値に対して操作を実行します。
即値はゼロにすることはできません。

15 13	12	11 7	6 2	1 0
funct3	imm[5]	rd/rs1	Imm[4:0]	op
3	1	5	5	2
C.ADDI	nzimm[5]	dest	nzimm[4:0]	C1
C.ADDIW	imm[5]	dest≠0	imm[4:0]	C1
C.ADDI16SP	nzimm[9]	2	nzimm[4|6|8:7|5]	C1

-- nzimm って non zero immediate で ゼロ即値じゃないってことかな

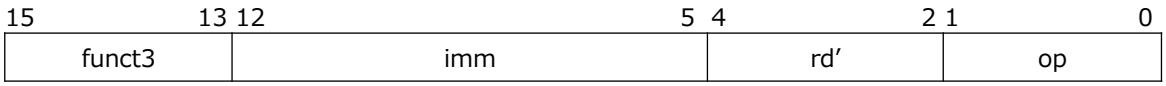
C.ADDI は、非ゼロ符号拡張 6 ビット即値をレジスタ rd の値に加算し、その結果を rd に書き込みます。
C.ADDI は、addi rd、rd、nzimm [5：0]に拡大します。

C.ADDIW は RV64C / RV128C のみの命令で、同じ計算を実行しますが 32 ビットの結果を生成し、結果を 64 ビットに符号拡張します。
C.ADDIW は、追加、rd、imm [5：0]に拡張されます。
即値は C.ADDIW の場合はゼロであり、これは sext.w rd に対応します。

C.ADDI16SP はオペコードを C.LUI と共有しますが、宛先フィールドは x2 であす。
C.ADDI16SP は、非ゼロ符号拡張 6 ビット即値をスタック・ポインタ（sp = x2）の値に加算します。ここで、イミディエートは、範囲内の 16 の倍数（-512,496）を表すようにスケーリングされます。
C.ADDI16SP は、プロシージャのプロローグとエピローグでスタックポインタを調整するために使用されます。
addi x2、x2、nzimm [9：4]に展開されます。

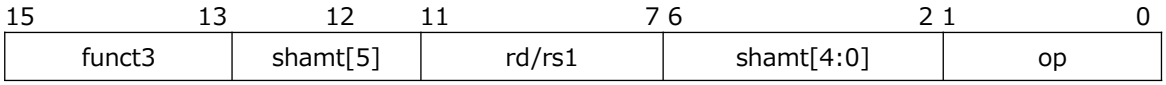
-- プロローグ、エピローグ っていうのがいまいちわからない、何をさすんだろう。
-- -512,496 って 16 進で FFF8_2E10、正としても 0007_D1F0 なのでなんか中途半端。どっから出てきた数字だろう。
- あ、いや違った、もしかすると -512 と 496 かな、FE00 と 01F0 なのでこっちが正解っぽい。496 は 512-16 だし。

標準的な RISC-V 呼び出し規約では、スタックポインタ sp は常に 16 バイトに整列しています。



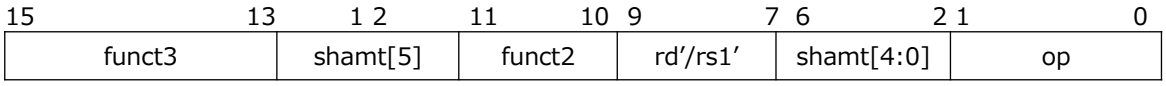
3	8	3	2
C.ADDI4SPN	nzuimm[5:4|9:6|2|3]	dest	C0

C.ADDI4SPN は、CIW 形式の RV32C / RV64C のみの命令で、ゼロで拡張された非ゼロの即値を 4 でスケーリングした値をスタックポインタ x2 に加算し、その結果を rd’に書き込みます。
この命令は、スタック割り当て変数へのポインタを生成するために使用され、addi rd0、x2、nzuimm [9：2]に展開されます。



3	1	5	5	2
C.SLLI	shamt[5]	dest≠0	shamt[4:0]	C2

C.SLLI は、レジスタ rd の値の論理左シフトを実行し、結果を rd に書き込む CI 形式の命令です。
シフト量は shamt フィールドにエンコードされます。shamt [5]は RV32C ではゼロでなければなりません。
RV32C および RV64C の場合、シフト量はゼロでない必要があります。
RV128C では、64 のシフトをエンコードするためにシフト量ゼロが使用されます。
C.SLLI は、shamt = 0 の RV128C を除いて、slli rd、rd、shamt [5：0]に展開され、slli rd、rd、64 に展開されます。



3	1	2	3	5	2
C.SRLI	shamt[5]	C.SRLI	dest	shamt[4:0]	C1
C.SRAI	shamt[5]	C.SRAI	dest	shamt[4:0]	C1

C.SRLI は、レジスタ rd 'の値の論理右シフトを実行し、その結果を rd'に書き込む CB 形式の命令です。
シフト量は shamt フィールドにエンコードされます。shamt [5]は RV32C ではゼロでなければなりません。
RV32C および RV64C の場合、シフト量はゼロでない必要があります。
RV128C では、64 のシフトをエンコードするためにシフト量ゼロが使用されます。
さらに、シフト量は RV128C に対して符号拡張されているため、有効なシフト量は 1-31,64,96-127 です。
C.SRLI は、srli rd '、rd'、shamt [5：0]に展開され、ただし shamt = 0 の RV128C は srli rd '、rd'、64 に展開されます。

C.SRAI は C.SRLI と同様に定義されますが、代わりに算術右シフトを実行します。
C.SRAI は srai rd '、rd'、shamt [5：0]に展開されます。

左シフトは、通常、右シフトより頻繁で、左シフトはアドレス値を拡大するために頻繁に使用されるためです。
したがって、右シフトはより少ない符号化空間が与えられ、他のすべての直後が符号拡張された符号化象限に配置されます。
RV128 の場合、即座に 6 ビットシフト量を符号拡張することが決定されました。
デコードの複雑さを軽減することは別として、128 ビットアドレスポインタの上位部分に配置されたタグの抽出を可能にするために、96-127 の右シフト量が 64-95 よりも有用であると考えています。
RV128C は、RV32C および RV64C と同じポイントでフリーズしないため、128 ビットのアドレス空間コードの一般的な使用を評価できます。

15 13	12	11 10	9 7	6 2	1 0
funct3	imm[5]	funct2	rd′/rs1′	imm[4:0]	op
3	1	2	3	5	2
C.ANDI	imm[5]	C.ANDI	dest	imm[4:0]	C1

C.ANDI は、レジスタ rd 'の値と符号拡張された 6 ビットの即値とのビット単位の論理積を計算し、その結果を rd'に書き込む CB 形式の命令です。
C.ANDI は andi rd '、rd'、imm [5：0]に展開されます。

整数レジスタ - レジスタ演算

15 12	11 7	6 2	1 0
funct4	rd/rs1	rs2	op
4	5	5	2
C.MV	dest≠0	src≠0	C2
C.ADD	dest≠0	src≠0	C2

これらの命令は CR 形式を使用します。

C.MV はレジスタ rs2 の値をレジスタ rd にコピーします。
C.MV は、add rd、x0、rs2 に展開される。

C.ADD は、レジスタ rd と rs2 の値を加算し、結果をレジスタ rd に書き込みます。
C.ADD は、add rd、rd、rs2 に展開されます。

15 10	9 7	6 5	4 2	1 0
funct6	rd'/rs′	funct	rs2′	op
6	3	2	3	2
C.AND	dest	C.AND	src	C1
C.OR	dest	C.OR	src	C1
C.XOR	dest	C.XOR	src	C1
C.SUB	dest	C.SUB	src	C1
C.ANDW	dest	C.ANDW	src	C1
C.SUBW	dest	C.SUBW	src	C1

これらの命令は、CS フォーマットを使用します。

C.AND はレジスタ rd 'と rs20 の値のビット単位の AND を計算し、その結果をレジスタ rd'に書き込みます。
C.AND は rd '、rd'、rs2 'に展開されます。

C.OR はレジスタ rd 'と rs2'の値のビットごとの論理和を計算し、結果をレジスタ rd 'に書き込みます。
C.OR は rd '、rd'、rs2 'に展開されます。

C.XOR は、レジスタ rd 'と rs2'の値のビットごとの XOR を計算し、結果をレジスタ rd 'に書き込みます。
C.XOR は xor rd '、rd'、rs2 'に展開されます。

C.SUB は、レジスタ rd 'の値をレジスタ rd'の値から減算し、その結果をレジスタ rd 'に書き込む。
C.SUB は sub rd '、rd'、rs2 'に展開されます。

C.ADDW は、レジスタ rd 'と rs2'に値を加算した後、加算結果の下位 32 ビットを符号拡張して結果をレジスタ rd 'に書き込む RV64C / RV128C 専用の命令です。
C.ADDW は addw rd '、rd'、rs2 'に展開されます。

C.SUBW は、レジスタ rd 'の値をレジスタ rd'の値から減算し、その結果の下位 32 ビットを符号拡張して結果をレジスタ rd 'に書き込む RV64C / RV128C 専用命令です。
C.SUBW は subw rd '、rd'、rs2 'に展開されます。

この 6 つの命令群は、個々に大きな節約を提供するものではなく、多くのエンコードスペースを占有せず、実装が簡単であり、静的および動的圧縮で価値のある改善を提供します。

定義違法命令

15 13	12	11 7	6 2	1 0
0	0	0	0	0
3	1	5	5	2
0	0	0	0	0

すべてのビットがゼロである 16 ビット命令は、不当命令として永久に予約されています。

メモリ空間のゼロエンドまたは存在しない部分を実行しようとする試みをトラップするのに役立つ不正な命令であることをすべてゼロ命令で予約します。
すべてゼロでない値は、非標準拡張では再定義されるべきではありません。
同様に、存在しないメモリ領域に見られる別の共通の値をキャプチャするには、すべてのビットが 1（RISC-V 可変長符号化方式の非常に長い命令に対応）に設定された命令を不正なものとして予約します。

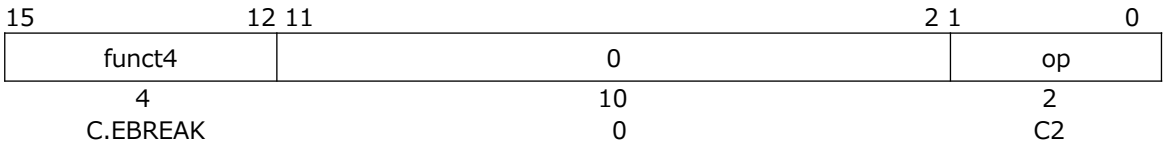
NOP 命令

15 13	12	11 7	6 2	1 0
funct3	imm[5]	rd/rs1	imm[4:0]	op
3	1	5	5	2
C.NOP	0	0	0	C1

C.NOP は、PC を前進させることを除いて、ユーザーが見ることのできる状態を変更しない CI 形式の命令です。
C.NOP は c.addi x0、0 として符号化され、addi x0、x0、0 に展開されます。

その意味は、A 拡張と C 拡張の両方をサポートすると主張する実装は、有効な C 命令を含む LR / SC シーケンスが最終的に完了することを保証しなければならないということです。

ブレークポイント命令



デバッガは、デバッグ環境にコントロールを戻すために、ebreak に展開される C.EBREAK 命令を使用することができます。
C.EBREAK はオペコードを C.ADD 命令と共有しますが、rd と rs2 が両方ともゼロであるため、CR 形式も使用できます。

12.6 LR / SC シーケンスにおける C 命令の使用

C 拡張をサポートする実装では、セクション 7.2 で説明したように、最終的な成功の保証を保持しながら、LR / SC シーケンス内で許可された I 命令の圧縮形式を使用できます。

12.7 RVC 命令セットリスト

表 12.3 に、RVC の主要なオペコードのマップを示します。
下位 2 ビットがセットされたオペコードは、基本 ISA の命令を含む 16 ビットよりも広い命令に対応する。
いくつかの命令は特定のオペランドに対してのみ有効です。 無効な場合、オペコードが将来の標準拡張のために予約されていることを示す RES、 オペコードが非標準拡張用に予約されていることを示す NSE。 オペコードが将来の標準マイクロアーキテクチャヒントのために予約されていることを示すために HINT を使用します。
HINT とマークされた命令は、ヒントが効果を持たない実装では no-ops として実行する必要があります。

HINT 命令は、パフォーマンスに影響する可能性があるがアーキテクチャ状態に影響を与えないマイクロアーキテクチャヒントの将来の追加をサポートするように設計されています。
単純な実装が HINT エンコーディングを無視し、HINT をアーキテクチャ状態を変更しない通常の操作として実行できるように、HINT エンコーディングが選択されています。
たとえば、宛先レジスタが x0 の場合、C.ADD は HINT で、5 ビットの rs2 フィールドは HINT の詳細をエンコードします。
しかし、簡単な実装では、HINT をレジスタ x0 への加算として単に実行するだけで、効果はありません。

Inst[15:13] Inst[1:0]	000	001	010	011	100	101	110	111	
00	ADDISPN	FLD FLD LQ	LW	FLW LD LD	Reserved	FSD FSD SQ	SW	FSW SD SD	RV32 RV64 RV128
01	ADDI	JAL ADDIW ADDIW	LI	LUI/ADDI16SP	MISC-ALU	J	BEQZ	BNEZ	RV32 RV64 RV128
10	SLLI	FLDSP FLDSP LQ	LWSP	FLWSP LDSP LDSP	J[AL]R/MV/ADD	FSDSP FSDSP SQ	SWSP	FSWSP SDSP SDSP	RV32 RV64 RV128
11	>16b								

表 12.3：RVC オペコードマップ

表 12.4-12.6 に、RVC 命令を示します。

15–13	12–11–10–9–8–7	6–5	4–3–2	1–0	
000	0		0	00	Illegal instruction
000	Nzuimm[5:4|9:6|2|3]		rd′	00	C.ADDI4SPN(RES,nzuimm=0)
001	uimm[5:3]	uimm[7:6]	rd′	00	C.FLD(RV32/64)
001	Uimm[5:4|8]	uimm[7:6]	rd′	00	C.LQ(RV128)
010	uimm[5:3]	uimm[2|6]	rd′	00	C.LW
011	uimm[5:3]	uimm[2|6]	rd′	00	C.FLW(RV32)
011	uimm[5:3]	uimm[7:6]	rd′	00	C.LD(RV64/128)
100	—			00	Reserved
101	uimm[5:3]	uimm[7:6]	rs2′	00	C.FSD(RV32/64)
101	uimm[5:4|8]	uimm[7:6]	rs2′	00	C.SQ(RV128)
110	uimm[5:3]	uimm[2|6]	rs2′	00	C.SW
111	uimm[5:3]	uimm[2|6]	rs2′	00	C.FSW(RV32)
111	uimm[5:3]	uimm[7:6]	rs2′	00	C.SD(RV64/128)

表 12.4：RVC、Quadrant 0 の命令一覧

15–13	12	11–10	9–8–7	6–5	4–3–2	1–0	
000	0	0		0		01	C.NOP
000	nzimm[5]	rs1/rd≠0		nzimm[4:0]		01	C.ADDI(HINT,nzimm=0)
001	imm[11|4|9:8|10|6|7|3:1|5]					01	C.JAL(RV32)
001	imm[5]	rs1/rd≠0		imm[4:0]		01	C.ADDIW(RV64/128;RES,rd=0)
010	imm[5]	rd≠0		imm[4:0]		01	C.LI(HINT,rd=0)
011	nzimm[9]	2		nzimm[4|6|8:7|5]		01	C.ADDI16SP(RES,nzimm=0)
011	nzimm[17]	Rd≠{0,2}		nzimm[16:12]		01	C.LUI(RES,nzimm=0;HINT,rd=0)
100	nzuimm[5]	00	rs1′/rd′	nzuimm[4:0]		01	C.SRLI(RV32 NSE,nzuimm[5]=1)
100	0	00	rs1′/rd′	0		01	C.SRLI64(RV128;RV32/64 HINT)
100	nzuimm[5]	01	rs1′/rd′	nzuimm[4:0]		01	C.SRAI(RV32 NSE,nzuimm[5]=1)
100	0	01	rs1′/rd′	0		01	C.SRAI64(RV128;RV32/64 HINT)
100	imm[5]	10	rs1′/rd′	imm[4:0]		01	C.ANDI
100	0	11	rs1′/rd′	00	rs2′	01	C.SUB
100	0	11	rs1′/rd′	01	rs2′	01	C.XOR
100	0	11	rs1′/rd′	10	rs2′	01	C.OR
100	0	11	rs1′/rd′	11	rs2′	01	C.AND
100	1	11	rs1′/rd′	00	rs2′	01	C.SUBW(RV64/128;RV32 RES)
100	1	11	rs1′/rd′	01	rs2′	01	C.ADDW(RV64/128;RV32 RES)
100	1	11	—	10	—	01	Reserved
100	1	11	—	11	—	01	Reserved
101	imm[11|4|9:8|10|6|7|3:1|5]					01	C.J
110	imm[8|4:3]		rs1′	imm[7:6|2:1|5]		01	C.BEQZ
111	imm[8|4:3]		rs1′	imm[7:6|2:1|5]		01	C.BNEZ

表 12.5：RVC、Quadrant 1 の命令一覧

15 14 13	12	11 10 9 8 7	6 5 4 3 2	1 0	
000	nzuimm[5]	rs1/rd≠0	nzuimm[4:0]	10	C.SLLI(HINT,rd=0;RV32 NSE,nzuimm[5]=1)
000	0	rs1/rd≠0	0	10	C.SLLI64(RV128;RV32/64 HINT; HINT, rd=0
001	uimm[5]	rd	uimm[4:3|8:6]	10	C.FLDSP(RV32/64)
001	uimm[5]	rd≠0	uimm[4|9:6]	10	C.LQSP(RV128;RES,rd=0)
010	uimm[5]	rd≠0	uimm[4:2][7:6]	10	C.LWSP(RES,rd=0)
011	uimm[5]	rd	uimm[4:2][7:6]	10	C.FLWSP(RV32)
011	uimm[5]	rd≠0	uimm[4:3][8:6]	10	ClLDSP(RV64/128;RES, rd=0)
100	0	rs≠0	0	10	C.JR(RES,rs1=0)
100	0	rd≠0	rs2≠0	10	C.MV(HINT, rd=0)
100	1	0	0	10	C.EBREAK
100	1	rs≠0	0	10	C.JALR
100	1	rs1/rd≠0	rs2≠0	10	C.ADD(HINT, rd=0)
101	uimm[5:3|8:6]		rs2	10	C.FSDSP(RV32/64)
101	uimm[5:3|8:6]		rs2	10	C.SQSP(RV128)
110	uimm[5:3|8:6]		rs2	10	C.SWSP
111	uimm[5:3|8:6]		rs2	10	C.FSWSP(RV32)
111	uimm[5:3|8:6]		rs2	10	C.SDSP(RV64/128)

表 12.6：RVC、Quadrant 2 の命令リスト

-- 1 ページ空き、次ページへ

第 13 章

"B"ビット操作のための標準拡張、バージョン 0.0

この章は、ビットフィールドの挿入、抽出、テストのための命令、回転、ファンネルのシフト、ビットとバイトの置換についての命令を含む、ビット操作命令を提供する将来の標準拡張のプレースホルダーです。

ビット操作命令は、特に外部的にパックされたデータ構造を扱う場合、いくつかのアプリケーション領域で非常に有効ですが、すべてのドメインで有用ではないためベースの ISA から除外し、必要なオペランドをすべて提供するための複雑さや命令フォーマットを追加することができます。

私たちは、B の拡張がベースの 30 ビット命令空間内でブラウンフィールド符号化になると予想しています。

-- 1 ページ空き、次ページへ

第 14 章

動的に翻訳された言語用の "J"標準拡張、バージョン 0.0

この章は、動的に翻訳された言語をサポートする将来の標準拡張のプレースホルダーです。

多くの一般的な言語は、通常、Java や Javascript を含む動的翻訳を介して実装されています。
これらの言語は、ダイナミックチェックとガベージコレクションのための追加の ISA サポートから恩恵を得ることができます。

-- 1 ページ空き、次ページへ

第 15 章

トランザクションメモリのための "T"標準拡張、バージョン 0.0

この章は、トランザクションメモリ操作を提供するための将来の標準拡張のプレースホルダです。

過去 20 年間にわたる多くの研究、および初期の商用実装にもかかわらず、複数のアドレスを含むアトミック操作をサポートする最良の方法についてはまだまだ議論があります。

私たちの現在の考えは、元のトランザクションメモリ提案のラインに沿って、限られた容量の小さなトランザクションメモリバッファを含めることです。

-- 1 ページ空き、次ページへ

第 16 章

パック℃-SIMD 命令の "P"標準拡張命令、バージョン 0.1

第 5 回 RISC-V ワークショップでの議論では、大規模な浮動小数点 SIMD 演算の V 拡張を標準化することを支持して、浮動小数点レジスタのこのパックド SIMD 提案を破棄したいとの要望が示されました。
しかし、小さな RISC-V 実装の整数レジスタで使用するためのパックド SIMD 固定小数点演算に関心がありました。

この章では、RISC-V 用の標準的なパックド SIMD 拡張の概要を説明します。
将来のパックド SIMD 拡張の標準セットに対して、命令サブセット名 "P"を予約しました。
他の拡張機能の多くは、整数ユニットとは別のワイドデータレジスタとデータパスを利用して、パックド SIMD 拡張を構築することができます。

Lincoln Labs TX-2 [9]で初めて導入された Packed-SIMD 拡張は、データ並列コードでより高いスループットを提供する一般的な方法となっています。
以前の商用マイクロプロセッサのインプリメンテーションには、Intel i860、HP PA-RISC MAX [19]、SPARC VIS [29]、MIPS MDMX [12]、PowerPC AltiVec [8]、Intel x86 MMX / SSE [24,26] が含まれていおり、最近のデザインには Intel x86 AVX [20]や ARM Neon [11]などが含まれています。
この章では、packed-SIMD を追加するための標準的なフレームワークについて説明しますが、このような設計には積極的に取り組んでいません。
我々の意見では、パックド SIMD 設計は、既存のワイド・データパス・リソースを再利用する際の妥当な設計ポイントであり、重要な追加リソースをデータ並列実行に専念させる場合は、従来のベクタ・アーキテクチャに基づく設計がより良い選択であり、V 拡張を使用する必要があります。

RISC-V パックド SIMD 拡張命令は、浮動小数点レジスタ（f0～f31）を再利用します。
これらのレジスタは、FLEN = 32～FLEN = 1024 の幅を持つように定義できます。
標準浮動小数点命令サブセットは、幅 32 ビット（ "F"）、64 ビット（2D "）、または 128 ビット（" Q "）のレジスタを必要とします。

浮動小数点レジスタは、パックド SIMD 値の代わりに使用するのが当然です、整数レジスタ（PA-RISC および Alpha パック SIMD 拡張）は、制御レジスタおよびアドレス値の整数レジスタを解放し、SIMD 浮動小数点実行のためのスカラー浮動小数点ユニットの再利用を簡素化し、デカップリングされた整数/浮動小数点ハードウェア設計に自然につながります。
浮動小数点ロードおよびストア命令エンコーディングはまた、より広いパックド SIMD レジスタを処理するための空間を有します。
ただし、パックド SIMD 値に浮動小数点レジスタを再利用すると、浮動小数点値のコード化された内部フォーマットを使用するのが難しくなります。

既存の浮動小数点ロード命令とストア命令は、メモリからfレジスタにさまざまなサイズのワードをロードして格納するために使用されます。
ベースISAは32ビットと64ビットのロードとストアをサポートしますが、LOAD-FP命令とSTORE-FP命令のエンコードでは、表16.1に示すように8種類の異なる幅をエンコードできます。
パックドSIMD操作で使用する場合、ハードウェアに非自然にアライメントされたロードおよびストアをサポートすることが望ましいです。

フィールド幅	コード	ビットサイズ
000	B	8
001	H	16
010	W	32
011	D	64
100	Q	128
101	Q2	256
110	Q4	512
111	Q8	1024

表16.1：LOAD-FPとSTORE-FPの幅エンコーディング

パックドSIMD計算命令は、f個のレジスタにパックされた値で動作します。
各値は、8ビット、16ビット、32ビット、64ビット、または128ビットのいずれでもかまいません。また、整数と浮動小数点の両方の表現をサポートできます。
例えば、64ビットのパックドSIMD拡張は、各レジスタを1×64ビット、2×32ビット、4×16ビット、または8×8ビットのパックド値として扱うことができます。

単純なパックドSIMD拡張は、未使用の32ビット命令オペコードに収まるかもしれないが、より広範なパックドSIMD拡張は、おそらく専用の30ビット命令空間を必要とするでしょう。

-- 2018/08/05

第 17 章

ベクトル操作のための "V"標準拡張、バージョン 0.2