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

↑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）。

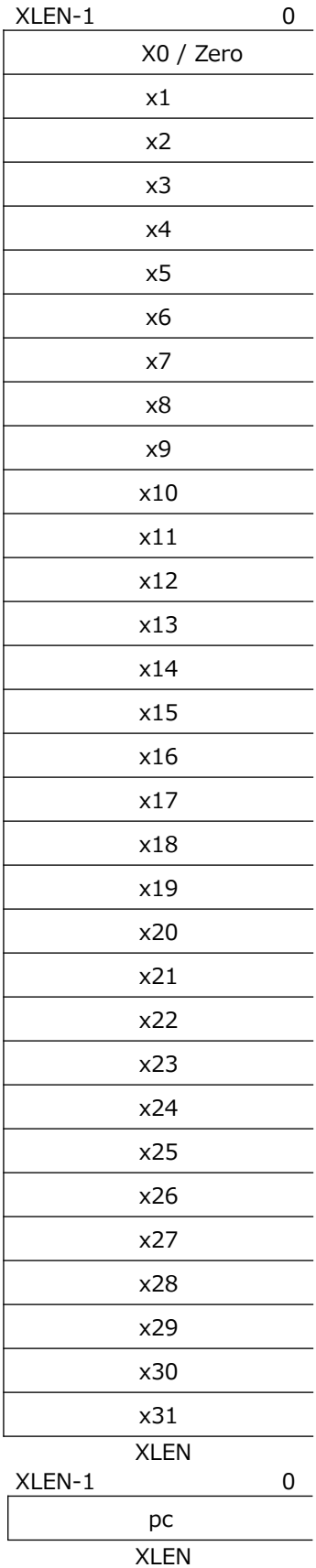


図 2.1：RISC-V ユーザーレベルの基本整数レジスタの状態

-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。

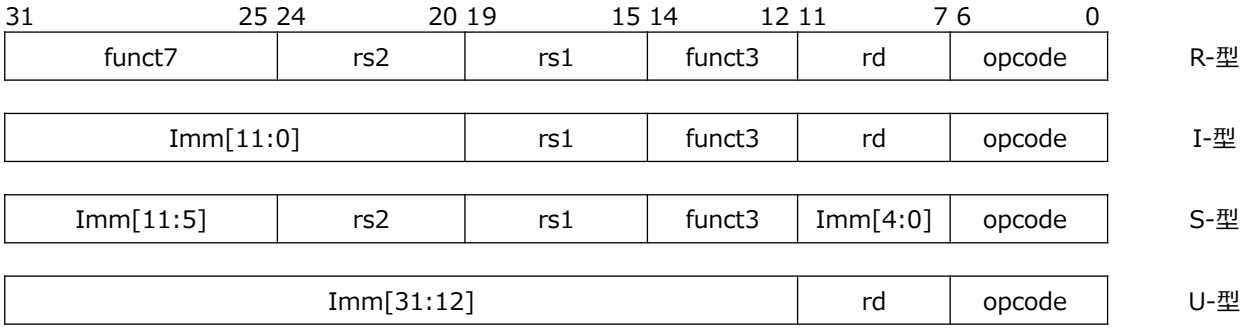


図 2.2 : RISC-V の基本命令フォーマット
各即値サブフィールドは、通常行われる命令の即値フィールド内のビット位置ではなく、生成される即値のビット位置（imm [x]）でラベル付けされます。

-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。

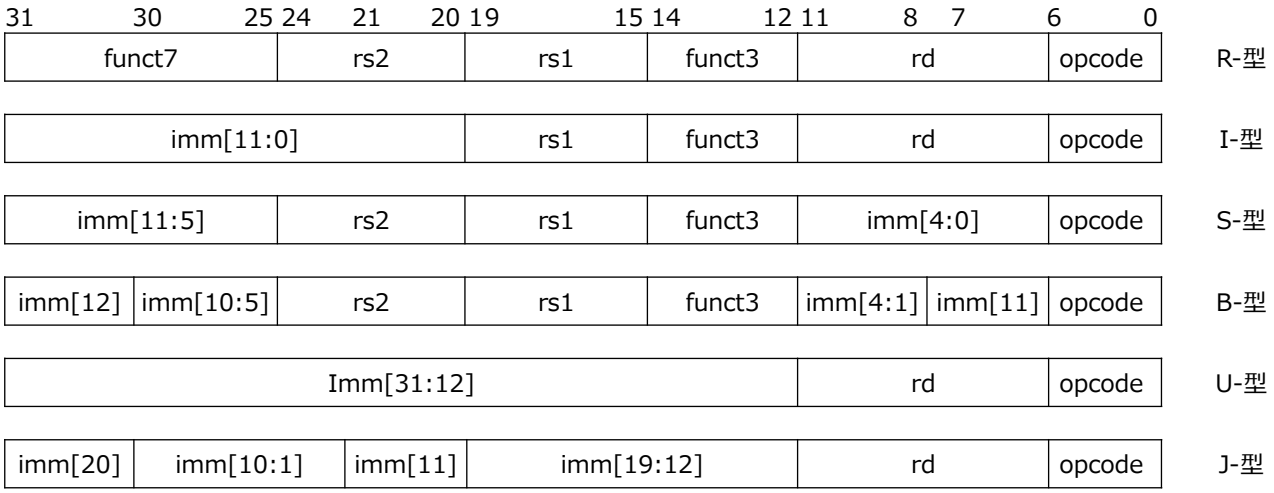


図 2.3: 即値変形を示す RISC V の基本命令形式。
↑variants は種類とかに訳した方がいいかな。即値の種類。しかし J-type とかなんちゅう並びだ。

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

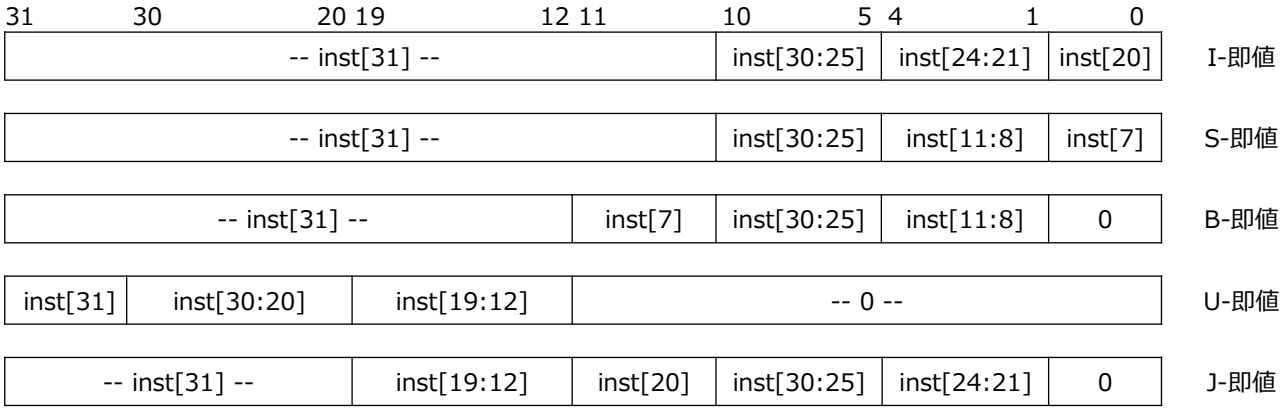


図 2.4：RISC-V 命令で生成される即値の型
フィールドは、値を構成するために使用される命令ビットでラベル付けされています。
符号拡張は常に inst [31]を使用します。

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

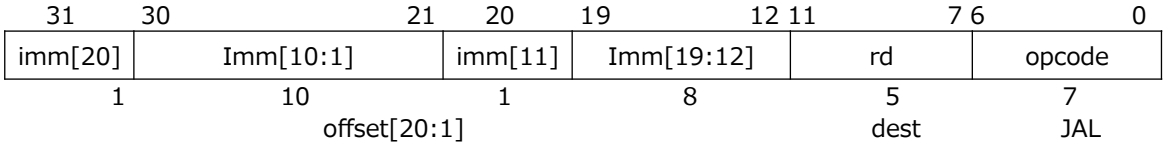
RV32I には、無条件ジャンプと条件分岐の 2 種類の制御転送命令があります。
RV32I の制御転送命令には、アーキテクチャ上見える遅延スロットはありません。

無条件ジャンプ

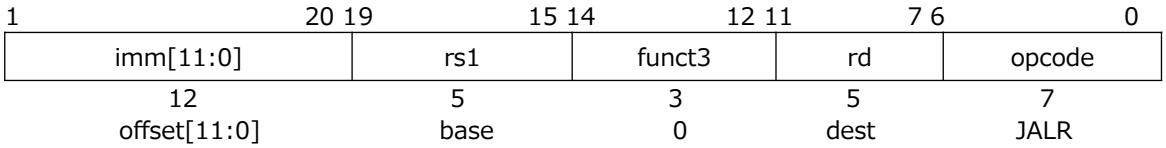
ジャンプおよびリンク（JAL）命令は J 型形式を使用し、J 即時は 2 バイトの倍数で符号付きオフセットを符号化します。
オフセットは符号拡張され、ジャンプターゲットアドレスを形成するために PC に追加されます。
したがって、ジャンプは ±1 MiB 範囲を対象にすることができます。
JAL は、ジャンプ（pc + 4）に続く命令のアドレスをレジスタ rd に格納します。
標準ソフトウェア呼び出し規約では、x1 をリターンアドレスレジスタとして使用し、x5 を代替リンクレジスタとして使用します。

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。

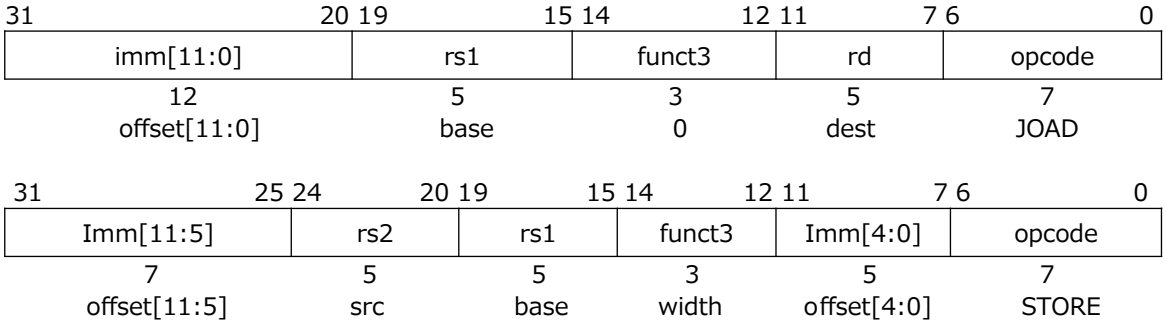
他のいくつかのアーキテクチャとは異なり、RISC-V ジャンプ（rd = x0 の JAL）命令は、常に真の条件の条件付き分岐命令の代わりに常に無条件分岐に対して使用する必要があります。
RISC-V ジャンプは PC 相対でもあり、分岐よりもずっと広いオフセット範囲をサポートし、条件分岐予測テーブルを圧迫することはありません。

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

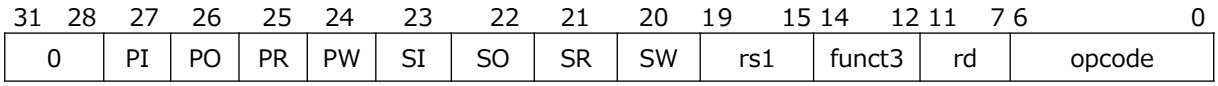
このセクションは、現在のプログラミング言語のメモリモデルを効率的にサポートできるように、RISC-V メモリモデルが現在改訂中であるため、古くなっています。
修正された基本メモリモデルには、少なくとも同じハートからの同じアドレスへのロードを並べ替えることができず、命令間の構文データ依存性が尊重されることを含む、さらなる順序制約が含まれます。
↑ hart ハート ってどこを指しているのか？

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

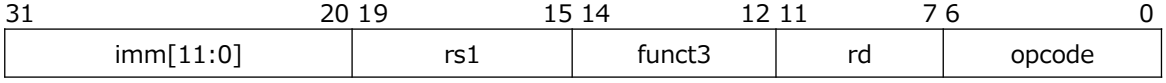


フェンス命令は、他の RISC-V ハートや外部デバイスやコプロセッサで見られるように、デバイス I / O とメモリアクセスを順序付けるために使用されます。
デバイス入力（I）、デバイス出力（O）、メモリ読み出し（R）、およびメモリ書き込み（W）の任意の組み合わせは、それらの任意の組み合わせに関して順序付けられます。
正式には、他の RISC-V ハートまたは外部装置は、フェンスの前に設定された先行のオペレーションの前に、フェンスに続いて後続セットのオペレーションを観察することはできません。
↑非公式に、他の RISC-V ハートまたは外部デバイスは、フェンスの前に設定された前任者の任意の操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。

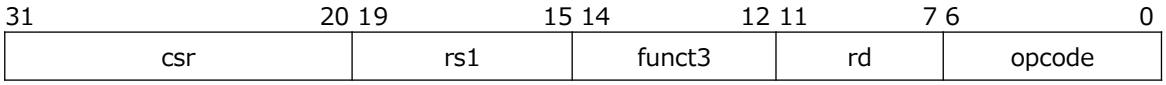
CSRRWI、CSRRSI、および CSRRCI バリアントは、CSRRW、CSRRS、および CSRRC にそれぞれ類似していますが、ただし、整数レジスタの値ではなく rs1 フィールドにエンコードされた 5 ビットの符号なし即値(uimm[4:0]) フィールドをゼロ拡張した XLEN ビット値を使用して CSR を更新する点が異なります。
CSRRSI と CSRRCI の場合、uimm [4 : 0]フィールドがゼロの場合、これらの命令は CSR に書き込まれず、CSR 書き込みで発生する可能性のある副作用を引き起こしません。
CSRRWI の場合、rd = x0 の場合、命令は CSR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

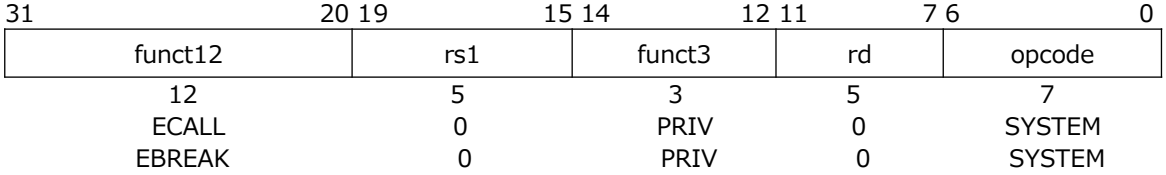
これらの基本カウンタは、基本的なパフォーマンス分析、適応的かつ動的な最適化、およびアプリケーションがリアルタイムストリームで動作するために不可欠であるため、すべての実装で提供することを義務づけています。
パフォーマンスの問題を診断するために追加のカウンタを用意する必要があり、これらのカウンタは、ユーザーレベルのアプリケーションコードから低オーバーヘッドでアクセスできるようにする必要があります。

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