**RISC-V 命令セットマニュアル**

**第一巻：非特権ＩＳＡ**

ドキュメントバージョン 20191213

編集者: アンドリュー・ウォーターマン1, クルステ・アサノヴィク1*,*[[1]](#footnote-1)

1サイファイブ株式会社

カリフォルニア大学バークレー校EECS部門CS部門

andrew@sifive.com krste@berkeley.edu

2019年12月13日

日本語訳　柴田貴康　@shibatchii

202２年１月１日

全てのバージョンの仕様へのコントリビューターは、アルファベット順です。(訂正を提案するために編集者に連絡してください):アービンド、クルステ・アサノヴィック、リマス・アビジエニス、ジェイコブ・バッハマイヤー、クリストファー・F・バッテン、アレン・J・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、チュアンフア・チャン、

デヴィッド・チスノール、ポール・クレイトン、パーマー・ダベルト、ケン・ドクサー、ロジャー・エスパサ、シェイクド・フルール、ステファン

フロイデンベルガー、マーク・ゴーティエ、アンディ・グリュー、ヤン・グレイ、マイケル・ハンブルグ、ジョン・ハウザー、デビッド・ホーナー

ブルース・ハルト、ビル・ハフマン、アレクサンドル・ヨアヌー、オロフ・ヨハンソン、ベン・ケラー、デヴィッド・クラケミーア

ユンスプ・リー、ポール・ローウェンスタイン、ダニエル・ルスティグ、ヤチン・マネルカー、リュック・マランゲ、マーガレット・マートノーシ、ジョセフ・マイヤーズ、ビジャヤナンド・ナガラヤン、リシユール・ニヒル、ヨナス・オーベルハウザー、ステファン・オーリア、アルバート・ウー

ジョン・オースターハウト、デビッド・パターソン、クリストファー・プルテ、ホセ・レナウ、ジョシュ・シャイド、コリン・シュミット、ピーター

スーウェル、スミット・サーカー、マイケル・テイラー、ウェズリー・テルプストラ、マット・トーマス、トミー・ソーン、キャロライン

トリッペル、レイ・ヴァンデウォーカー、ムリリダラン・ビジャヤラガヴァン、ミーガン・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、デレク・ウィリアムズ、アンドリュー・ライト、リヌード・ザンディク、張静雄。

この文書はクリエイティブ・コモンズ 表示 4.0 国際ライセンスの下で公開されています。

この文書は、以下のライセンスの下でリリースされた 「RISC-V命令セットマニュアル、ボリュームI:ユーザーレベルISAバージョン2.1」の派生物です:'2010--2017アンドリュー・ウォーターマン、ユンサップ・リー、デビッド・パターソン、クルステ・アサノビック.クリエイティブ・コモンズ・アトリビューション4.0インターナショナル・ライセンス。

''RISC-V命令セットマニュアル、ボリュームI:ユーザーレベルISA、ドキュメントバージョン20191213"、編集者アンドリュー・ウォーターマンとクルステ・アサノヴイク、RISC-V財団、2019年12月として引用してください。

# 序文

このドキュメントでは、RISC-Vの非特権アーキテクチャについて説明します。

批准とマークされたISAモジュールは、現時点で批准されています。*凍結とマーク*されたモジュールは、批准のために提出される前に大きな変更はないと予想されます。*ドラフトとマーク*されたモジュールは、批准前に変更されることが予想されます。

このドキュメントには、以下のバージョンのRISC-V ISAモジュールが含まれています。

|  |  |  |
| --- | --- | --- |
| ベース | バージョン | ステータス |
| ＲＶＷＭＯ | 2.0 | **批准** |
| **RV32I** | **2.1** | **批准** |
| **ＲＶ６４Ｉ** | **2.1** | **批准** |
| *ＲＶ３２Ｅ* | *1.9* | *ドラフト* |
| *ＲＶ１２８Ｉ* | *1.7* | *ドラフト* |
| 拡張 | バージョン | ステータス |
| **M** | **2.0** | **批准** |
| **A** | **2.1** | **批准** |
| **F** | **2.2** | **批准** |
| **D** | **2.2** | **批准** |
| **Q** | **2.2** | **批准** |
| **C** | **2.0** | **批准** |
| *カウンター* | *2.0* | *ドラフト* |
| *L* | *0.0* | *ドラフト* |
| *B* | *0.0* | *ドラフト* |
| *J* | *0.0* | *ドラフト* |
| *T* | *0.0* | *ドラフト* |
| *P* | *0.2* | *ドラフト* |
| *V* | *0.7* | *ドラフト* |
| **Zicsr** | **2.0** | **批准** |
| **Zifenci** | **2.0** | **批准** |
| *Zam* | *0.1* | *ドラフト* |
| *Ztso* | *0.1* | *凍結* |

このバージョンでの変更点は以下の通りです。

* A拡張、現在はバージョン2.1ですが、2019年12月に理事会で批准されました。
* 定義されたビッグエンディアンISAバリアント。
* ユーザーモード割り込みの拡張Nを第2巻に移動しました。

i

## ドキュメント版 20190608-Base-Ratified 序文

このドキュメントでは、RISC-Vの非特権アーキテクチャについて説明します。

この時点で、RVWMOメモリモデルはRatifiedされています。RatifiedとマークされたISAモジュールは、この時点で批准されています。*Frozen（フローズン）」とマーク*されたモジュールは、批准のために公開される前に大きな変更はないと予想されます。*Draft（ドラフト）」とマーク*されたモジュールは、批准前に変更されると予想されます。

このドキュメントには、以下のバージョンのRISC-V ISAモジュールが含まれています。

|  |  |  |
| --- | --- | --- |
| ベース | バージョン | ステータス |
| ＲＶＷＭＯ | 2.0 | **批准** |
| **RV32I** | **2.1** | **批准** |
| **ＲＶ６４Ｉ** | **2.1** | **批准** |
| *ＲＶ３２Ｅ* | *1.9* | *ドラフト* |
| *ＲＶ１２８Ｉ* | *1.7* | *ドラフト* |
| 拡張 | バージョン | ステータス |
| **Zifencei** | **2.0** | **批准** |
| **Zicsr** | **2.0** | **批准** |
| **M** | **2.0** | **批准** |
| *A* | *2.0* | 凍結 |
| **F** | **2.2** | **批准** |
| **D** | **2.2** | **批准** |
| **Q** | **2.2** | **批准** |
| **C** | **2.0** | **批准** |
| *Ztso* | *0.1* | *凍結* |
| *カウンター* | *2.0* | *ドラフト* |
| *L* | *0.0* | *ドラフト* |
| *B* | *0.0* | *ドラフト* |
| *J* | *0.0* | *ドラフト* |
| *T* | *0.0* | *ドラフト* |
| *P* | *0.2* | *ドラフト* |
| *V* | *0.7* | *ドラフト* |
| *N* | *1.1* | *ドラフト* |
| *Zam* | *0.1* | *ドラフト* |

このバージョンでの変更点は以下の通りです。

* 2019年初頭に理事会で批准されたISAモジュールのために**批准**に記述を移動しました。
* 批准からA拡張を削除しました。
* ISAモジュールのバージョンとの混同を避けるために、ドキュメントのバージョンスキームを変更しました。
* これは、批准されたRVWMOメモリモデルの存在と、以前のベースISAにあったFENCE.I、カウンタ、CSR命令を除外したことを反映したもので、ベース整数ISAのバージョン番号を2.1に増加させました。
* バージョン2.1では正準NaNが変更され、バージョン2.2ではNaN-boxingスキームが定義され、FMIN命令とFMAX命令の定義が変更されたことを反映して、FとD拡張のバージョン番号を2.2に増加させました。
* ISA仕様をプラットフォームプロファイルのマンデートから分離する動きの一環として、「特権を持たない」命令を参照するためにドキュメントの名前を変更しました。
* 実行環境、ハーツ、トラップ、メモリアクセスの定義をより明確かつ正確にしました。
* 定義された命令セットのカテゴリ。 *標準*、*予約済み*、*カスタム*、*非標準*、*不適合*。
* RISC-Vでは代替エンディアンの動作がまだ定義されていないため、代替エンディアンでの動作を意味する文章を削除しました。
* ロードとストアの誤整合の動作の説明を変更しました。この仕様では、ユーザーモードでのロードとストアの不整合の目に見えない処理を義務付けるのではなく、実行環境インタフェースでの不整合アドレスのトラップを目に見える形で許可するようになりました。また、エミュレートされるべきではない不整合アクセス(アトミックを含む)に対して、アクセス例外を報告できるようになりました。
* FENCE.I を必須ベースから外し、Zifencei ISA の名前を持つ別の拡張機能に移動しました。FENCE.IはLinuxのユーザーABIから削除され、大きなインコヒーレントな命令キャッシュとデータキャッシュを持つ実装では問題となる。しかし、唯一の標準的な命令フェッチコヒーレンス機構であることに変わりはありません。
* RV32Eを他の拡張機能で使用する際の禁止事項を削除しました。
* RV32EとRV64Iの章で、特定のエンコーディングが不正な命令例外を生成するというプラットフォーム固有の義務を削除しました。
* カウンタ/タイマ命令は、現在、必須の基本ISAの一部とはみなされていないため、CSR命令は別のチャプターに移動され、バージョン2.0としてマークされ、非特権カウンタは別のチャプターに移動されました。カウンタは、カウンタの不正確さを含む未解決の問題があるため、批准の準備ができていません。
* CSRアクセスオーダーモデルを追加しました。
* 2ビットの*fmtフィールド*で、浮動小数点命令の16ビット半精度浮動小数点フォーマットを明示的に定義しました。
* FMIN.*fmt* と FMAX.*fmt の*符号付きゼロ動作を定義し、提案されているIEEE 754-201x 仕様のminimumNumber と maximumNumber 演算に適合するように、シグナリング-NaN 入力での動作を変更しました。
* メモリ一貫性モデルであるRVWMOが定義されています。
* ずれたAMOを許可し、そのセマンティクスを指定する “Zam"拡張子が定義されました。
* RVWMOよりも厳しいメモリ一貫性モデルを強制する “Ztso"拡張が定義されました。
* 説明文と解説の改善。
* IALIGNという用語を、命令アドレス整列制約を説明するための略語として定義しました。
* P拡張の章のテキストを削除しました。
* V拡張の章のテキストを削除しました。

## ドキュメントバージョン2.2への序文

これは、RISC-V のユーザレベルアーキテクチャを記述した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、以下のバージョンのRISC-V ISAモジュールが含まれています。

|  |  |  |
| --- | --- | --- |
| ベース | *バージョン* | *ドラフトフローズン？* |
| RV32I | 2.0 | Y |
| ＲＶ３２Ｅ | 1.9 | N |
| ＲＶ６４Ｉ | 2.0 | Y |
| ＲＶ１２８Ｉ | 1.7 | N |
| 拡張 | バージョン | 冷凍？ |
| M | 2.0 | Y |
| A | 2.0 | Y |
| F | 2.0 | Y |
| D | 2.0 | Y |
| Q | 2.0 | Y |
| L | 0.0 | N |
| C | 2.0 | Y |
| B | 0.0 | N |
| J | 0.0 | N |
| T | 0.0 | N |
| P | 0.1 | N |
| V | 0.7 | N |
| N | 1.1 | N |

現在までのところ、RISC-Vファウンデーションによって正式に批准された部分はありませんが、上記の ``凍結""と表示されている部分は、仕様の曖昧さや穴を解決する以外に、批准の過程で変更されることはないと予想されます。

このバージョンの主な変更点は以下の通りです。

* この文書の前のバージョンは、原著者によるクリエイティブ・コモンズ表示4.0国際ライセンスの下で公開されており、この文書のこのバージョンと将来のバージョンは同じライセンスの下で公開される予定です。
* すべての拡張子を正典的な順序で優先するように章を再編成しました。
* 説明文と解説の改善。
* LUI/JALRとAUIPC/JALRのペアのより効率的なマクロ操作の融合をサポートするために、JALRの暗黙のヒントを修正しました。
* ロードリザーブ/保存条件付きシーケンスの制約を明確にしました。
* 制御および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の高次ビットの目的と動作を明確にしました。
* FNMADD.*fmt*とFNMSUB.*fmt*命令の説明を修正しました。
* インストラクションFMV.S.XとFMV.X.Sは、それぞれFMV.W.XとFMV.X.Wに改名され、セマンティクスとの整合性を高めましたが、変更はありませんでした。旧名称は引き続きツールでサポートされます。
* NaN-boxingモデルを用いて、より広いfレジスタに保持されるより狭い*(<*FLEN)浮動小数点値の動作を指定しました。
* FMA(∞ , 0, qNaN)の例外動作を定義しました。
* 整数レジスタを使用した固定小数点演算のために、P拡張が整数レジスタを使用した整数パック-SIMD提案に作り直される可能性があることを示すメモを追加しました。
* V ベクトル命令セット拡張のドラフト案。
* N ユーザーレベルのトラップ拡張の初期の草案。
* 擬似命令のリストを拡張したもの。
* RISC-V ELF psABI仕様[1]に取って代わられた呼び出し規約の章の削除。
* Cの拡張機能が凍結され、バージョン2.0にリナンバリングされました。

## ドキュメントバージョン2.1への序文

これは RISC-V のユーザレベルアーキテクチャを記述したドキュメントのバージョン 2.1 です。凍結されたユーザレベルISAベースと拡張IMAFDQバージョン2.0は、この文書の前のバージョン[25]から変更されていませんが、いくつかの仕様の穴が修正され、文書が改善されていることに注意してください。ソフトウェアの規約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善。
* 各章ごとにバージョン番号を分けてください。
* 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、*64* ビット*以上の*長い命令エンコーディングを修正しました。
* CSR命令は、カウンタレジスタが導入される基本整数形式で記述されるようになりました。これは、浮動小数点セクション（および付属の特権アーキテクチャマニュアル）で後から紹介されただけではなく、カウンタレジスタが導入される場所でもあります。
* SCALL 命令と SBREAK 命令の名前がそれぞれ ECALL と EBREAK に変更されました。これらのエンコーディングと機能に変更はありません。
* 浮動小数点型NaNの取り扱いを明確にし、新しい正準NaN値を追加しました。
* 浮動小数点から整数への変換でオーバーフローする値が返されることを明確にしました。
* LR/SCの成功と必要な失敗を明確にし、圧縮された命令の使用を含めたシーケンスの使用を明確にした。
* 整数レジスタ数を減らすための新しいRV32EベースISA提案は、MAC拡張をサポートしています。
* 改訂された呼称規約。。
* ソフトフロート呼び出し規約のための緩和されたスタックアライメントとRV32E呼び出し規約の記述。
* C圧縮拡張のための修正案、バージョン1.9。

## バージョン2.0への序文

これはユーザISA仕様の2回目のリリースであり、ベースとなるユーザISAと一般的な拡張(すなわちIMAFD)の仕様は、今後の開発のために固定されたままにしておきたいと考えています。このISA仕様のバージョン1.0 [24]以降、以下の変更が行われました。

* ISAは整数ベースに分割されており、いくつかの標準的な拡張子を持っています。
* 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配置されています。
* ベースのISAはリトルエンディアンのメモリシステムを持つように定義されており、ビッグエンディアンまたはバイエンディアンは非標準のバリアントとして定義されています。
* 原子命令拡張にLoad-Reserved/Store-Conditional (LR/SC)命令が追加されました。
* AMOとLR/SCはリリース一貫性モデルに対応しています。
* FENCE命令では、メモリの微細化とI/Oのオーダリングが可能です。
* フェッチアンドXOR(AMOXOR)用のAMOが追加され、AMOSWAPのエンコーディングが変更されました。
* PC に 20 ビッ トの上位イミディエイトを追加する AUIPC 命令は、現在の PC 値のみを読み出す RDNPC 命令に取って代わります。これにより、位置に依存しないコードでは大幅な節約になります。
* これにより、JAL命令はU-Type形式の明示的なデスティネーションレジスタを持つ命令に移行し、J命令は削除され、*rd*=x0のJALに置き換わりました。これにより、暗黙の宛先レジスタを持つ唯一の命令が削除され、ベースのISAからJ-Type命令形式が削除されました。これに伴い、JALリーチは減少しますが、ベースISAの複雑さは大幅に減少します。
* JALR命令のスタティック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では、*rd*および*rs1*レジスタ指定子と重複しています。
* 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、ハードウェアを簡素化し、補助情報をファンクションポインタに格納できるようになりました。
* MFTX.SおよびMFTX.D命令は、それぞれFMV.X.SおよびFMV.X.Dに改名されました。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に改名されました。
* MFFSR命令とMTFSR命令がそれぞれFRCSRとFSCSRに改名されました。fcsrの丸めモードと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のFRRM、FSRM、FRFLAGS、FSFLAGS命令が追加されました。
* FMV.X.SとFMV.X.D命令のオペランドのソースをrs*2*からrs*1に*変更しました。
* この変更により、データパスの設計が簡素化されます。
* FCLASS.SとFCLASS.Dの浮動小数点分類命令を追加しました。
* よりシンプルなNaN生成・伝搬方式を採用しました。
* RV32I では、システム性能カウンタが64 ビッ ト幅に拡張され、上位32 ビッ トと下位32 ビッ トへの独立した読み出しアクセスが可能になりました。正準的なNOPおよびMVエンコーディングが定義されました。
* 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 ビット命令用に定義されています。
* 128ビットのアドレス空間バリアントRV128の説明が追加されました。
* 32ビット基本命令フォーマットの主要オペコードは、ユーザー定義のカスタム拡張に割り当てられています。
* ストアがデータを *rd* から取得していることを示唆していた誤植は、*rs2 を*参照するように修正されました。

# 内容

**序文 i**

1. **イントロダクション 1**
   1. RISC-V ハードウェア プラットフォーム用語 . 2
   2. RISC-Vソフトウェア実行環境とハーツ . 3
   3. RISC-V ISAの概要 4
   4. メモリ 6
   5. 基本命令の長さの符号化 。 7
   6. 例外、トラップ、割り込み 10
   7. 未定義の行動と価値観 . 11
2. **RV32I 基本整数命令セット、バージョン2.1 13**
   1. 基本整数ISAのプログラマモデル 13
   2. 基本命令フォーマット . 15
   3. 即時符号化バリアント 16
   4. 整数計算命令 . 17
   5. 制御転送命令 20
   6. ロードとストア命令 . 24
   7. メモリの順序付け命令 . 26
   8. 環境コールとブレークポイント 27
   9. HINTの命令 . 28
3. **Zifenci"説明書-フェッチフェンス、バージョン2.0 31**
4. **RV32E基本整数命令セット、バージョン1.9 33**
   1. RV32E プログラマーズモデル . 33
   2. RV32E命令セット 34
5. **RV64I 基本整数命令セット、バージョン2.1 35**
   1. レジスタ状態 35
   2. 整数計算命令 . 35
   3. ロードとストアの命令 . 37
   4. HINTの命令 . 38
6. **RV128I 基本整数命令セット、バージョン1.7 41**
7. **“M"整数の乗算と除算のための標準拡張、バージョン2.0 43**
   1. 乗算演算 43
   2. 除算演算 . 44
8. **“A"原子命令のための標準拡張、バージョン2.1 47**
   1. 原子命令の順序命令 . 47
   2. ロードリザーブ/保存条件付き命令 . 48
   3. ストア条件付き命令の最終的な成功 . 51
   4. アトミックメモリ演算 52
9. **“Zicsr"、制御および状態レジスタ(CSR)命令、バージョン2.0 55**
   1. CSR命令 . 55
10. **カウンター 59**
    1. ベースカウンタとタイマ 59
    2. ハードウェアパフォーマンスカウンタ . 61
11. **“F"単精度浮動小数点の標準拡張機能、バージョン2.2 63**
    1. F レジスタ状態 。 63
    2. 浮動小数点制御および状態レジスタ . 65
    3. NaNの生成と伝播 66
    4. 非正規化数演算 67
    5. 単一精度のロードとストアの命令 . 67
    6. 単精度浮動小数点演算命令 . 67
    7. 単精度浮動小数点変換と動作命令 . 69
    8. 単精度浮動小数点比較命令 . 71
    9. 単精度浮動小数点分類命令 71
12. **“D"倍精度浮動小数点の標準拡張機能、バージョン 2.2 73**
    1. D レジスタ状態 。 73
    2. 狭義の値のＮａＮボクシング . 73
    3. 倍精度ロードとストア命令 . 74
    4. 倍精度浮動小数点演算命令 . 75
    5. 倍精度浮動小数点変換と移動命令 . 75
    6. 倍精度浮動小数点比較命令 . 77
    7. 倍精度浮動小数点分類命令 77
13. **“Q"4倍精度浮動小数点の標準拡張機能、バージョン2.2 79**
    1. 4 倍精度ロードおよびストア命令 . 79
    2. 4 倍精度計算命令 . 80
    3. 4 倍精度変換および移動命令 . 80
    4. 4 倍精度浮動小数点比較命令 . 81
    5. 4倍精度浮動小数点分類命令 82
14. **RVWMOメモリ一貫性モデル、バージョン0.1 83**
    1. RVWMOメモリモデルの定義 84
    2. CSR依存性追跡の粒度 . 88
    3. ソースおよびデスティネーションレジスタのリスト 88
15. **“L"浮動小数点10進数用の標準拡張機能、バージョン0.0 95**
    1. 10進浮動小数点レジスタ 95
16. **“C"圧縮命令のための標準拡張、バージョン2. 0 97**
    1. 概要 . 97
    2. 圧縮命令フォーマット . 99
    3. ロードとストア命令 . 101
    4. 制御転送命令 104
    5. 整数計算命令 . 106
    6. LR/SCシーケンスにおけるC命令の使用法 . 110
    7. HINT命令 . 110
    8. RVC の命令セットのリスト 。 112
17. **“B"ビット操作のための標準拡張、バージョン0.0 115**
18. **”J"動的翻訳された言語のための標準拡張機能、バージョン0.0 117**
19. **“T"トランザクションメモリの標準拡張、バージョン0.0 119**
20. **“P"パックドSIMD命令のための標準拡張、バージョン0.2 121**
21. **“V"ベクトル演算のための標準拡張、バージョン0.7 123**
22. **“Zam" ミズアラインド・アトミックのための標準拡張、v0.1 125**
23. **“Ztso" トータルストアオーダーのための標準拡張機能, v0.1 127**
24. **RV32/64Gインストラクションセット一覧 129**
25. **RISC-V アセンブリプログラマハンドブック 137**
26. **RISC-V拡張 141**
    1. 拡張用語 . 141
    2. RISC-V 拡張設計思想 . 144
    3. 固定幅32ビット命令フォーマット内での拡張機能 144
    4. 整列された64ビット命令拡張子の追加 . 146
    5. VLIW エンコーディングをサポートしています 。 146
27. **ISA 拡張子の命名規則 149**
    1. 大文字と小文字の区別 149
    2. 基本整数ＩＳＡ 149
    3. 命令セット拡張名 149
    4. バージョン番号 150
    5. アンダースコア . 150
    6. 追加の標準拡張名 150
    7. スーパーバイザーレベルの命令セット拡張 . 151
    8. ハイパーバイザーレベルの命令セット拡張 . 151
    9. マシンレベルの命令セット拡張 . 151
    10. 非標準の拡張子名 151
    11. サブセット命名規則 152
28. **歴史と謝辞 153**
    1. “なぜ新しいISAを開発するのか？" 153
    2. ISAマニュアル改訂1.0からの経緯 155
    3. ISAマニュアル改訂2.0からの経緯 . 156
    4. リビジョン2.1からの経緯 158
    5. リビジョン2.2からの経緯 158
    6. リビジョン2.3からの経緯 159
    7. 資金調達 159
29. **RVWMO説明資料、バージョン0.1 161**
    1. なぜRVWMOなのか？ 161
    2. リトマス試験 . 162
    3. RVWMO規則説明 . 163
       1. プリザーブドプログラムオーダーとグローバルメモリオーダー 164
       2. 荷重値公理 164
       3. 原子性公理 167
       4. 進歩公理 168
       5. 重複するアドレスの順序 (規則 1-3) 168
       6. フェンス（規則4） . 170
       7. 明示的な同期化 (規則5～8) 171
       8. 構文的な依存関係 (規則 9--11) 173
       9. パイプラインの依存関係 (規則 12--13) 176
    4. 主記憶を超えて 177
       1. コヒーレンスとキャッシュ性 178
       2. I/O順序性 178
    5. コードの移植とマッピングのガイドライン . 180
    6. 実施ガイドライン . 184
       1. 将来的に考えられる拡張機能 187
    7. 既知の問題 . 188
       1. ミックスサイズRSW 188
30. **形式メモリモデル仕様書、バージョン0.1 191**
    1. 合金の形式的公理的仕様 . 192
    2. 群集における形式的公理的指定 197
    3. オペレーショナルメモリモデル 201
       1. 命令内擬似コード実行 204
       2. 命令インスタンス状態 206
       3. ハート状態 207
       4. 共有メモリ状態 207
       5. トランジション . 208
       6. 制限事項 216

**第1章**

# 序章

RISC-V (risk-five""と発音します)は新しい命令セットアーキテクチャ(ISA)で、元々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設計されましたが、現在では産業界での実装のための標準的なフリーでオープンなアーキテクチャになることを期待しています。RISC-Vを定義する上での私たちの目標は以下の通りです。

* 学術・産業界で自由に利用できる完全に*オープンな*ISA。
* シミュレーションやバイナリ変換だけではなく、ネイティブハードウェアを直接実装するのに適した*本物の*ISA。
* 特定のマイクロアーキテクチャスタイル(マイクロコード化、インオーダー、デカップリング、アウトオブオーダーなど)やインプリメンテーション技術(フルカスタム、ASIC、FPGAなど)のための「オーバーアーキテクト」を避け、これらのいずれでも効率的なインプリメンテーションを可能にするISA。
* ISAは、カスタマイズされたアクセラレータや教育目的のためのベースとしてそれ自体で使用可能な*小さな*ベース整数ISAと、汎用的なソフトウェア開発をサポートするためのオプションの標準拡張に分かれています。
* 2008年に改訂されたIEEE-754浮動小数点規格に対応しています[7]。
* 広範なISA拡張と特殊なバリアントをサポートするISA。
* アプリケーション、オペレーティングシステムカーネル、ハードウェア実装のための32ビットと64ビットのアドレス空間バリアント。
* 異種マルチプロセッサを含む、高度に並列化されたマルチコアまたはマルチコアの実装をサポートするISA。
* オプションの*可変長命令により*、利用可能な命令エンコーディングスペースを拡張し、パフォーマンス、スタティックコードサイズ、エネルギー効率を向上させるための密な*命令エンコーディングを*サポートします。
* ハイパーバイザー開発を容易にするための完全仮想化可能なISA。
* 新しい特権アーキテクチャ設計による実験を簡素化するISA。

*我々の設計決定に関する解説は、この段落のようにフォーマットされています。読者が仕様そのものにしか興味がない場合には、この非標準的なテキストは省略することができます。*

*RISC-Vという名前は、UC Berkeleyの5番目のメジャーなRISC ISA設計を表すために選ばれました。*

*(RISC-I [15]、RISC-II [8]、SOAR [21]、SPUR [11]が最初の4つでした)。また、ローマ数字の ``V"" が ``variations"" と ``vector"" を意味することを、以下のように表現しています。*

また、さまざまなデータ並列アクセラレータを含むさまざまなアーキテクチャ研究のサポートがISA設計の明確な目標であるため、「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ます*。*

RISC-V ISAは、可能な限り実装の詳細を避けて定義されています(ただし、実装主導の決定については解説が含まれています)ので、特定のハードウェアの設計というよりは、多様な実装へのソフトウェア的に見えるインターフェースとして読むべきです。RISC-Vマニュアルは2巻構成となっています。このボリュームでは、オプションの特権なしISA拡張を含む基本的な特権*なし*命令の設計をカバーしています。非特権命令とは、特権モードや特権アーキテクチャによって動作が異なる場合がありますが、一般的にはすべての特権モードですべての特権アーキテクチャで使用可能な命令のことを指します。第2巻では、最初の(古典的な)特権アーキテクチャの設計を提供しています。マニュアルは IEC 80000-13:2008 の規約を使用しており、1 バイトは 8 ビットです。

*特権のないISA設計では、キャッシュラインサイズなどの特定のマイクロアーキテクチャー機能や、ページ変換などの特権的なアーキテクチャーの詳細への依存を排除しようとしました。これは、単純化のためであると同時に、代替のマイクロアーキテクチャまたは代替の特権アーキテクチャに対する最大限の柔軟性を可能にするためでもあります。*

## 1.1　RISC-Vハードウェアプラットフォーム用語

RISC-Vハードウェアプラットフォームは、1つ以上のRISC-V互換の処理コアと他の非RISC-V互換のコア、固定機能アクセラレータ、様々な物理メモリ構造、I/Oデバイス、およびコンポーネントが通信できるようにするための相互接続構造を含むことができます。

コンポーネントは、独立した命令フェッチユニットを含む場合、*コアと呼ば*れます。RISC-V互換コアは、マルチスレッドによって複数のRISC-V互換ハードウェアスレッド、つまり*ハートを*サポートしている場合があります。

RISC-Vコアには、特殊な命令セット拡張機能や*コプロセッサが追加されている*場合があります。*コプロセッサと*は、RISC-Vコアに接続され、ほとんどがRISC-V命令ストリームによってシーケンスされるユニットを指しますが、追加のアーキテクチャ状態と命令セット拡張を含み、場合によってはプライマリRISC-V命令ストリームに対して限定的な自律性を持つユニットを指します。

*アクセラレータという*用語は、プログラマブルではない固定機能ユニット、または自律的に動作することができるが特定のタスクに特化したコアのいずれかを指しています。RISC-Vシステムでは、プログラマブル・アクセラレータの多くは、特殊な命令セット拡張やカスタマイズされたコプロセッサを備えたRISC-Vベースのコアになると予想されます。RISC-Vアクセラレータの重要なクラスにI/Oアクセラレータがあり、I/O処理タスクをメインのアプリケーションコアからオフロードします。

RISC-Vハードウェアプラットフォームのシステムレベルの構成は、シングルコアのマイクロコントローラから、共有メモリのマルチコアサーバノードからなる数千ノードのクラスタまで多岐にわたります。小規模なシステムオンチップであっても、マルチコンピュータやマルチプロセッサの階層構造にすることで、開発作業をモジュール化したり、サブシステム間の安全な分離を実現したりすることができます。

## 1.2　RISC-Vのソフトウェア実行環境とハーツ

RISC-V プログラムの動作は、実行環境に依存します。RISC-V実行環境インタフェース(EEI)は、プログラムの初期状態、ハートがサポートする特権モードを含む環境内のハートの数とタイプ、メモリとI/O領域のアクセス性と属性、各ハート上で実行されるすべての命令の動作(すなわち、ISAはEEIの構成要素の1つです)、環境コールを含む実行中に発生する割り込みや例外の処理を定義します。EEIの例としては、Linuxアプリケーションバイナリインタフェース(ABI)やRISC-Vスーパーバイザバイナリインタフェース(SBI)などがあります。RISC-V実行環境の実装は、純粋なハードウェア、純粋なソフトウェア、またはハードウェアとソフトウェアの組み合わせが可能です。例えば、オペコードトラップやソフトウェアエミュレーションを使用して、ハードウェアにはない機能を実装することができます。実行環境の実装例としては、以下のようなものがあります。

* ベアメタル"ハードウェアプラットフォームは、ハードウェアが物理プロセッサのスレッドによって直接実装され、命令が物理アドレス空間に完全にアクセスできるようになっています。ハードウェアプラットフォームは、電源投入時のリセットから始まる実行環境を定義します。
* ユーザレベルのハートを利用可能な物理プロセッサのスレッドに多重化し、仮想メモリを介してメモリへのアクセスを制御することで、複数のユーザレベルの実行環境を提供するRISC-Vオペレーティングシステム。
* ゲストOSに複数のスーパーバイザレベルの実行環境を提供するRISC-Vハイパーバイザー。
* Spike, QEMU, rv8 などの RISC-V エミュレータは、基盤となる x86 システム上で RISC-V ハーツをエミュレートし、ユーザレベルまたはスーパーバイザレベルの実行環境を提供することができます。

*ベア・ハードウェア・プラットフォームは、アクセス可能なハードウェア、メモリ、および他のデバイスが環境を構成し、初期状態はパワーオンリセット時のものである、EEIを定義するために考慮することができます。一般的に、ほとんどのソフトウェアは、ハードウェアへのより抽象的なインターフェースを使用するように設計されており、より抽象的なEIは、異なるハードウェアプラットフォーム間でより大きな移植性を提供するからです。多くの場合、EEIはお互いの上に重ねられており、1つの高レベルのEEIが別の低レベルのEEIを使用しています。*

ある実行環境で動作するソフトウェアから見れば、ハートはその実行環境内で自律的にRISC-V命令をフェッチして実行する資源です。この点では、実行環境によって実ハードウェア上に時分割多重化されていても、ハートはハードウェアスレッドリソースのように振る舞います。いくつかの EEIs は、例えば、新しいハートをフォークするための環境コールを介して、追加のハートの生成と破壊をサポートしています。

実行環境は、各ハートの最終的な前進を確実にする責任があります。あるハートに対しては，本仕様の第2巻で定義されている割り込み待ち命令のようなイベントを明示的に待つメカニズムを実行している間は，その責任は中断され，ハートが終了した場合にはその責任は終了します．以下のイベントが前進を構成すします。

* 命令の廃止。
* セクション1.6で定義されているトラップ。
* 前進を構成するための延長線上に定義されたその他のイベント。

*hartという用語は、ソフトウェア・スレッド・プログラミングの抽象化とは対照的に、抽象的な実行リソースを表す用語を提供するために、Lithe [13, 14]の研究で導入されました。*

*ハードウェアスレッド(ハート)とソフトウェアスレッドコンテキストの重要な違いは、実行環境内で実行されているソフトウェアは、それぞれのハートの進行を引き起こす責任がなく、それは外部の実行環境の責任であるということです。そのため、実行環境内のソフトウェアから見れば、環境のハートはハードウェアスレッドのように動作します。*

*実行環境の実装は、ゲストハートのセットを、それ自身の実行環境が提供するより少ないホストハートに時間多重化するかもしれませんが、ゲストハートが独立したハードウェアスレッドのように動作するようにしなければなりません。特に、ゲストハートがホストハートよりも多い場合、実行環境はゲストハートを先取りすることができなければならず、ゲストハート上のゲストソフトウェアがゲストハートの制御を""降伏""するのを無期限に待ってはいけません。*

## 1.3　RISC-V ISAの概要

RISC-V ISAはベース整数ISAとして定義され、どのようなインプリメンテーションでも必ず存在しなければならないもので、ベースISAにオプションの拡張を加えたものです。ベース整数ISAは、分岐遅延スロットを持たず、オプションで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けば、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、オペレーティングシステム(特権操作を追加したもの)に適切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慎重に制限されており、よりカスタマイズされたプロセッサのISAを構築するための便利なISAとソフトウェアツールチェーン「スケルトン」を提供します。

RISC-V ISA*といえ*ば便利ですが、RISC-Vは実際には関連するISAのファミリーであり、その中には現在4つのベースISAがあります。各ベース整数命令セットは、整数レジスタの幅とそれに対応するアドレス空間のサイズ、整数レジスタの数によって特徴づけられています。第2章と第5章で説明したRV32IとRV64Iの2つの主要な基底整数バリアントがあり、それぞれ32ビットまたは64ビットのアドレス空間を提供します。私たちは以下のような用語を使用します。

XLEN は、整数レジスタの幅をビット単位(32 または 64)で参照するために使用します。第4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基本命令セットのRV32Eサブセットについて説明します。第6章では、フラット128ビットのアドレス空間(XLEN=128)をサポートするベース整数命令セットの将来のRV128Iバリエーションをスケッチしています。基本整数命令セットは、符号付き整数値に 2 の補数表現を使用します。

*64ビットのアドレス空間は大規模なシステムの要件ですが、32ビットのアドレス空間は、今後数十年の間、多くの組み込み機器やクライアント機器にとって十分なものであり、メモリ・トラフィックとエネルギー消費を低減するために望ましいものであると考えています。さらに、教育目的には32ビットのアドレス空間で十分です。最終的には、より大きなフラット128ビットのアドレス空間が必要になるかもしれないので、RISC-V ISAフレームワークの中でこれを収容できるようにしました。*

*RISC-Vの4つのベースISAは、別個のベースISAとして扱われています。よくある質問は、なぜ単一のISAが存在しないのか、特にRV32IはRV64Iの厳密なサブセットではないのかということです。いくつかの初期のISAデザイン(SPARC、MIPS)では、新しい64ビットハードウェア上で既存の32ビットバイナリの実行をサポートするために、アドレス空間サイズを大きくする際に、厳密なスーパーセットポリシーを採用していました。*

*ベース ISA を明示的に分離する主な利点は、他のベース ISA に必要なすべての操作をサポートする必要がなく、各ベース ISA を必要に応じて最適化できることです。例えば、RV64Iは、RV32Iの狭いレジスタに対処するためにのみ必要な命令やCSRを省略することができます。RV32Iバリアントでは、より広いアドレス空間のバリアントでのみ必要とされる命令のために予約されたエンコーディング空間を使用することができます。*

*デザインを1つのISAとして扱わない主な欠点は、1つのベースISAを別のベースISAでエミュレートするために必要なハードウェアが複雑になることです(例:RV32IとRV64Iの上のRV32I)。しかし、アドレッシングや不正な命令トラップの違いは、一般的に、スーパーセット命令の完全なエンコーディングを使用しても、どのような場合でもハードウェアで何らかのモード切り替えが必要になることを意味し、異なるRISC-VベースのISAは類似しているため、複数のバージョンをサポートすることは比較的低コストです。厳密なスーパーセット設計により、レガシー32ビットライブラリを64ビットコードでリンクできるようにするという提案もありますが、ソフトウェアの呼び出し規約やシステムコールインターフェースの違いにより、互換性のあるエンコーディングであっても実際には非現実的です。*

*RISC-V 特権アーキテクチャは、同じハードウェア上で異なるベースのISAをエミュレートすることをサポートするために、各レベルで非特権ISAを制御するためのフィールドを*misa*に提供しています。新しいSPARCとMIPS ISAリビジョンでは、64ビットシステム上で32ビットコードを変更せずに実行するサポートが廃止されていることに注意してください。*

*関連する質問としては、RV32I (ADD) の 32 ビット加算と*

*RV64I (ADDW)？ADDWオペコードは、RV32Iでは32ビット加算に、ADDDでは*

*での64 ビッ ト加算には、RV32I での32 ビッ ト加算には同じオペコードADD を使用し、RV64I での32 ビッ ト加算には異なるオペコードADDW を使用する既存のデザインではなく、RV64I での64 ビッ ト加算には同じオペコードADD を使用するようにします。の両方で32 ビ ッ ト 負荷に同じLW オ プ コ ー ド を使用す る 方が整合性があ り ます。*

*RV32IとRV64Iです。RISC-V ISAの最初のバージョンには、この代替デザインのバリエーションがありましたが、2011年1月に現在のRISC-Vデザインに変更されました。私たちの焦点は、32ビットISAとの互換性を提供することではなく、64ビットISAで32ビット整数をサポートすることでした。*

*RV32Iには\*Wという接尾辞が付いています(例えばADDWですが、ANDWではなくAND)。また、プラットフォームの要件をISA仕様に折り込まなければならないと考えていたため、RV32Iの命令はすべてRV64Iで必要とされていたことになります。今さらエンコーディングを変更するのは遅すぎますが、上記の理由から実用的な意味はほとんどありません。*

*RV32I システムの拡張として \*W バリアントを有効にして、RV64I と将来の RV32 バリアントに共通のエンコーディングを提供できるようにすることができると指摘されています。*

RISC-Vは広範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。各ベース整数ISAは、1つ以上のオプションの命令セット拡張で拡張することができ、RISC-Vの各命令セットエンコーディング空間(および関連するCSRなどのエンコーディング空間)を、*標準*、*予約済み*、*カスタムの3*つの独立したカテゴリに分けています。標準エンコーディングはファウンデーションによって定義されており、同じベースISAの他の標準拡張と競合してはならない。予約エンコーディングは現在定義されていないが、将来の標準拡張のために保存されている。*非標準という用語は、*ファウンデーションによって定義されていない拡張を説明するために使用します。カスタムエンコーディングは、標準拡張には決して使用してはならず、ベンダ固有の非標準拡張で使用できるようになっています。標準または予約されたエンコーディングのいずれかを使用する非標準の拡張を説明するために、非*適合という*用語を使用します(すなわち、カスタム拡張は非適合ではあり*ません*)。命令セット拡張は一般的に共有されていますが、ベースのISAに応じてわずかに異なる機能を提供する場合があります。第26章では、RISC-V ISAを拡張する様々な方法を説明します。また、第27章で詳しく説明しているRISC-Vベース命令と命令セット拡張の命名規則を開発しました。

より一般的なソフトウェア開発をサポートするために、整数の乗除算、原子演算、単精度および倍精度の浮動小数点演算を提供するための一連の標準拡張機能が定義されています。

ベースとなる整数の ISA は ``I"" と名付けられています。(整数レジスタの幅によってはRV32またはRV64がプレフィックス)で、整数演算命令、整数ロード、整数ストア、コントロールフロー命令が含まれています。標準的な整数の乗除算命令の拡張子は ``M""と呼ばれ、整数レジスタに保持されている値を乗算したり割ったりする命令を追加します。標準的なアトミック命令拡張は ``A""と呼ばれ、プロセッサ間同期のためにメモリの読み込み、変更、書き込みをアトミックに行う命令を追加します。標準的な単精度浮動小数点命令拡張モジュール(``F"")は浮動小数点レジスタ、単精度の計算命令、単精度のロードとストアを追加します。標準的な倍精度浮動小数点拡張子 ``D""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、ストアを追加します。標準の圧縮命令拡張 ``C""は、一般的な命令をより狭い16ビット形式で提供します。

基本の整数ISA と標準のGC 拡張を超えて、新しい命令がすべてのアプ リ ケーシ ョ ンに大きな利益をもたらすことは稀であると考えています。エネルギー効率の懸念がより大きな特殊化を強制しているため、ISA 仕様の必要な部分を単純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を単一の実体として扱い、時間の経過とともに命令が追加されると新しいバージョンに変化しますが、RISC-Vでは、基本と各標準拡張を時間の経過とともに一定に保ち、その代わりに新しい命令をさらにオプションの拡張として重ねていきます。例えば、ベースとなる整数ISAは、その後の拡張に関わらず、完全にサポートされたスタンドアロンISAとして継続されます。

## 1.4　メモリ

RISC-Vハートは、すべてのメモリアクセスに対して2*XLENバイト*のシングルバイトアドレス空間を持っています。メモリの*ワード*は32ビット(4バイト)と定義されます。これに対応して、*ハーフワードは*16ビット(2バイト)、*ダブルワードは*64ビット(8バイト)、*クアッドワードは*128ビット(16バイト)となります。メモリアドレス空間は円形であり、アドレス2*XLEN* - 1のバイトがアドレス0のバイトに隣接するようになっています。そのため、 ハー ド ウ ェ アに よ っ て実行 さ れ る メ モ リ ア ド レ ス計算では、 オーバーフ ロ ーは無視 さ れず、 代わりにモジュロ2*XLEN* にラップ さ れます。

実行環境は、ハードウェアリソースのハートのアドレス空間へのマッピングを決定します。ハートのアドレス空間の異なるアドレス範囲には、(1) 空き領域、(2) *メインメモリ*、(3) 1つ以上の*I/Oデバイスが*含まれています。I/Oデバイスの読み書きは目に見える副作用があるかもしれませんが、メインメモリへのアクセスはできません。実行環境がハートのアドレス空間内のすべてをI/Oデバイスと呼ぶことは可能ですが、通常はその一部をメインメモリとして指定することが予想されます。

RISC-Vプラットフォームが複数のハーツを持つ場合、2つのハーツのアドレス空間は完全に同じであったり、完全に異なっていたり、あるいは部分的に異なっていてもリソースのサブセットを共有していて、同じアドレス範囲にマッピングされていたり、異なっていたりすることがあります。

*純粋に「ベアメタル」環境では、全てのハーツは物理アドレスのみでアクセスされる同一のアドレス空間を見ることができます。しかし、実行環境にアドレス変換を採用したオペレーティングシステムが含まれている場合、各ハ ートには、大部分または完全に独自の仮想アドレス空間が与えられるのが一般的です。*

RISC-V マシンの各命令の実行には 1 つ以上のメモリアクセスが必要で、暗黙*的アクセス*と*明示的*アクセスに分けられます。実行される各命令に対して、暗黙*の*メモリ読み出し（命令フェッチ）が行われ、実行するエンコードされた命令が得られます。多くのRISC-V命令は、命令フェッチを超えたメモリアクセスを行いません。特定のロード/ストア命令は、命令によって決定されたアドレスでメモリの*明示的な*読み出し/書き込みを行います。実行環境によっては、命令実行時に、非特権ISAで文書化されている以上の暗*黙の*メモリアクセス(アドレス変換の実装など)を行う場合があります。

実行環境は、各種類のメモリアクセスに対して、非空白アドレス空間のどの部分がアクセス可能かを決定します。例えば、命令フェッチのために暗黙的に読み込める場所のセットは、ロード命令によって明示的に読み込める場所のセットと重複する場合と重複しない場合があり、ストア命令によって明示的に書き込める場所のセットは、読み込める場所のサブセットに過ぎない場合があります。通常、アクセスできないアドレスでメモリにアクセスしようとすると、例外が発生します。アドレス空間の空の場所は決してアクセスできません。

特に指定がない場合を除き、例外を発生させず、副作用のない暗黙のリードは、マシンがリードが必要であることを証明する前であっても、任意の早い時期に推測的に発生する可能性があります。例えば、有効な実装では、最も早い機会にメインメモリの全てを読み込み、後の命令フェッチのために可能な限り多くのフェッチ可能な（実行可能な）バイトをキャッシュし、命令フェッチのためにメインメモリの読み込みを二度と行わないようにすることができます。特定の暗黙の読み取りが同じメモリ位置への書き込みの後にのみ順序付けられることを確実にするために、ソフトウェアはこの目的のために定義された特定のフェンスまたはキャッシュ制御命令を実行しなければなりません（第3章で定義されたFENCE.I命令のような）。

あるハートによって行われたメモリアクセス（暗黙的または明示的）は、別のハートや、同じメモリにアクセスできる他のエージェントによって知覚されるように、異なる順序で発生しているように見えるかもしれません。しかし、このように認識されるメモリアクセスの再順序付けは、適用されるメモリ一貫性モデルによって常に制約されています。RISC-Vのデフォルトのメモリ一貫性モデルは、第14章と付録で定義されているRISC-V Weak Memory Ordering (RVWMO)である。オプションとして、23章で定義されているより強力なTotal Store Orderingモデルを採用してもよい。実行環境はまた、メモリアクセスの知覚された再順序付けをさらに制限する制約を追加してもよい。RVWMOモデルはRISC-V実装で最も弱いモデルであるため、このモデル用に書かれたソフトウェアは、すべてのRISC-V実装の実際のメモリ整合性規則と互換性があります。暗黙の読み出しと同様に、ソフトウェアは、想定されるメモリ一貫性モデルと実行環境の要件を超えたメモリアクセスの特定の順序を確保するために、フェンスまたはキャッシュ制御命令を実行しなければなりません。

## 1.5　ベース命令長符号化

ベースのRISC-V ISAは固定長の32ビット命令を持ち、32ビット境界で自然に整列されなければなりません。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おり、各命令は任意の数の16ビット命令パー*セルの*長さにすることができ、パーセルは16ビット境界上で自然に整列されます。第16章で説明した標準の圧縮ISA拡張は、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、アライメント制約を緩和して、すべての命令(16ビットおよび32ビット)を任意の16ビット境界上でアラインメントできるようにし、コード密度を向上させます。

我々は、実装が強制する命令アドレス整列制約を参照するために、IALIGN（ビット単位で測定される）という用語を使用します。IALIGNは基本ISAでは32ビットですが、圧縮ISA拡張を含むいくつかのISA拡張は、IALIGNを16ビットに緩和します。IALIGNは16または32以外の値を取ることはできません。

ILEN（ビット単位で測定）という用語を使用して、実装がサポートする最大命令長を参照するために使用します。基本命令セットのみをサポートする実装では、ILENは32ビットです。より長い命令をサポートする実装では、ILENの値が大きくなります。

図1.1に標準的なRISC-Vの命令長エンコーディング規約を示します。ベースのISAのすべての32ビット命令の最下位2ビットは11に設定されています。オプションの圧縮16ビット命令セット拡張は、最下位2ビットが00、01、または10になります。

### 拡張命令長符号化

32ビット命令の符号化スペースの一部は、32ビットより長い命令のために暫定的に割り当てられています。現時点では、このスペースの全体が予約されており、32ビットより長い命令の符号化については、以下の提案はフリーズしていないと考えられます。

32 ビッ ト以上でエンコードされた標準の命令セット拡張は、48 ビッ ト長と 64 ビッ ト長の規則を図 1.1 に示すように、追加の低次ビットを 1 に設定します。80 ビッ トから 176 ビッ トまでの命令長は、最初の 5 つの 16 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エンコードされる。ビット[14:12]が111に設定されたエンコーディングは、将来のより長い命令エンコーディングのために予約されています。

|  |
| --- |
| xxxxxxxxxxxxxxaa |

16ビット (aaa not≠ 11)

|  |  |
| --- | --- |
| xxxxxxxxxxxxxxxxxxxx | xxxxxxxxxxxbbb11 |

32ビット(bbb ≠111)

|  |  |  |
| --- | --- | --- |
| …xxxx | xxxxxxxxxxxxxxxxxxxx | xxxxxxxxxx011111 |
|  |  |  |
| …xxxx | xxxxxxxxxxxxxxxxxxxx | Xxxxxxxxx0111111 |
|  |  |  |
| …xxxx | xxxxxxxxxxxxxxxxxxxx | xnnnxxxxx1111111 |
|  |  |  |
| …xxxx | xxxxxxxxxxxxxxxxxxxx | x111xxxxx1111111 |

48ビット

64ビット

(80+16\*nnn)bit,nnn≠111

Reserved for ≥192-bit

バイトアドレス ベース+4 ベース+2 ベース

図 1.1.1.RISC-Vの命令長エンコーディング。現時点では、16ビットと32ビットのエンコーディングのみがフリーズしていると考えられています。

*圧縮フォーマットのコードサイズとエネルギーの節約を考えると、私たちは圧縮フォーマットのサポートをISAエンコーディングスキームに組み込みたいと考えていましたが、よりシンプルな実装を可能にするために、圧縮フォーマットを必須にしたくありませんでした。また、実験や大規模な命令セットの拡張をサポートするために、オプションでより長い命令を許可したいと考えていました。*

*私たちのエンコーディング規約では、コアとなるRISC-V ISAのエンコーディングをより厳密にする必要がありましたが、これはいくつかの有益な効果があります。*

*。標準的な IMAFD ISA の実装は、命令キャッシュに最も重要な 30 ビットだけを保持する必要があります(6.25%の節約)。命令キャッシュのリフィルでは、ロービットクリアと遭遇した命令は、キャッシュに格納する前に、不正な命令例外動作を維持するために、不正な30ビット命令に再コード化されなければなりません。*

*さらに重要なのは、ベースのISAを32ビット命令ワードのサブセットに凝縮することで、標準外の拡張やカスタム拡張に使用できるスペースを増やすことができたことかもしれません。特に、 ベース のRV32I ISA は32 ビッ ト命令ワードのエンコーディング スペースの1/8 以下しか使用していません。第26章で説明したように、標準の圧縮命令拡張をサポートする必要のない*インプリメンテーション*では、標準の32ビット命令セット拡張のサポートを維持したまま、不適合な30ビット命令空間を3つ追加して32ビット固定幅フォーマットにマッピングすることができます。さらに、実装が32ビットを超える長さの命令を必要としない場合、不適合拡張のためにさらに4つの主要なオペコードを回復することができます。*

ビット[15:0]のすべてのゼロを持つエンコードは、不正な命令として定義されます。これらの命令は最小の長さとみなされ、16ビットの命令セット拡張があれば16ビット、そうでなければ32ビットとなります。ビ ッ ト [ILEN-1:0] のすべての 1 を用いたエン コ ーデ ィ ン グ も 違法であ り 、 こ れ ら の命令は ILEN ビ ッ ト の長 さ と 見な さ れます。

*すべてのゼロビットを含む命令の長さは、ゼロ化された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、合法ではないという特徴を考慮しています。同様に、我々はまた、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で観察される他の一般的なパターンをキャッチするために、すべての1を含む命令エンコーディングが不正な命令であることを留保します。*

*またはメモリデバイスが壊れています。*

*ソフトウェアは、すべてのRISC-V実装において、ゼロを含む自然整列された32ビットワードを不正命令として動作させることができ、不正命令が明示的に必要とされるソフトウェアで使用することができます。すべてのワードに対応する既知の不正な値を定義することは、可変長エンコーディングのため、より困難です。ソフトウェアが最終的なターゲットマシンのILENを知らない可能性があるため、ソフトウェアは一般的にすべての1のILENビットの不正な値を使用することはできません(例えば、ソフトウェアが多くの異なるマシンで使用される標準的なバイナリライブラリにコンパイルされている場合)。すべてのマシンが32ビット命令サイズをサポートしなければならないため、すべての1の32ビットワードを不正なものとして定義することも検討されましたが、これは、ILEN>32のマシン上の命令フェッチユニットが、そのような命令が保護境界を境にしたときにアクセスフォルトではなく不正な命令例外を報告する必要があり、可変命令長のフェッチとデコードを複雑にします。*

RISC-VベースのISAにはリトルエンディアンまたはビッグエンディアンのメモリシステムがあり、特権アーキテクチャではさらにバイエンディアン動作が定義されています。命令は、メモリシステムのエンディアンに関係なく、16ビットのリトルエンディアンパーセルのシーケンスとしてメモリに格納されます。1 つの命令を構成するパーセルは、増加するハーフワードアドレスに格納され、最もアドレスの低いパーセルは命令仕様の中で最も低い番号のビットを保持します。

*当初、RISC-Vメモリシステムにリトルエンディアンのバイト順を選択したのは、リトルエンディアンシステムが現在商業的に主流であるからです（すべてのx86システム、iOS、Android、ARMのWindows）。マイナーな点としては、ハードウェア設計者にとってリトルエンディアンメモリシステムの方が自然であることも分かっています。しかし、IPネットワーキングのような特定のアプリケーション分野では、ビッグエンディアンのデータ構造で動作し、また、レガシーコードベースの一部はビッグエンディアンプロセッサを想定して構築されているため、RISC-Vのビッグエンディアンとバイエンディアンの亜種を定義しています。*

*メモリシステムのエンディアンとは無関係に、命令小包がメモリに格納される順序を固定し、長さを符号化するビットが常にハーフワードアドレスの順序で最初に現れるようにしなければなりません。これにより、可変長命令の長さは、最初の１６ビット命令区画の最初の数ビットのみを検査することにより、命令フェッチユニットによって迅速に決定される。*

*さらに、命令パーセル自体をリトルエンディアンにして、命令エンコーディングをメモリシステムのエンディアンから完全に切り離すようにしています。この設計は、ソフトウェアツールとバイエンディアンハードウェアの両方にメリットがあります。そうでなければ、例えばRISC-Vのアセンブラやディスアセンブラは、バイエンディアンシステムでは実行中にエンディアンモードが動的に変化する可能性があるにもかかわらず、常に意図したアクティブなエンディアンを知る必要があります。対照的に、命令に固定エンディアンを与えることで、慎重に書かれたソフトウェアがエンディアンに依存しないようにすることが可能な場合もあります。*

*バイナリ形式であっても、位置に依存しないコードのように*

*命令をリトルエンディアンのみにするという選択は、しかしながら、次のような結果をもたらします。*

*機械命令をエンコードまたはデコードするRISC-Vソフトウェア。ビッグエンディアンのJITコンパイラなどでは、命令メモリに格納する際にバイトオーダーを入れ替えなければなりません。*

*リトルエンディアン命令のエンコーディングに固定することを決めたら、当然のことながら、オペコードフィールドを分割しないように命令フォーマットのLSB位置にレングスエンコーディングビットを配置することになりました。*

## 1.6 例外、トラップ、割り込み

*例外と*は、現在の RISC-V hart の命令に関連して実行時に発生する異常な状態を指します。*割り込みと*は、RISC-Vハーツの制御が予期せぬ転送される原因となる外部の非同期イベントを指します。*トラップ*とは、例外または割り込みによって発生したトラップハンドラへの制御の転送を指します。

以下の章では、実行中に例外が発生する可能性のある条件を説明します。ほとんどの RISC-V EEI の一般的な動作は、命令上で例外がシグナルされたときにハンドラへのトラップが発生するというものです(浮動小数点例外を除いて、標準の浮動小数点拡張ではトラップは発生しません)。割り込みが生成され、ハートにルーティングされ、ハートによって有効にされる方法は、EEIに依存します。

*我々が使用している ``exception""と ``trap""はIEEE-754の浮動小数点規格と互換性があります。*

トラップがどのように処理され、ハート上で実行されているソフトウェアに表示されるかは、実行環境に依存します。実行環境内で実行しているソフトウェアの観点からは、実行時にハートが遭遇したトラップは、4つの異なる効果をもたらします。

**封じ込められたトラップ:** トラップは、実行環境内で実行されているソフトウェアから見え、処理されます。例えば、ハー ト上でスーパバイザモードとユーザモードの両方を提供する EEI では、ユーザモードのハー トによる ECALL は、一般的に同じハー ト上で実行されているスーパバイザモードのハンドラへの制御の転送になります。同様に、同じ環境でハートが割り込みを受けると、割り込みハンドラはそのハート上でスーパーバイザモードで実行されます。

**要求されたトラップ：** トラップは、実行環境への明示的な呼び出しで、実行環境内のソフトウェアに代わってアクションを要求する同期例外です。例としては、システムコールがあります。この場合、要求されたアクションが実行環境によって実行された後、ハート上で実行が再開される場合と再開されない場合があります。例えば、システムコールは、ハートを削除したり、実行環境全体を整然と終了させることができます。

**見えないトラップ: トラップ**は実行環境によって透過的に処理され、トラップが処理された後は通常通り実行が再開されます。例としては、欠落した命令のエミュレート、デマンドページ化された仮想メモリシステムでの非常駐ページフォルトの処理、マルチプログラムされたマシンでの別のジョブのためのデバイス割り込みの処理などがあります。これらの場合、実行環境内で実行しているソフトウェアはトラップを認識しません（これらの定義ではタイミングの影響は無視します）。

**致命的なトラップ:** このトラップは致命的な失敗を表し、実行環境が実行を終了させる原因となります。例としては、仮想メモリのページ保護チェックに失敗したり、ウォッチドッグタイマの期限切れを許可したりすることが挙げられます。各 EEI は、実行がどのように終了し、外部環境に報告されるかを定義する必要があります。

各種類のトラップの特徴は以下の通りです。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 含まれている | 要求された | 目に見えない | 致命的な |
| 実行終了？ | N | N1 | N | Y |
| ソフトウェアは気づかない？ | N | N | Y | Y2 |
| 環境に左右される？ | N | Y | Y | Y |

表 1.1.1.トラップの特性。注意事項。1) 終了が要求されることがある。2) 不正確な致命的トラップはソフトウェアによって観測される可能性がある。

EEIは各トラップが正確に処理されるかどうかを定義していますが、可能な限り正確さを維持することを推奨します。含まれているトラップや要求されたトラップは、実行環境内のソフトウェアによって不正確であることが観察されます。目に見えないトラップは、定義上、実行環境内で実行されているソフトウェアによって正確さや不正確さを観察することはできません。致命的なトラップは、既知のエラー命令が即時終了を引き起こさない場合、実行環境内で動作するソフトウェアによって不正確であることが観察されます。

このドキュメントでは特権を持たない命令を記述しているため、トラップについてはほとんど言及されていません。含まれているトラップを処理するためのアーキテクチャ手段は、より豊富な EEI をサポートするための他の機能とともに、特権アーキテクチャマニュアルで定義されています。要求されたトラップを発生させるためだけに定義された特権的でない命令は、ここで文書化されています。目に見えないトラップは、その性質上、このドキュメントの対象外です。ここで定義されておらず、他の手段で定義されていない命令エンコーディングは、致命的なトラップを引き起こすかもしれません。

## 1.7　未定義の行動と値

アーキテクチャは、実装が何をしなければならないか、そして何をしてもよいかという制約を完全に記述しています。アーキテクチャが意図的に実装を制約しない場合には、unspecifiedという用語が明示的に使用されます。

未定義という用語は、意図的に制約されていない動作や値を指します。これらの動作や値の定義は、拡張、プラットフォーム標準、または実装に依存します。拡張機能、プラットフォーム標準、または実装文書は、基本アーキテクチャが不特定と定義するケースをさらに制約するための規範的な内容を提供することがあります。

基本アーキテクチャと同様に、拡張機能は許容される動作と値を完全に記述しなければならず、意図的に制約を受けないケースには未定義という用語を使うべきです。これらのケースは制約されているか、他の拡張機能、プラットフォーム標準、または実装によって定義されているかもしれません。

**第2章**

# RV32I基底整数命令セット、バージョン2.1

この章では、RV32I基底整数命令セットのバージョン2.0について説明します。

*RV32Iは、コンパイラターゲットを形成するのに十分で、最新のオペレーティングシステム環境をサポートするように設計されていました。また、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40個のユニークな命令が含まれていますが、単純な実装ではECALL/EBREAK命令を1つのSYSTEMハードウェア命令でカバーし、常にトラップすることができ、FENCE命令をNOPとして実装することができ、ベースとなる命令数を合計38個に減らすことができます。RV32Iは、他のほとんどのISA拡張をエミュレートすることができます（ただし、アトミック性のために追加のハードウェアサポートを必要とするA拡張を除く）。*

*実際には、マシンモード特権アーキテクチャを含むハードウェア実装では、6つのCSR命令も必要になります。*

*ベースの整数ISAのサブセットは教育的な目的には有用かもしれませんが、ベースは実際のハードウェア実装をサブセットする動機がほとんどないように定義されており、ミスアラインメントメモリアクセスのサポートを省略し、すべてのSYSTEM命令を1つのトラップとして扱うことを超えてはなりません。*

*RV32Iの解説のほとんどは、RV64Iのベースにも適用されます。*

## 2.1　基底整数ISAのプログラマモデル

図2.1にベース整数ISAの非特権状態を示します。RV32I の場合、32 x レジスタはそれぞれ32 ビッ ト幅、すなわち XLEN=32 です。汎用レジスタx1～x31には、各種命令がブール値の集合として解釈する値や、符号付き2進整数や符号なし2進整数の2の補数として解釈する値が格納されています。

さらに1つの非特権レジスタがあります：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ています。

XLEN-1 0

|  |
| --- |
| x0 / ゼロ |
| x1 |
| x2 |
| x3 |
| x4 |
| x5 |
| x6 |
| x7 |
| x8 |
| x9 |
| x10 |
| x11 |
| x12 |
| x13 |
| x14 |
| x15 |
| x16 |
| x17 |
| x18 |
| x19 |
| x20 |
| x21 |
| x22 |
| x23 |
| x24 |
| x25 |
| x26 |
| x27 |
| x28 |
| x29 |
| x30 |
| x31 |

ＸＬＥＮ

XLEN-1 0

ＰＣ

ＸＬＥＮ

図2.1.RISC-Vベースの非特権整数レジスタの状態。

*基本整数 ISAには専用のスタックポインタやサブルーチンのリターンアドレスリンクレジスタはありません。しかし、標準的なソフトウェアの呼び出し規約では、呼び出しのリターンアドレスを保持するためにレジスタ*x1を*使用し、代替リンクレジスタとしてレジスタ*x5を使用することができます。*標準的な呼び出し規則では、スタックポインタとしてレジスタ*x2を使用します。

*ハードウェアは、*x1 *または* x5 を*使用する関数の呼び出しや戻り値を高速化することを選択する場合があります。JAL および JALR 命令の説明を参照してください。*

*オプションの圧縮16ビット命令フォーマットは、*x1が*リターンアドレスレジスタ、*x2が*スタックポインタであることを前提に設計されています。他の規約を使用しているソフトウェアは正しく動作しますが、コードサイズが大きくなる可能性があります。*

*利用可能なアーキ 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 ギー消費量に大きな影響を与え 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あることは間違いありませんが、3つのアドレス形式を使用して16ビット命令で16個のレジスタを持つ完全な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だが、命令数が増えて効率が悪くなる。私たちは、ベースとなるハードウェアの実装を簡単にするために、（Xtensaの24ビット命令のような）中間の命令サイズは避けたいと考えており、32ビット命令サイズを採用すれば、32個の整数レジスタを簡単にサポートすることができます。整数レジスタの数を増やすことで、ループアンローリング、ソフトウェアパイプライン、キャッシュタイリングなどを多用する高性能コードのパフォーマンスを向上させることができます。*

*これらの理由から、ベースのISAには従来の32整数レジスタのサイズを選択した。動的レジスタの使用は、頻繁にアクセスされる少数のレジスタに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へのアクセスエネルギーを削減するように最適化することができます[20]。オプションの圧縮された16ビット命令フォーマットは、ほとんどの場合、8個のレジスタにしかアクセスできないため、密な命令エンコーディングを提供することができます。*

*必要に応じて、はるかに大きなレジスタスペース（フラットまたは階層構造のいずれか）を使用することができます。*

*リソースに制約のある組み込みアプリケーションのために、16個のレジスタしか持たないRV32Eサブセットを定義しました（第4章）。*

## 2.2 基本命令形式

ベースのRV32I 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。すべての命令は固定の32ビット長で、メモリ内の4バイト境界にアラインメントされていなければなりません。ターゲットアドレスが4バイトアラインメントされていない場合、取り込まれた分岐または無条件ジャンプで、命令アドレスがミザラインされた例外が発生します。この例外は、分岐命令またはジャンプ命令で報告され、ターゲット命令では報告されません。取られていない条件分岐では、命令アドレスが一致しない例外は発生しません。

*ベースのISA命令のアラインメント制約は、16ビット長または16ビット長の他の奇数倍数の命令拡張が追加された場合（すなわち、IALIGN=16）、2バイト境界まで緩和されます。*

*命令アドレスミスアライメント例外は、デバッグを助けるために命令ミスアライメントの原因となる分岐やジャンプで報告され、また、これらがミスアライメントが発生する唯一の場所であるIALIGN=32のシステムのためのハードウェア設計を単純化します。*

予約命令をデコードする際の動作は不定です。

*プラットフォームによっては、標準使用のために予約されたオペコードが不正命令例外を発生させることを要求することがあります。他のプラットフォームは、予約されたオペコード空間が不適合な拡張のために使用されることを許可してもよい。*

RISC-V ISA では、デコードを簡単にするために、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形式で同じ位置に配置しています。CSR命令(第9章)で使用される5ビットの即時子を除いて、即時子は常に符号拡張されており、一般的に命令の左端の利用可能なビットに向かって詰められ、ハードウェアの複雑さを減らすために割り当てられています。特に、符号拡張回路を高速化するために、すべての即時値の符号ビットは常に命令のビット31にあります。

*レジスタ指定子のデコードは通常、実装ではクリティカルパス上にあるため、命令フォーマットはすべてのフォーマットですべてのレジスタ指定子を同じ位置に保つように選択されていますが、フォーマット間で即時ビットを移動させる必要があります（RISC-IV、別名SPUR [11]と共有されている特性）。*

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| func7 | rs2 | rs1 | funct3 | rd | オペコード | R型 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | オペコード | I型 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | オペコード | S型 |

|  |  |  |  |
| --- | --- | --- | --- |
| imm[31:12] | rd | オペコード | U型 |

図2.2. RISC-V の基本命令フォーマット。各即時サブフィールドには、通常のように命令の即時フィールド内のビット位置ではなく、生成される即時値のビット位置（imm[x ]）がラベル付けされています。

*実際には、ほとんどの即時命令は小さいか、すべてのXLENビットを必要とします。我々は、通常の命令で利用可能なオペコードスペースを増やすために、非対称即時分割（通常の命令の12ビットに加えて、20ビットの特別なロード上位即時命令）を選択しました。*

*MIPS ISAのようにいくつかの即時性のためにゼロ拡張を使用することの利点が見られず、ISAを可能な限りシンプルに保ちたかったため、即時性は符号拡張されている。*

## 2.3　即時符号化バリアント

図2.3に示すように、即時性の取り扱いに基づく命令フォーマット(B/J)には、さらに2つのバリエーションがあります。

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| func7 | rs2 | rs1 | funct3 | rd | オペコード | R型 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | オペコード | I型 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | オペコード | S型 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | オペコード | B型 |

|  |  |  |  |
| --- | --- | --- | --- |
| imm[31:12] | rd | オペコード | U型 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | オペコード | R型 |

図2.3: 即時バリアントを示すRISC-V基本命令フォーマット

S 形式 と B 形式の唯一の違いは、B 形式では12 ビッ トのイミディエイト フィールドが分岐オフセットを2 の倍数でエンコードするために使用されている点です。命令エンコードされたイミディエイトの全ビットを従来のようにハードウェアで左に1つずつシフトするのではなく、中間ビット(imm[10:1])と符号ビットは固定位置にとどまり、S フォーマットの最下位ビット(inst[7] )は B フォーマットの高次ビットをエンコードします。

同様に、U 形式 と J 形式の唯一の違いは、20 ビッ トのイミディエイトを左に12 ビッ トずらしてU 形式のイミディエイトを形成し、1 ビッ トずらしてJ 形式のイミディエイトを形成することです。ＵフォーマットとＪフォーマットのイミディエイトにおける命令ビットの位置は、他のフォーマットとの重複を最大にするために、また、他のフォーマットとの重複を最大にするために選択されます。

図2.4は、それぞれの基本命令フォーマットによって生成される即時値を示しており、どの命令ビット（inst[*y*]）が即時値の各ビットを生成するかを示すラベルが付けられています。

31 30 20 19 12 11 10 5 4 1 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ― imm[31] ― | inst[30:25] | inst[24:21] | inst[20] | I即値 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ― imm[31] ― | inst[30:25] | inst[11:8] | inst[7] | S即値 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ― inst[31] ― | inst[7] | inst[30:25] | inst[11:8] | 0 | B即値 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| inst[31] | inst[30:20] | inst[19:2] | ― 0 ― | U即値 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ― inst[31] ― | inst[19:12] | inst[20] | inst[30:25] | inst[24:21] | 0 | J即値 |

図2.4: RISC-V命令で生成されるイミディエイトの種類。フィールドには、その値を構築するために使用される命令ビットが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XLEN>32)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は常に命令のビット31に保持され、命令のデコードと並行して符号拡張が行われるようになっています。*

*より複雑な イ ンプ リ メ ン ト では、 分岐計算 と ジャンプ計算のために別々の加算器を使用する場合があ り 、 命令の種類に よ っ て イ ン タ イ ミ ン テ ィ ブ ビ ッ ト の位置を一定に保つ こ と の利点はないかも し れませんが、 最も単純な イ ンプ リ メ ン ト のハー ド ウ ェ ア コ ス ト を削減 し たい と考えました。B と J の命令エンコーディングのビットを回転させることで、動的なハー ド ウ ェ ア マ ッ ク ス を使用 し て イ ン タ イ ミ ン グ を 2 倍にす る のではな く 、 命令信号のファンアウト と イ ン タ イ ミ ン グ マ ッ ク ス の コ ス ト を約 2 倍に削減 し ま し た。 スクランブルされた イ ン タ イ ミ ン グ エン コ ーデ ィ ン グは、 ス タ テ ィ ッ ク ア ッ プ ま たは時間前のコンパイルにごくわずかな時間を追加します。命令を動的に生成する場合、若干の追加オーバヘッドがありますが、最も一般的なショートフォワードブランチには簡単な即時エンコーディングがあります。*

## 2.4　整数演算命令

ほとんどの整数演算命令は、整数レジスタファイルに保持されている値のXLENビットを操作します。整数計算命令は、I型フォーマットを使用したレジスター即値演算、またはR型フォーマットを使用したレジスター演算としてエンコードされています。宛先は、register-immediate命令、register-register命令ともに、register *rd*となります。整数演算命令では演算例外は発生しません。

*多くのオーバーフロー検査はRISC-V ブランチを使用して安価に実装できるため、整数演算のオーバーフロー検査のための特別な命令セットのサポートは基本命令セットには含めませんでした。*

*符号なし加算のオーバーフロー検査は、加算後に 1 つの追加の分岐命令を必要とするだけです:* add t0, t1, t2; bltu t0, t1, overflow*。*

*符号付き加算の場合、オペランドの符号が既知の場合、オーバーフローチェックは加算後に1つの分岐を必要とするだけです：*addi t0, t1, +imm; blt t0, t1, overflow*。これは、即時オペランドを持つ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では、他のオペランドが負の場合にのみ、和がオペランドの一方よりも小さくなるという観測を利用して、加算後に3つの追加命令が必要となります。*

add t0, t1, t2

slti t3, t2, 0

slt t4, t0, t1

bne t3, t4, overflow

*RV64Iでは、オペランド上の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することができます。*

**整数レジスタ-初歩命令**

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | オペコード |

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 はレジスタ rs1 に符号拡張 12 ビットのイミディエイトを追加します。算術オーバーフローは無視され、結果は単に結果の下位XLENビットです。ADDI rd, rs1, 0 は MV rd, rs1 アセンブラ擬似命令を実装するために使用されます。

SLTI (set less than immediate) は、両方が符号付き数値として扱われるとき、レジスタ *rs1 が符号拡張され*たイミディエイトよりも小さい場合、レジスタ *rd* に値 1 を配置し、そうでない場合は 0 が *rd* に書き込まれます。SLTIUは似ていますが、値を符号なし数字として比較します（すなわち、即時は最初にXLENビットに符号拡張され、その後符号なし数字として扱われます）。注意：SLTIU *rd, rs1, 1 は、rs1 が*ゼロに等しい場合は *rd を* 1 にセットし、そうでない場合は *rd を* 0 にセットします (アセンブラ擬似命令 SEQZ *rd, rs*)。

ANDI, ORI, XORI は、レジスタ *rs1* と符号拡張 12 ビット即時に対してビット単位で AND, OR, XOR を実行し、その結果を *rd* に格納する論理演算です。注意: XORI *rd, rs1,-1 は*レジスタ *rs1 の*ビット単位の論理反転を行います (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 | オペコード |

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 SLAI dest OP-IMM

定数によるシフトは、I型フォーマットの特殊化として符号化されます。シフトされるオペランドは*rs1*にあり、シフト量はI-immediateフィールドの下位5ビットに符号化されます。右シフト型はビット30にエンコードされています。ＳＬＬＩＩは論理的な左シフト（ゼロは下位ビットにシフトされる）、

ＳＲＬＩは論理的な右シフト（ゼロは上位ビットにシフトされる）、ＳＲＡＩは算術的な右シフト（元の符号ビットは空になった上位ビットにコピーされる）である。

31 12 11 7 6 0

|  |  |  |
| --- | --- | --- |
| imm[３１:12] | rd | オペコード |

20 ５ ７

U-即値[31:12] dest LUI

U-即値[31:12] dest AUIPC

LUI (load upper immediate) は 32 ビットの定数を構築するために使用され、U 型フォーマットを使用します。LUI は、宛先レジスタ *rd* の上位 20 ビットに U-immediate 値を配置し、下位 12 ビットをゼロで埋めます。

AUIPC (add upper immediate to pc) は、pc-relative address を構築するために使用され、U タイプのフォーマットを使用します。AUIPCは20ビットのU-immediateから32ビットのオフセットを形成し、最下位12ビットをゼロで埋め、このオフセットをAUIPC命令のアドレスに加算し、その結果をレジスタ*rd*に格納します。

*AUIPC 命令は、制御フロー転送とデータアクセスの両方で PC から任意のオフセットにアクセスするための 2 つの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の組み合わせは、任意の32ビットPC関連アドレスに制御を転送することができ、AUIPCと通常のロード命令やストア命令の12ビットイミディエイトオフセットは、任意の32ビットPC関連データアドレスにアクセスすることができます。*

*現在の PC は U-immediate を 0 に設定することで取得できます。 JAL +4 命令を使用してローカル PC (JAL に続く命令の) を取得することもできますが、単純なマイクロアーキテクチャーではパイプラインが途切れたり、複雑なマイクロアーキテクチャーでは BTB 構造を汚染したりする可能性があります。。*

**整数レジスタの操作**

RV32Iは、いくつかの算術Rタイプの演算を定義しています。すべての演算は、ソース・オペランドとしてrs*1*と*rs2レジスタを*読み出し、その結果をレジスタ*rd*に書き込みます。*funct7*と*funct3*フィールドは演算のタイプを選択します。

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct7 | rs2 | rs1 | funct3 | rd | オペコード |

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

0000000 src2 ｓｒｃ1 SUB/SRA dest OP

ADD は、rs*1* と rs*2 の*加算を実行します。SUB は rs*1* から r*s2 の減算を行います*。オーバーフローは無視され、結果の XLEN の下位ビットが宛先 *rd* 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、rs*1 < rs2 の*場合は 1 を *rd* に書き込み、そうでない場合は *0 を*書き込みます。

注意: SLTU *rd*, *x0*, *rs2 は、rs2 が* 0 でない場合は *rd を* 1 にセットし、そうでない場合は *rd を* 0 にセットします (アセンブラ擬似命令 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。

SLL,SRL,SRA は、レジスタ *rs1 の*値に対して、レジスタ rs*2 の*下位 5 ビットに保持されているシフト量により、論理左シフト、論理右シフト、算術右シフトを行います。

**NOP命令**

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | オペコード |

１２ ５ ３ ５ ７

0 0 ＡＤＤＩ 0 OP-IMM

NOP 命令は、PC を前進させ、該当するパフォーマンス・カウンタをインクリメントすることを除いて、アーキ テクチャ的に見え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。

*NOP は、コードセグメントをマイクロアーキテクト的に重要なアドレス境界に合わせたり、インラインコードの修正のためのスペースを残したりするために使用することができます。NOP をエンコードする方法はたくさんありますが、我々は、より読みやすい分解出力のためだけでなく、マイクロアーキテクチャーの最適化を可能にするために、正統的な NOP エンコーディングを定義します。他のNOPエンコーディングは、HINT命令（2.9項）で使用可能になっています。*

*ADDIはNOPエンコーディングのために選択されましたが、これは（デコードで最適化されていなければ）様々なシステムで実行するために最も少ないリソースを必要とする可能性が高いためです。特に、命令は1つのレジスタを読み込むだけです。また、ADDI 機能ユニットは、加算が最も一般的な動作であるため、スーパースカラ ー設計で利用できる可能性が高くなります。特に、アドレス生成機能ユニットは、ベース+オフセット・アドレス計算に必要なのと同じハードウェアを使用してADDIを実行することができますが、レジスタレジスタADDまたは論理/シフト演算は追加のハードウェアを必要とします。*

## 2.5　制御転送手順

RV32Iには、無条件ジャンプと条件分岐の2種類の制御伝達命令があります。RV32Iの制御伝達命令には、アーキテクチャ的に見える遅延スロットがあり*ません。*

**無条件ジャンプ**

ジャンプ・リンク(JAL)命令はJ型フォーマットを使用し、J-immediateは2バイトの倍数で符号付きオフセットをエンコードします。オフセットは符号付きで拡張され、ジャンプ命令のアドレスに追加され、ジャンプターゲットアドレスとなります。したがって、ジャンプは±1MiBの範囲をターゲットにすることができます。JALは、ジャンプに続く命令(pc+4)のアドレスをレジスタ*RDに*格納します。標準的なソフトウェアの呼び出し規則では、x1をリターンアドレスレジスタ、x5をオルタネートリンクレジスタとして使用しています。

*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（例えば、圧縮コード内のレジスタを保存したり復元したりするもの）の呼び出しをサポートしています。レジスタx5は、標準の呼び出し規約ではテンポラリにマップされ、通常のリンクレジスタとは1ビットしか異なるエンコーディングを持っているため、代替リンクレジスタとして選択されました。*

プレーンな無条件ジャンプ(アセンブラ擬似命令J)は、*rd*=x0のJALとしてエンコードされています。

31 ３０ 21 20 １９ 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | オペコード |

１ １０ １ 8 ５ ７

オフセット[20:1] dest JAL

間接ジャンプ命令JALR(ジャンプアンドリンクレジスタ)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*符号拡張された 12 ビットの I-immediate を加算し、その結果の最下位ビットを 0 にすることで得られます。ジャンプ(pc+4)に続く命令のアドレスがレジスタ *rd* に書き込まれます。結果が不要な場合は、レジスタx0を宛先として使用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 | オペコード |

１２ ５ ３ ５ ７

オフセット[11:0] base 0 dest JALR

*無条件ジャンプ命令はすべて、位置に依存しないコードをサポートするために PC 相対アドレッシングを使用しています。JALR 命令は、2 つの命令シーケンスで 32 ビット絶対アドレス範囲内の任意の場所にジャンプできるように定義されています。LUI 命令は、最初にターゲット アドレスの上位 20 ビットを* rs1 に*ロードしてから、JALR は下位ビットを追加します。同様に、AUIPCと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、条件分岐命令とは異なり、12 ビットのイミディエイトを 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。これにより、ハードウェアでのイミディエイトのフォーマットが1つ増えるのを避けることができます。実際には、JALR のほとんどの用途では、ゼロ即時を持つか、LUI または AUIPC とペアになるので、わずかな範囲の減少は重要ではありません。*

*JALRターゲットアドレスを計算する際に最下位ビットをクリアすることは、ハードウェアを少し単純化し、ファンクションポインタの下位ビットを補助情報を格納するために使用することを可能にします。この場合、エラーチェックが若干失われ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しても、通常はすぐに例外が発生します。*

*ベースの*rs1=x0*で使用される場合、JALRは、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するために使用することができ、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することができます。あるいは、ABIは、アドレス空間の他の場所にあるライブラリを指すように汎用レジスタを割り当てることができます。*

JALおよびJALR命令は、ターゲットアドレスが4バイト境界に整列していない場合、命令-アドレス-misaligned例外を生成します。

*命令-アドレス-misaligned例外は、圧縮命令セット拡張Cのような16ビットのアラインメント命令を持つ拡張機能をサポートしているマシンでは使用できません。*

リターンアドレス予測スタックは高性能な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や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では、使用されるレジスタ番号を介して暗黙のうちに命令の使用方法のヒントが符号化されています。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)にプッシュしなければなりません。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。

*他のいくつかのISAでは、リターンアドレススタックの操作をガイドするために、間接ジャンプ命令に明示的なヒントビットを追加しています。我々は、これらのヒントに使用されるエンコーディングスペースを削減するために、レジスタ番号と呼び出し規則に関連付けられた暗黙のヒントを使用しています。*

|  |  |  |  |
| --- | --- | --- | --- |
| rs | rs1 | rs1=rd | ＲＡＳ作用 |
| *！リンク* | *！リンク* | - | 無し |
| *！リンク* | *リンク* | - | ポップ |
| *リンク* | *！リンク* | - | プッシュ |
| *リンク* | *リンク* | 0 | ポップしてからプッシュ |
| *リンク* | *リンク* | 1 | プッシュ |

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のいずれかの場合に*リンクが*trueになります。

*2つの異なるリンクレジスタ（*x1*と*x5*）が*rs1*と*rd*として与えられた場合、RASはポップされ、コアーチンをサポートするためにプッシュされます。rs*1*とrd*が*同じリンクレジスタ(x*1*または*x5の*いずれか)である場合、RASは、以下のシーケンスのマクロオープンフュージョンを可能にするためにのみプッシュされます：*lui ra, imm20; jalr ra, imm12(ra)*と*auipc ra, imm20; jalr ra, imm12(ra)

**条件分岐**

すべての分岐命令はB型命令フォーマットを使用します。12 ビットの B-即値 は、2 バイトの倍数で符号付きオフセットをエンコードします。オフセットは符号拡張され、分岐命令のアドレスに追加され、ターゲット・アドレスとなります。条件分岐範囲は±4KiBです。

31 30 25 24 20 19 15 14 12 11 8 7 6 0

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| imm[12] | imm[10:5] | rs2 | ｒｓ１ | funct3 | imm[4:1] | imm[11] | オペコード |

１ 6 ５ 5 ３ 4 1 7

オフセット[12|10:5] ｓｒｃ2 src1 BEQ/BNE offset[11|4:1] 分岐

オフセット[12|10:5] ｓｒｃ2 src1 BLT[U] offset[11|4:1] 分岐

オフセット[12|10:5] ｓｒｃ2 src1 BGE[U] offset[11|4:1] 分岐

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 rs*2 が*それぞれ等しい場合とそうでない場合に分岐します。BLT、BLTUは、rs*1がrs2*より小さい場合、それぞれ符号付き比較、符号なし比較を使用して分岐します。BGE、BGEU は、rs*1* が rs*2* 以上の場合は、それぞれ符号付き比較、符号なし比較を使用して分岐します。なお、BGT、BGTU、BLE、およびBLEUは、オペランドをそれぞれBLT、BLTU、BGE、BGEUに反転させて合成することができます。

*符号付き配列の境界は、負のインデックスがあれば、非負の境界よりも大きく比較されるため、1つのBLTU命令でチェックすることができます。*

ソフトウェアは、連続したコードパスが最も一般的なパスであり、あまり頻繁に取られることのないコードパスは行外に配置されるように最適化されなければなりません。また、ソフトウェアは、少なくとも最初に遭遇したときには、後方への分岐は取られると予測され、前方への分岐は取られないと仮定する必要があります。動的予測器は、予測可能な分岐の振る舞いを素早く学習しなければなりません。

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、常に真の条件を持つ条件分岐命令ではなく、常に無条件分岐に使用されるべきです。RISC-VジャンプはPC関連でもあり、分岐よりもはるかに広いオフセット範囲をサポートしており、条件分岐予測テーブルを汚すことはありません。

*条件分岐は、条件コード(x86, ARM, SPARC, PowerPC)を使用したり(Alpha, MIPS)、1つのレジスタをゼロと比較したり(Alpha, MIPS)、2つのレジスタを等しく比較したり(MIPS)するのではなく、(PA-RISC, Xtensa, MIPS R6でも行われているように)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います。この設計は、比較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や一時レジスタの使用を回避し、スタティックコードサイズと動的命令フェッチのトラフィックを削減するという観察から動機づけられました。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スタティック ロジッ クに移行した後には）技術的でない回路遅延が必要となるため、算術的な大きさの比較とほぼ同じくらいのコストがかかるということです。フューズドの比較分岐命令のもう1つの利点は、フロントエンドの命令スト リームで分岐が早期に観測されるため、分岐を早期に予測できることです。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、条件コードを使用した設計の利点があるかもしれませんが、このようなケースは比較的まれだと考えています。*

*我々は、命令エンコーディングに静的分岐ヒントを含めることを検討しましたが、考慮しませんでした。これらは動的予測器への負荷を軽減することができますが、最良の結果を得るためには、より多くの命令エンコーディングスペースとソフトウェアプロファイリングが必要となり、本番ランがプロファイリングランと一致しない場合にはパフォーマンスが低下する可能性があります。*

*予測不可能な短い前方分岐を効果的に置き換えることができる条件付き移動や述語命令を検討しましたが、これは含まれていませんでした。条件付きムーブはこの2つのうちで最も単純ですが、例外が発生する可能性のある条件付きコード（メモリアクセスや浮動小数点演算）では使用が困難です。述語化は、システムにフラグの状態を追加し、フラグを設定したりクリアしたりする命令を追加し、すべての命令にエンコーディングの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とも、アウトオブオーダーのマイクロアーキテクチャーに複雑さを追加し、述語がFalseの場合には、宛先のアーキテクチャー・レジスタの元の値をリネームされた宛先の物理レジスタにコピーする必要があるため、暗黙の第3のソース・オペランドが追加されます。また、ブランチの代わりに述語を使用するという静的なコンパイル時の決定は、コンパイラ・トレーニング・セットに含まれていない入力に対してパフォーマンスを低下させる可能性があります。*

*予測不可能なショートフォワードブランチを内部的に予測されたコードに動的に変換し、ブランチの誤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の様々なマイクロアーキテクチャー技術が存在しており[6, 10, 9]、商用プロセッサに実装されています[17]。最も単純な技術は、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のみをフラッシュするか、広い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で、誤った予測されたショートフォワードブランチから回復する際のペナルティを軽減するだけです。アウトオブオーダー・コアのためのより複雑な技術では、分岐シャドウ内の命令に内部述語を追加し、内部述語の値を分岐命令が書き込んで、分岐命令とそれに続く命令を他のコードと比較して推測的にアウトオブオーダーで実行できるようにします[17]。*

条件分岐命令は、ターゲットアドレスが4バイト境界にアラインメントされておらず、分岐条件が真と評価された場合、命令-アドレス-ミスアラインド例外を発生させます。分岐条件の評価が false の場合、命令-アドレス-ミスアラインド例外は発生しません。

*命令-アドレス-misaligned例外は、圧縮命令セット拡張Cのような16ビットのアラインメント命令を持つ拡張機能をサポートしているマシンでは使用できません。*

## 2.6　ロードとストアの命令

RV32Iはロードストア・アーキテクチャで、ロード命令とストア命令のみがメモリにアクセスし、演算命令はCPUレジスタ上でのみ動作する。RV32Iは、バイトアドレスの32ビットアドレス空間を提供します。EEIは、アドレス空間のどの部分がどの命令でアクセスしても問題ないかを定義します(例えば、一部のアドレスは読み取り専用になっていたり、ワードアクセスのみをサポートしていたりする場合があります)。宛先がx0のロードは、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さなければなりません。

EEIはメモリシステムがリトルエンディアンかビッグエンディアンかを定義します。RISC-Vでは、エンディアンはバイトアドレス不変です。

*エンディアンがバイトアドレス不変であるシステムでは、次のような特性を持ちます：あるエンディアンのあるアドレスでバイトがメモリに保存されている場合、どのエンディアンでもそのアドレスからバイトサイズのロードを行うと、保存された値が返されます。*

*リトルエンディアン構成では、マルチバイトストアは最も重要度の低いメモリバイトアドレスに最も重要度の低いレジスタバイトを書き込み、その後、重要度の高い順に他のレジスタバイトを書き込みます。ロードも同様に、重要度の低いメモリバイトアドレスの内容を重要度の低いレジスタバイトに転送します。*

*ビックエンディアン構成では、マルチバイトストアは最も重要度の高いレジスタバイトを最も低いメモリバイトアドレスに書き込み、その後、重要度の高い順に他のレジスタバイトを書き込みます。ロードも同様に、重要度の高い方のメモリバイトアドレスの内容を重要度の低い方のレジスタバイトに転送します。*

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | オペコード |

１２ ５ ３ ５ ７

オフセット[11:0] base 幅 dest ロード

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | オペコード |

7 5 ５ ３ ５ ７

オフセット[11:5] ｓｒｃ base 幅 オフセット[4:0] ストア

ロード命令とストア命令は、レジスタとメモリの間で値を転送します。ロードはI型フォーマットでエンコードされ、ストアはS型です。有効アドレスは、レジスタ *rs1 を*符号拡張された 12 ビットのオフセットに加算することで得られます。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 32 ビットの値を *rd* にロードします。LHはメモリから16ビットの値をロードした後、符号を32ビットに拡張してから*rdに格納します*。LHUはメモリから16ビットの値をロードした後、ゼロが32ビットに拡張されてから*rdに格納されます*。LBとLBUは8ビットの値に対して類似して定義されています。SW、SH、およびSB命令は、レジスタ*rs2*のロービットからメモリに32ビット、16ビット、および8ビットの値を格納します。

EEIにかかわらず、実効アドレスが自然に整列されているロード及びストアは、アドレス・ミスアライン例外を発生させてはなりません。

実効アドレスが参照されるデータ型に自然に整列されていない（すなわち、32ビットアクセスの場合は4バイト境界上に、16ビットアクセスの場合は2バイト境界上に）ロード及びストアは、EEIに依存した動作をします。

EEIは、ロードとストアの位置ずれが完全にサポートされていることを保証する場合があり、実行環境内で実行しているソフトウェアは、含まれていたり、致命的なアドレスの位置ずれトラップを経験することはありません。この場合、ロードとストアの不整合は、ハードウェアで処理するか、実行環境実装への不可視トラップを介して処理するか、あるいは、アドレスに応じてハードウェアと不可視トラップの組み合わせで処理することができます。

EEIは、不整合なロードとストアが目に見えないように処理されることを保証しない場合があります。この場合、自然に整列されていないロードとストアは、正常に実行を完了するか、例外を発生させるかのどちらかになります。発生した例外は、アドレス整合性のない例外かアクセス障害の例外のどちらかになります。アラインメントのずれがなければ完了するはずのメモリアクセスに対して、アラインメントのずれがエミュレートされてはならない場合、例えば、メモリ領域へのアクセスが副作用をもたらす場合などには、アドレス整合性のない例外の代わりにアクセス例外を発生させることができます。EEIがミスアラインメントされたロードとストアが目に見えないように処理されることを保証しない場合、EFIはアドレスミスアラインメントによって引き起こされた例外が、インクルードトラップ（実行環境内で実行しているソフトウェアがトラップを処理することを可能にする）または致命的なトラップ（実行を終了する）になるかどうかを定義しなければなりません。

*レガシーコードを移植する際には、時折不整合アクセスが必要とされ、任意の形式のpacked-SIMD拡張を使用したり、外部的にpackedされたデータ構造を処理したりする際に、アプリケーション上でのパフォーマンスを向上させます。EEIが通常のロードとストア命令を介して不整合アクセスをサポートすることを選択できるようにした私たちの論理的根拠は、不整合ハードウェアサポートの追加を単純化することです。1つの選択肢としては、ベースISAでの不整合アクセスを禁止し、その後、ソフトウェアが不整合アクセスを処理するのを助ける特別な命令、または不整合アクセスのための新しいハードウェア・アドレッシング・モードのいずれかの不整合アクセスのための別個のISAサポートを提供することが考えられます。特別な命令は使用が難しく、ISAを複雑にし、しばしば新しいプロセッサ状態を追加したり（SPARC VISのアラインメントアドレスオフセットレジスタなど）、既存のプロセッサ状態へのアクセスを複雑にしたり（MIPS LWL/LWRパーシャルレジスタ書き込みなど）します。さらに、ループ指向のパックドＳＩＭＤコードの場合、オペランドがミスアライメントされているときの余分なオーバーヘッドは、オペランドのアライメントに応じて複数の形態のループを提供するようにソフトウェアを動機付け、これはコード生成を複雑にし、ループ起動時のオーバーヘッドを追加する。新しい誤整合ハードウェアアドレッシングモードは、命令エンコーディングに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。*

アラインメントされていないロードとストアが正常に完了したとしても、実装によっては、これらのアクセスの実行が非常に遅くなる可能性があります（例えば、見えない・トラップを介して実装されている場合など）。さらに、自然にアラインメント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ていますが、アラインメントされていないロードとストアは、アトミック性を保証するために追加の同期を必要とする可能性があります。

*私たちは、ミスアライメントされたアクセスに対してアトミック性を義務づけていないので、実行環境の実装では、不可視のマシン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ミスアライメントされたアクセスを処理することができます。もしハードウェアの不整合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とストア命令を使用するだけで、これを利用することができます。ハードウェアは、ランタイムアドレスがアラインメントされているかどうかに応じて、アクセスを自動的に最適化することができます。*

## 2.7　メモリ順命令

31 28 27 26 25 24 23 22 21 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| fm | PI | PO | PR | PW | SI | SO | SR | SW | rs1 | func3 | rd | オペコード |

4 1 1 1 1 1 1 1 1 5 3 5 ７

FM 前任者 後継者 0 FENCE 0 雑多-MEM

FENCE命令は、他のRISC-Vハートや外部デバイスやコプロセッサから見たデバイスI/Oやメモリアクセスの順序を決めるために使用されます。デバイス入力(I)、デバイス出力(O)、メモリリード(R)、メモリライト(W)の任意の組み合わせは、同じものの任意の組み合わせに関して順序付けられます。非公式には、他のRISC-Vハートや外部デバイスは、FENCEに続く後継セット内の任意の操作を、FENCEに先行する前の前任セット内の任意の操作よりも先に観測することはできません。

第14章では、RISC-Vメモリ一貫性モデルの正確な説明があります。

EEIは、どのようなI/O操作が可能かを定義し、特に、ロード命令とストア命令によってアクセスされたときに、どのメモリアドレスがメモリの読み取りと書き込みではなく、デバイスの入力とデバイスの出力操作として扱われ、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ビットとWビットではなく、IビットとOビットを使用して順序付けられた、キャッシュされていないロードとストアでアクセスされます。命令セット拡張は、FENCEのIビットとOビットを使用して順序付けされる新しいI/O命令を記述することもあります。

|  |  |  |
| --- | --- | --- |
| fm フィールド | ニーモニック | 意味 |
| 0000 | 無し | ノーマルフェンス |
| 1000 | TSO | FENCE RW,RW を使用した場合：書き込み-読み取り順序を除外する。  他：*将来の使用のために予約されている。* |
| *他* | | *将来の使用のために予約されています。* |

表2.2.フェンスモードのエンコーディング。

fence モードフィールド *fm は、fence の意味論*を定義します。*fm*=0000 の FENCE は、前任者セットのすべてのメモリ操作を、後任者セットのすべてのメモリ操作の前に命令します。

オプションのFENCE.TSO命令は、*fm*=1000、*predecessor*=RW、*succeedessor*=RWのFENCE命令としてエンコードされています。FENCE.TSOは、前任者セットのすべてのロード操作を後任者セットのすべてのメモリ操作の前に、前任者セットのすべてのストア操作を後任者セットのすべてのストア操作の前に命令します。これにより、FENCE.TSOの前任者セットの非AMOストア操作は、後継者セットの非AMOロードと一緒に順序付けられないままになります。

*FENCE.TSO エンコーディングは、元のベースの FENCE 命令エンコーディングのオプション拡張として追加されました。元の定義では、実装はセットビットを無視して FENCE をグローバルとして扱うことを要求しているので、これは後方互換性のある拡張です。*

FENCE命令の中の未使用のフィールド--*rs1*と*rd*--は、将来の拡張でより細かい粒度のフェンスのため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。同様に、表2.2の多くの*fm*及び前任者/後任者セット設定も、将来の使用のために予約されています。

基本実装は、そのような予約済み構成をすべて fm=0000 の通常のフェンスとして扱い、標準ソフトウェアは、予約済みでない構成のみを使用しなければならない

*私たちは、シンプルなマシンの実装や将来のコプロセッサやアクセラレータの拡張による高性能を可能にするために、緩和されたメモリモデルを選択しました。デバイスドライバハート内での不必要な直列化を避け、追加されたコプロセッサやI/Oデバイスを制御するための代替的な非メモリパスをサポートするために、メモリのR/W順序からI/O順序を分離しています。単純な実装では、さらに前任*者*フィールドと*後任者*フィールドを無視して、すべての操作で常に保守的なフェンスを実行することができます。*

## 2.8　環境コールとブレークポイント

SYSTEM命令は、特権アクセスを必要とする可能性のあるシステム機能にアクセスするために使用され、I型命令フォーマットを使用してエンコードされています。これらの命令は、2つの主要なクラスに分けることができます：制御レジスタおよびステータスレジスタ(CSR)を原子的に読み書きする命令と、その他のすべての潜在的な特権命令です。CSR命令については第9章で説明し、基本的な非特権命令については次のセクションで説明します。

*SYSTEM命令は、よりシンプル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することを可能にするために定義されています。より洗練された実装では、ハードウェアで各システム命令をより多く実行することがあります。*

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| funct12 | rs1 | funct3 | rd | オペコード |

１２ ５ ３ ５ ７

ECALL ０ PRIV 0 SYSTEM

EBREAK ０ PRIV 0 SYSTEM

これら2つの命令は、サポートする実行環境に正確に要求されたトラップを発生させます。

ECALL命令は実行環境へのサービス要求を行うために使用されます。EEIはサービス要求のためのパラメータがどのように渡されるかを定義しますが、通常は整数レジスタファイル内の定義された場所にあります。

EBREAK 命令は、制御をデバッグ環境に戻すために使用されます。

*ECALL と EBREAK は以前は SCALL と SBREAK という名前でした。これらの命令は同じ機能とエンコーディングを持っていますが、スーパーバイザレベルのオペレーティングシステムやデバッガを呼び出すよりも一般的に使用できることを反映して名前が変更されました。*

*EBREAK は、主にデバッガが実行を停止してデバッガにフォールバックさせるために使用するように設計されました。また、EBREAK は標準の gcc コンパイラでも、実行すべきでないコードパスをマークするために使用されています。*

*EBREAK のもう一つの用途は、実行環境にデバッガが含まれており、EBREAK 命令の周りに構築された代替システムコールインタフェースを介してサービスを提供することができる「セミホステ ィング」をサポートすることです。*

*RISC-V ベースの ISA は 1 つ以上の EBREAK 命令を提供しないため、RISC-V のセミホステ ィングでは、セミホステイング EBREAK とデバッガが挿入した EBREAK を区別するために、特別な命令のシーケンスを使用します。*

slli x0, x0, 0x1f ＃ エントリー NOP

ebreak # デバッガへのブレーク

srai x0, x0, 7 # セミホスト呼の番号をエンコードするNOP 7

*これら3つの命令は32ビット幅の命令でなければならないことに注意してください。つまり、第16章で説明されている圧縮16ビット命令の中にあってはなりません。*

*シフトNOP命令は、HINTSとして使用可能な状態であると考えられています。*

*セミホスティングはサービスコールの一形態であり、既存の ABI を使用して ECALL としてより自然にエンコードされますが、そのためにはデバッガが ECALL を傍受できるようにする必要があります。私たちは標準の ABI で ECALL を使用することに移行するつもりですが、その場合、セミホスティングは既存の標準のサービス ABI を共有することができます。*

*ARMプロセッサも、新しいデザインのセミホスト呼び出しにBKPTではなくSVCを使用するようになったことに注意してください。*

## 2.9　HINT命令

RV32I はHINT 命令用に大規模なエンコーディング スペースを確保しており、通常はマイクロアーキテ クチャに性能のヒントを伝えるために使用されます。HINT は、*RD*=X0 の整数演算命令としてエンコードされます。そのため、NOP 命令と同様に、HINT 命令は pc と該当するパフ ォーマ ン ス カ ウ ン タ ーを前進させる以外は、 アーキ テ ク チ ャ ル上で目に見える状態を変更 し ません。実装では、エンコードされたヒントを常に無視することが許されています。

*このHINTエンコーディングは、単純なインプリメンテーションがHINTを完全に無視し、代わりにアーキテクチャの状態を変異させない通常の計算命令としてHINTを実行できるように選択されています。例えば、ADDは宛先レジスタが*x0*であればHINTです。5ビットのrs*1*と*rs2*フィールドはHINTへの引数をエンコードします。5 ビットの rs1 と rs2 フィールドは HINT への引数をエンコードしますが、単純な実装では、単に* x0 を*書き込む* rs1 *と* rs2 の *ADD として HINT を実行することができ、アーキテクチャ的には目に見える効果はありません。*

表2.3にすべてのRV32I HINTコードポイントを示します。HINT空間の91%は標準HINT用に予約されていますが、現在は定義されていません。残りのHINT スペースはカスタムHINT 用に予約されており、このサブスペースでは標準HINT は定義されません。

*現在のところ標準的なヒントは定義されていません。標準的なヒントには、最終的にはメモリシステムの空間的・時間的ロカリティヒント、分岐予測ヒント、スレッドスケジューリングヒント、セキュリティタグ、シミュレーション/エミュレーションのためのインストルメンテーションフラグが含まれることを期待しています。*

|  |  |  |  |
| --- | --- | --- | --- |
| 命令 | 制約 | コードポイント | 目的 |
| LUI | rd=x0 | 220 | 将来の標準使用のために予約されています。 |
| AUIPC | rd=x0 | 220 |
| ADDI | rd=x0, and either  rs1≠x0 or imm≠0 | 217 - 1 |
| ANDI | rd=x0 | 217 |
| ORI | rd=x0 | 217 |
| XORI | rd=x0 | 217 |
| ADD | rd=x0 | 210 |
| SUB | rd=x0 | 210 |
| AND | rd=x0 | 210 |
| OR | rd=x0 | 210 |
| XOR | rd=x0 | 210 |
| SLL | rd=x0 | 210 |
| SRL | rd=x0 | 210 |
| SRA | rd=x0 | 210 |
| FENCE | pred=0 or succ=0 | 25 - 1 |
| SLTI | rd=x0 | 217 | カスタム使用のために予約されています。 |
| SLTIU | rd=x0 | 217 |
| SLLI | rd=x0 | 210 |
| SRLI | rd=x0 | 210 |
| SRAI | rd=x0 | 210 |
| SLT | rd=x0 | 210 |
| SLTU | rd=x0 | 210 |

表2.3：RV32IのHINT命令。

**第3章**

# “Zifencii"命令-フェッチ フェンス、 バージョン2.0

この章では、命令メモリへの書き込みと同じハート上での命令フェッチの間の明示的な同期を提供するFENCE.I命令を含む "Zifencei "拡張を定義します。現在のところ、この命令は、あるハートに見えるストアがその命令フェッチにも見えることを保証する唯一の標準的なメカニズムです。

*我々は “保存命令語" 命令を考慮したが，（MAJC[19]のように）含まなかった．JITコンパイラはFENCE.Iの前に大量の命令のトレースを生成し，翻訳された命令をIキャッシュに存在しないことが知られているメモリ領域に書き込むことで命令キャッシュのスヌーピング/無効化のオーバーヘッドを償却することができます．*

*FENCE.I命令は様々な実装をサポートするように設計されています。単純な実装ではFENCE.Iが実行されるとローカルの命令キャッシュと命令パイプラインがフラッシュされます。より複雑な実装では、データ(命令)キャッシュのミスに対して命令(データ)キャッシュをスヌーピングしたり、ローカルストア命令によって書き込まれた行をプライマリ命令キャッシュから無効にするために、包括的な統一されたプライベートL2キャッシュを使用したりすることができます。このように命令キャッシュとデータキャッシュがコヒーレントに保たれている場合、あるいはメモリシステムがキャッシュされていないＲＡＭのみで構成されている場合は、フェッチパイプラインだけをＦＥＮＣＥ.Iでフラッシュする必要があります。*

*FENCE.I命令は以前はベースI命令セットの一部でした。これを必須ベースから外したのは主に2つの問題がありますが、執筆時点ではまだ命令フェッチの一貫性を維持するための唯一の標準的な方法です。*

*第一に、いくつかのシステムでは、ＦＥＮＣＥ.Ｉは実装にコストがかかることが認識されており、メモリモデルタスクグループで代替メカニズムが議論されています。特に、インコヒーレントな命令キャッシュとインコヒーレントなデータキャッシュを持つデザインの場合、または命令キャッシュのリフィルがコヒーレントなデータキャッシュをスヌープしない場合、ＦＥＮＣＥ.Ｉ命令に遭遇したときに両方のキャッシュを完全にフラッシュしなければならない。この問題は、統一キャッシュまたは外部メモリシステムの前に複数のレベルのＩおよびＤキャッシュがある場合に悪化する。*

*第二に、この命令はUnixのようなOS環境でユーザレベルで利用できるようにするには十分に強力ではありません。FENCE.I はローカルハートを同期させるだけで、OS は FENCE.I の後にユーザハートを別の物理ハートにリスケジュールすることができます。これは、OSがコンテキストマイグレーションの一部として追加のFENCE.Iを実行する必要があります。このため、標準の Linux ABI は FENCE.I をユーザレベルから削除し、命令フェッチコヒーレンスを維持するためにシステムコールを必要とするようになりました。*

*議論中の命令フェッチコヒーレンスへの将来のアプローチは、rs1で指定された特定のアドレスのみをターゲットとするFENCE.Iのより制限されたバージョンを提供すること、および/または、ソフトウェアがマシンモードのキャッシュメンテナンス操作に依存するABIを使用することを可能にすることを含みます。*

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | オペコード |

１２ ５ ３ ５ ７

0 0 FENCE.I 0 雑多-MEM

FENCE.I 命令は命令とデータストリームを同期させるために使用されます。RISC-V は FENCE.I 命令を実行するまでは、命令メモリへのストアが RISC-V ハート上の命令フェッチから見えるようになることを保証しません。FENCE.I命令は、RISC-Vハ-ト上での後続の命令フェッチが、同じRISC-Vハ-トに対して既に表示されている以前のデータストアを見ることを保証します。FENCE.I命令は、マルチプロセッサシステムにおいて、他のRISC-Vハートの命令フェッチがローカルハートのストアを見ることを保証するものでは*ありません。*命令メモリへのストアを全てのRISC-Vハートから見えるようにするためには、書き込みハートは全てのリモートRISC-VハートにFENCE.Iの実行を要求する前にデータFENCEを実行しなければなりません。

FENCE.I命令の中の未使用のフィールド、*imm[11:0*]、*rs1*、*rdは*、将来の拡張でより細かいフェンスのため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はハート自身の命令フェッチでストアを命令するだけなので、アプリケーションコードはアプリケーションスレッドが別のハートにマイグレートされない場合にのみFENCE.Iに頼るべきです。EEIは効率的なマルチプロセッサ命令ストリーム同期のためのメカニズムを提供します。*

**第4章**

# RV32E基本整数命令セット、バージョン1.9

この章では、組み込みシステム向けに設計されたRV32Iの縮小版であるRV32Eベース整数命令セットのドラフト案について説明します。唯一の変更点は、整数レジスタの数を16個に減らすことです。この章では、RV32EとRV32Iの違いを概説しているだけなので、第2章の後に読んでください。

*RV32Eは、組み込みマイクロコントローラ用にさらに小さなベースコアを提供するために設計されました。この可能性については本書のバージョン2.0で言及していましたが、当初はこのサブセットを定義することに抵抗を感じていました。しかし、可能な限り小さな32ビット・マイクロコントローラに対する需要を考慮し、この分野での断片化を先取りするために、RV32EをRV32I、RV64I、RV128Iに加え、第4の標準ベースISAとして定義しました。また、高度にスレッド化された64ビットプロセッサのためにコンテキスト状態を減らすためのRV64Eの定義にも関心があります。*

## 4.1　RV32E プログラマーズモデル

RV32Eは、整数レジスタ数を16個の汎用レジスタ(x0〜x15)に削減し、ここでx0は専用のゼロレジスタです。

*小型のRV32Iコア設計では、上位16個のレジスタはメモリを除いたコアの総面積の約4分の1を消費するため、このレジスタを削除することでコア面積を約25%削減し、それに伴ってコア電力を削減できることがわかりました。*

*この変更により、異なる呼び出し規約とABIが必要になります。特に、RV32Eはソフトフロート呼び出し方式でのみ使用されます。RV32EとRV32Iで動作する新しい組み込みABIが検討されています。*

## 4.2　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ていますが、レジスタx0～x15のみが提供されています。将来の標準拡張では、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は使用されないため、これらはカスタム拡張で使用できます。

*RV32Eは、現在のすべての標準エクステンションと組み合わせることができます。F,D,Q拡張をRV32Eと組み合わせて16エントリの浮動小数点レジスタファイルを持つように定義することも検討されましたが、これは却下されました。浮動小数点レジスタの状態が減少したシステムをサポートするために、浮動小数点演算に整数レジスタを使用させ、浮動小数点のロード、ストア、浮動小数点レジスタと整数レジスタ間の移動を削除する “Zfinx"拡張機能を定義する予定です。*

**第5章**

# RV64I 基本整数命令セット、バージョン2.1

この章では、第2 章で説明したRV32I の変形版をベースにしたRV64I の基底整数命令セットについて説明します。この章では、RV32I との違いのみを説明していますので、前の章と合わせてお読みください。

## 5.1　登録状態

RV64Iは整数レジスタとサポートされているユーザーアドレス空間を64ビットに拡張します(図2.1のXLEN=64)。

## 5.2　整数演算命令

ほとんどの整数演算命令はXLENビット値で動作します。RV64Iの32ビット値を操作するための追加の命令が用意されており、オペコードのサフィックスに「W」が付いています。

これらの "\*W "命令は入力の上位32ビットを無視し、常に32ビット符号付き値を生成します。

すなわち、ビットXLEN-1～31は等しい。

*コンパイラと呼び出し規約は、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性を維持しています。32 ビット符号なし整数であっても、ビット 31 はビット 63 から 32 まで拡張されます。そのため、符号なし 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できません。既存の 64 ビット幅の SLTU および符号なし分岐比較は、この不変量の下でも符号なし 32 ビット整数で正しく動作します。同様に、32ビット符号拡張整数に対する既存の64ビット幅の論理演算は符号拡張特性を保持します。32ビット値に対する妥当な性能を確保するために、加算と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-初歩命令**

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | オペコード |

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 ビットのイミディエイトをレジスタ *rs1 に*追加し、32 ビットの結果の適切な符号拡張を *rd* に生成する RV64I 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ビットです。注意：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(アセンブラ擬似命令 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 | オペコード |

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 SLLIIW dest OP-IMM-32

000000 0 shamt[4:0] src SRLIIW dest OP-IMM-32

010000 0 shamt[4:0] src SRAIIW 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型フォーマットの特殊化として符号化されます。シフトするオペランドは*rs1*にあり、シフト量はRV64Iの場合はI-即値フィールドの下位6ビットに符号化されます。右シフトタイプはビット30にエンコードされています。SLLIは論理的な左シフト（ゼロは下位ビットにシフト）、SRLIは論理的な右シフト（ゼロは上位ビットにシフト）、SRAIは算術的な右シフト（元の符号ビットは空になった上位ビットにコピーされます）です。

SLLIW、SRLIW、およびSRAIWは、類似して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専用命令です。SLLIW、SRLIW、およびSRAIWの*imm*[5]≠0の符号化は予約されています。

*以前は、imm[5]≠0のSLLIW、SRLIW、SRAIWは不正な命令例外を発生させるように定義されていましたが、現在は予約済みとしてマークされています。これは下位互換性のある変更です。*

31 12 11 7 6 0

|  |  |  |
| --- | --- | --- |
| imm[３１:12] | rd | オペコード |

20 ５ ７

U-即値[31:12] dest LUI

U-即値[31:12] dest AUIPC

LUI(ロードアッパーイミディエイト)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31--12ビットに配置し、最下位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 に上位イミディエイトを追加) は RV32I と同じオペコードを使用します。AUIPC は pc の相対アドレスを構築するために使用され、U 型フォーマットを使用します。AUIPCは20ビットのU-即値に12の低次ゼロビットを追加し、符号はその結果を64ビットに拡張し、AUIPC命令のアドレスに追加し、その結果をレジスタrdに格納します。

**整数レジスタの操作**

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct7 | rs2 | rs1 | funct3 | rd | オペコード |

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 と SUBW は、ADD と SUB と同様に定義されていますが、32 ビットの値で動作し、符号付き 32 ビットの結果を生成する RV64I 専用命令です。オーバーフローは無視され、結果の下位32ビットは符号付きで64ビットに拡張され、宛先レジスタに書き込まれます。

SLL,SRL,SRA はレジスタ *rs1 の*値に対してレジスタ rs*2 に*保持されているシフト量で論理左シフト、論理右シフト、算術右シフトを行います。RV64Iではrs*2の*下位6ビットのみがシフト量として考慮されます。

SLLW、SRLW、SRAW は、類似して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 トの結果を生成するRV64I 専用命令です。シフ ト 量は*rs2[4:0]* で与えられます。

## 5.3　ロードと保存の命令

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合法かを定義します。

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | オペコード |

１２ ５ ３ ５ ７

オフセット[11:0] base 幅 dest ロード

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | オペコード |

7 5 ５ ３ ５ ７

オフセット[11:5] ｓｒｃ base 幅 オフセット[4:0] ストア

LD命令はメモリからRV64I用のレジスタ*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ビットの値をロードし、これを64ビットに符号拡張してからRV64I用のレジスタ*RD*に格納します。一方、LWU命令はメモリから32ビットの値をゼロ拡張してRV64I用のレジスタRDに格納します。LHとLHUは16ビットの値に対してアナログ的に定義され、LBとLBUは8ビットの値に対してアナログ的に定義されています。SD命令、SW命令、SH命令、SB命令はそれぞれレジスタ*rs2*の下位ビットから64ビット、32ビット、16ビット、8ビットの値をメモリに格納します。

## 5.4　HINT命令

RV32I(セクション2.9参照)のマイクロアーキテクトHINTであるすべての命令は、RV64IのHINTでもあります。RV64Iの追加の計算命令は、標準とカスタムのHINTエンコーディング空間の両方を拡張します。

表5.1にすべてのRV64I HINTコードポイントを示します。HINT空間の91%は標準HINT用に予約されていますが、現在は何も定義されていません。残りのHINT空間はカスタムHINT用に予約されており、このサブスペースでは標準HINTは定義されません。

|  |  |  |  |
| --- | --- | --- | --- |
| 命令 | 制約 | コードポイント | 目的 |
| LUI | *ｒｄ*＝ｘ０ | 220 | *将来の標準使用のために予約されています。* |
| AUIPC | *ｒｄ*＝ｘ０ | 220 |
| ADDI | *rd=*x0*, とどちらかの rs≠*x0*かimm≠*0 | 217  - 1 |
| ANDI | *ｒｄ*＝ｘ０ | 217 |
| ORI | *ｒｄ*＝ｘ０ | 217 |
| ＸＯＲＩ | *ｒｄ*＝ｘ０ | 217 |
| ＡＤＤＩＷ | *ｒｄ*＝ｘ０ | 217 |
| ADD | *ｒｄ*＝ｘ０ | 210 |
| SUB | *ｒｄ*＝ｘ０ | 210 |
| AND | *ｒｄ*＝ｘ０ | 210 |
| OR | *ｒｄ*＝ｘ０ | 210 |
| ＸＯＲ | *ｒｄ*＝ｘ０ | 210 |
| SLL | *ｒｄ*＝ｘ０ | 210 |
| SRL | *ｒｄ*＝ｘ０ | 210 |
| SRA | *ｒｄ*＝ｘ０ | 210 |
| ADDW | *ｒｄ*＝ｘ０ | 210 |
| SUBW | *ｒｄ*＝ｘ０ | 210 |
| ＳＬＬＷ | *ｒｄ*＝ｘ０ | 210 |
| SRLW | *ｒｄ*＝ｘ０ | 210 |
| ＳＲＡＷ | *ｒｄ*＝ｘ０ | 210 |
| FENCE | *pred=*0*またはsucc=*0 | 25  - 1 |
| SLTI | *ｒｄ*＝ｘ０ | 217 | *カスタム使用のために予約されています* |
| SLTIU | *ｒｄ*＝ｘ０ | 217 |
| SLLI | *ｒｄ*＝ｘ０ | 211 |
| SRLI | *ｒｄ*＝ｘ０ | 211 |
| SRAI | *ｒｄ*＝ｘ０ | 211 |
| SLLIW | *ｒｄ*＝ｘ０ | 210 |
| SRLIW | *ｒｄ*＝ｘ０ | 210 |
| SRAIW | *ｒｄ*＝ｘ０ | 210 |
| ＳＬＴ | *ｒｄ*＝ｘ０ | 210 |
| ＳＬＴＵ | *ｒｄ*＝ｘ０ | 210 |

表 5.1.RV64IのHINT命令。

**第6章**

# RV128I基本整数命令セット、バージョン1.7

*"コンピュータ設計において、回復するのが困難なミスが一つだけあります。それは、メモリアドレッシングとメモリ管理のための十分なアドレスビットを持っていないことです。"* ベルとストレッカー、ISCA-3、1976年。

この章では、フラット128 ビッ トのアドレス空間をサポートするRISC-V ISAのバリアントであるRV128Iについて説明します。このバリアントは、既存のRV32IとRV64Iのデザインを簡単に外挿したものです。

*整数レジスタの幅を広げる第一の理由は、より大きなアドレス空間をサポートするためです。64ビット以上のフラットなアドレス空間がいつ必要になるかは不明です。本稿執筆時点では、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、50 ビット以上のアドレス空間を必要とすることになります。倉庫規模のコンピュータの中には、すでにそれ以上の量のDRAMを搭載しているものもあり、新しい高密度ソリッドステート不揮発性メモリと高速相互接続技術により、さらに大きなメモリ空間の需要が高まる可能性があります。エクサスケール・システムの研究では、57 ビットのアドレス空間を占める 100PB メモリ・システムを目標としています。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。*

*歴史的に見ても、64ビット以上のアドレス空間が必要であることが明らかになると、アーキテクトは、セグメンテーション、96ビットのアドレス空間、ソフトウェアの回避策など、アドレス空間を拡張するための代替案について集中的な議論を繰り返し、最終的には128ビットのフラットなアドレス空間が最も単純で最良の解決策として採用されることになります。*

*実際の128ビットアドレス空間の使用状況に応じて設計を進化させる必要があるかもしれないため、現時点ではRV128の仕様は凍結していません。*

RV128Iは、RV32Iと同様にRV64Iをベースに、整数レジスタを128ビットに拡張したRV64Iをベースにしています（すなわち、XLEN=128）。ほとんどの整数演算命令は、XLENビットで動作するように定義されているため、変更はありません。レジスタの下位ビットの32ビット値を操作するRV64Iの“\*W”整数命令は保持されていますが、符号は31ビットから127ビットまで結果を拡張しています。128ビット整数レジスタの下位ビットに保持された64ビットの値を操作する“\*D"整数命令の新しいセットが追加され、その結果が63ビットから127ビットに拡張されました。”\*D"命令は標準32ビットエンコーディングの2つの主要なオペコード(OP-IMM-64とOP-64)を消費します。

*RV64との互換性を向上させるために、RV32からRV64への処理を逆にして、RV64I ADDを64ビットADDDに改名し、以前はOP-64メジャー・オペコードだったものに128ビットADDQを追加した（現在はOP-128メジャー・オペコードに改名されている）デコードを変更するかもしれません。*

即値（SLLI/SRLI/SRAI）によるシフトは、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ビットを使用するようになりました。

既存の LOAD メジャー・オペコードを使用して LDU (ロードダブル符号なし) 命令を追加し、クアッドワード値をロード/ストアするための LQ、SQ 命令を追加しました。メジャーコードのSTOREにはSQ、MISC-MEMにはLQが追加されています。

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、T（128ビット）整数フォーマットとの間の追加のFCVT命令とともに、FMV.X.Q.QおよびFMV.Q.X命令をサポートできるようになりました。

**第7章**

# “M"整数乗除算のための標準拡張機能、 バージョン2.0

この章では標準的な整数の乗除算命令の拡張子である ``M""と呼ばれる命令について説明します。

*ローエンドの実装を簡素化するために、整数の乗算と除算をベースから分離しています。また、整数の乗算と除算の演算が頻繁に行われないか、または付属のアクセラレータで処理する方が良いアプリケーションのために、整数の乗算と除算を分離しています。*

## 7.1　乗算演算

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct7 | rs2 | rs1 | funct3 | rd | オペコード |

7 5 ５ ３ ５ ７

MULDIV 乗数 乗算 MUL/MULH[[S]U] dest OP

MULDIV 乗数 乗算 MULW dest OP-32

MUL は rs1 を rs2 で XLENビット × XLEN ビットの乗算を行い、下位の XLEN ビットをデスティネーション レジスタに配置します。MULH、MULHU、および MULHSU は、同じ乗算を実行しますが、それぞれ 符号付 × 符号付、符号無 × 符号無、および 符号付 rs1 × 符号無 rs2 乗算の場合は、完全な 2×XLEN ビット積の上位 XLEN ビットを返します。同じプロダクトの高ビットおよび低ビット両方が要求されれば、推薦されたコード・シーケンスは次のとおりです。MULH[[S]U] *rdh, rs1, rs2*; MUL *rdl.rs1, rs2* ( ソース 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トでは、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させることができます。

*MULHSU は、マルチワード符号付き乗算で使用され、乗算器の最上位ワード（符号ビットを含む）に、乗算器の最下位ワード（符号なし）を乗算するために使用されます。*

MULWは、ソース・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デスティネーション・レジスタに配置するRV64命令です。

*RV64では、64ビット積の上位32ビットを取得するためにMULを使用す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らず、符号なし引数はその上位32ビットをクリアしなければなりません。引数が符号付きまたはゼロ拡張であることがわからない場合、代替手段として、両方の引数を左に32ビットシフトしてからMULH[[S]U]を使用することができます。*

## 7.2　分割操作

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct7 | rs2 | rs1 | funct3 | rd | オペコード |

7 5 ５ ３ ５ ７

MULDIV 除数 被除数 DIV[U]/REM[U] dest OP

MULDIV 除数 被除数 DIV[U]W/REM[U]W dest OP-32

DIV と DIVU は、rs*1* と *rs2 の XLEN ビット*×XLEN ビットの符号付きおよび符号なし整数除算を実行し、ゼロに丸めます。REMとREMUは、対応する除算演算の残りを提供します。REMの場合、結果の符号は配当の符号に等しい。

*符号付き除算と符号なし除算の両方について、被除数*＝除数 × *商 ＋ 剰余 を保持して*い*ます。*

商と剰余の両方が同じ除算から必要な場合、推奨されるコードシーケンスは以下の通りです。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ーでは、2つの別々の分割を実行する代わりに、これらを1つの分割演算に融合させることができます。

DIVW、DIVUWはRV64命令で、rs*1の*下位32ビットをrs*2の*下位32ビットで除算し、それぞれ符号付き整数、符号なし整数として扱い、32ビット商を64ビットに符号拡張して*rd*に格納します。REMWとREMUWは、それぞれ対応する符号付きと符号なしの剰余演算を提供するRV64命令です。REMWとREMUWの両方とも、ゼロによる除算を含め、常に32ビットの結果を64ビットに符号拡張します。

ゼロによる除算 と 除算オーバーフローのセマンティクスを表 7.1 にまとめます。ゼロによる除算の商はすべてのビットがセットされており、ゼロによる除算の残りは配当に等しくなります。符号付き除算のオーバフローは、 最多負の整数を-1 で除算した場合にのみ発生します。 オーバフローのある符号付き除算の商は配当に等しく、残りはゼロです。符号なし除算のオーバーフローは発生しません。

*私たちは、ゼロで割った整数に対して例外を発生させることを検討しましたが、これらの例外はほとんどの実行環境でトラップの原因となります。しかし、これは標準ISAの中で唯一の算術トラップになります（浮動小数点例外はフラグを設定してデフォルト値を書き込みますが、トラップは発生しません）、言語実装者はこの場合のために実行環境のトラップハンドラと対話する必要があります。*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 条件 | 被除数 | 除数 | DIVU[W] | REMU[W] | DIV[W] | REM[W] |
| ゼロによる除算 | *x* | 0 | 2*L*  - 1 | *x* | - 1 | *x* |
| オーバーフロー（符号付きのみ） | - 2*L* - 1 | - 1 | -- | -- | - 2*L* - 1 | 0 |

表 7.1:ゼロによる除算と除算オーバーフローの意味。L は演算の幅をビット単位で表します。DIV[U]とREM[U]の場合はXLEN、DIV[U]WとREM[U]Wの場合は32です。

*さらに、言語標準がゼロ分割例外が直ちに制御フローの変更を引き起こすことを義務付けている場合、各分割操作に1つの分岐命令を追加するだけでよく、この分岐命令は分割の後に挿入することができ、通常は非常に予測的に取られないはずで、実行時のオーバーヘッドをほとんど追加しません。*

*設定されたすべてのビットの値は、除算器回路を単純化するためにゼロで符号なし除算と符号付き除算の両方のために返されます。すべての1 の値は、最大の符号なし数を表す符号なし除算のために返す自然な値であり、単純な符号なし除算器の実装のための自然な結果でもあります。符号付き除算は符号なし除算回路を使っ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ことでハードウェアが単純化されます。*

**第8章**

# ”A"アトミック命令のための標準拡張、 バージョン2.1

“A"と名付けられた標準のアトミック命令拡張機能には、同じメモリ空間で動作する複数の RISC-V ハーツ間の同期をサポートするために、メモリをアトミックに読み出し-修正-書き戻しする命令が含まれています。提供される2種類のアトミック命令には、ロードリザーブ/ストアコンディショナル命令とアトミックフェッチアンドオペレーショ ン命令があります。どちらのタイプのアトミック命令も、無秩序、取得、解放、順次一貫性のあるセマンティクスを含む様々なメモリ一貫性の順序をサポートしています。これらの命令により、RISC-VはRCscメモリ一貫性モデルをサポートすることができます[5]。

*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最終的にリリース一貫性を標準メモリ一貫性モデルとすることに落ち着いたようで、RISC-Vの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。*

## 8.1　原子命令の順序指定

ベースのRISC-V ISAは緩和されたメモリモデルを持ち、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ドメインとI/Oドメインに分割されており、FENCE命令はこれら2つのアドレスドメインのうちの1つまたは両方へのアクセスを順序付けるオプションを提供します。

リリース一貫性[5]をより効率的にサポートするために、各アトミック命令は、他のRISC-Vハートから見た追加のメモリ順序制約を指定するために使用される2つのビット、*aq*と*rlを*持っています。これらのビットは、アトミック命令がアクセスしているアドレスドメインに応じて、メモリまたはI/Oの2つのアドレスドメインのうちの1つへのアクセスを順序付けます。他方のドメインへのアクセスには順序付け制約はありませんので、両方のドメインに渡って順序付けを行うには FENCE 命令を使用しなければなりません。

両方のビットがクリアの場合、アトミックメモリ操作には追加の順序制約は課されません。aq ビットのみがセットされている場合、アトミック・メモリ操作は取得アクセスとして扱われます。

すなわち、このRISC-Vハート上では、取得メモリ動作の前に次のメモリ動作が行われていないことが観察できます。

rl 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リリースアクセスとして扱われます。

つまり、この RISC-V ハート 上のどのメモリ操作よりも前にリリースメモリ操作が行われることは観察できません。

aqとrlの両方のビットがセットされている場合、アトミックメモリ操作は順次一貫しており、同じRISC-Vハート内で同じアドレスドメインに対して、以前のメモリ操作の前に、または後のメモリ操作の後に発生することは観察できません。

## 8.2　ロード予約/保存条件付き命令

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | aq | r1 | rs2 | rs1 | funct3 | rd | オペコード |

5 1 1 ５ 5 ３ 5 7

LR.W/D 順序 0 アドレス 幅 dest AMO

SC.W/D 順序 src アドレス 幅 dest AMO

単一のメモリ・ワードまたはダブルワードに対する複雑なアトミック・メモリ操作は、ロード・リザーブド(LR)命令とストア・コンディショナル(SC)命令を用いて実行される。LR.W は *rs1* のアドレスからワードをロードし、符号拡張値を *rd* に格納し、アドレス指定されたワードのバイトを含むバイトのセットである*予約セット*を登録します。SC.W は条件付きで *rs2* のワードを rs*1 の*アドレスに書き込みます: SC.W は予約がまだ有効で、予約セットに書き込み中のバイトが含まれている場合にのみ成功します。SC.W が成功した場合、命令は *rs2 のワードを*メモリに書き込み、*rd* にゼロを書き込みます。SC.W が失敗した場合は、メモリへの書き込みは行われず、0 以外の値を *rd に書き込みます*。成功・失敗に関わらず、SC.W命令を実行すると、このハートが保持していた予約が無効になります。LR.DとSC.Dはダブルワードに対して類似した動作をし、RV64でのみ利用可能である。RV64では、LR.WとSC.Wは*rd*に格納された値を拡張する。

*比較とスワップ(CAS)とLR/SCの両方を使用して、ロックフリーのデータ構造を構築することができる。議論を重ねた結果、いくつかの理由からLR/SCを選択した。*

*1) CASはABA問題に悩まされるが、LR/SCはデータ値の変化のみ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、これを回避できる。*

*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)をサポートするための新しい整数命令フォーマットと、異なるメモリシステムメッセージフォーマットを必要とし、マイクロアーキテクチャを複雑にする。*

*3) さらに、ABA問題を回避するために、他のシステムでは、データワードと一緒にカウンタをテストしてインクリメントすることを可能にするダブルワイドCAS(DW-CAS)を提供しています。これは、1回の命令で5つのレジスタを読み込んで2つのレジスタを書き込む必要があり、さらに新しいより大きなメモリシステムのメッセージタイプが必要となり、実装をさらに複雑にしている。*

*4) LR/SCは、CASでは2回のロードが必要なのに対し、1回のロードで済むため、多くのプリミティブの実装がより効率的になる（CAS命令の前に1回ロードして投機的計算のための値を取得し、CAS命令の一部として2回目のロードを行い、更新前に値が変更されていないかどうかをチェックする）。*

*CASに対するLR/SCの主な欠点はlivelockであるが、これは特定の状況下では、後述するように最終的な前進を保証するアーキテクチャで回避している。もう一つの懸念は、DW-CASを持つ現在のx86アーキテクチャの影響で、DW-CASを基本マシンプリミティブとする同期ライブラリや他のソフトウェアの移植が複雑になるのではないかということである。緩和要因として考えられるのは、最近x86にトランザクションメモリ命令が追加されたことで、DW-CASからの移行が起こる可能性がある。*

*より一般的には、複数単語のアトミックプリミティブが望ましいが、これがどのような形をとるべきかについてはまだかなりの議論があり、また、前進を保証することはシステムに複雑さを付加する。*

*私私たちの現在の考えは、オリジナルのトランザクションメモリの提案に沿って、オプションの標準拡張 "T "として、小さな限られた容量のトランザクションメモリバッファを含めることです。*

値 1 の故障コードは、指定されていない故障を表すために予約されています。他の故障コードはこの時点で予約されており、ポータブルソフトウェアは故障コードが0以外のものであると仮定しなければなりません。

*SLT/SLTU 命令に必要な既存の mux を使用して、単純な実装でこの値を返すことができるように、“不特定''を意味する故障コードを 1 としておきます。より具体的な失敗コードは、将来のバージョンやISAの拡張で定義されるかもしれません。*

LRとSCの場合、A拡張では、*rs1*に保持されているアドレスがオペランドのサイズに自然整列している必要がある(64ビットワードの場合は8バイト、32ビットワードの場合は4バイト)。アドレスが自然アラインメントされていない場合は、アドレス・ミスアラインド例外またはアクセス・フォルト例外が生成されます。アクセス・フォールト例外は、不整合アクセスをエミュレートしてはならない場合、不整合を除いて完了することができるメモリ・アクセスに対して生成することができます。

*ずれたLR/SCシーケンスをエミュレートすることは、ほとんどのシステムでは非現実的です。*

*LR/SCシーケンスの位置がずれていると、一度に複数の予約セットにアクセスする可能性も出てくるが、これは現在の定義では規定されていません。*

実装では、予約セットにアドレス指定されたデータワードまたはダブルワードの全バイトが含まれていれば、各LRに任意の大きさの予約セットを登録することができる。SCはプログラム順に最新のLRとしかペアリングできない。SC は、LR と SC の間で他のハートから予約セットへの格納が発生していないことが確認でき、かつ LR と SC の間に他の SC がプログラム順に存在しない場合にのみ成功する可能性がある。LR命令がアクセスしたバイトに対するハート以外のデバイスからの書き込みが、LRとSCの間で発生していないことが確認できた場合にのみ、SCが成功する可能性がある。このLRは有効アドレスとデータサイズが異なるが、予約セットの一部としてSCのアドレスを予約していた可能性があることに注意してください。

*このモデルに従うと、メモリ変換のあるシステムでは、先のLRが異なる仮想アドレスのエイリアスを使用して同じ場所を予約した場合、SCは成功するが、仮想アドレスが異なる場合は失敗することも許されます。*

*レガシーデバイスやバスに対応するために、RISC-Vハーツ以外のデバイスからの書き込みは、LRがアクセスしたバイトと重なった場合にのみ予約を無効にする必要があります。これらの書き込みは、予約セット内の他のバイトにアクセスしたときに予約を無効にする必要はありません。*

SC は、プログラム順に最新の LR の予約セットにアドレスが含まれていない場合、失敗すします。LRとSCの間で、他のハートからの予約セットへの格納が観測された場合、SCは失敗します。他のデバイスから LR がアクセスしたバイトに対する書き込みが LR と SC の間で発生する可能性がある場合、SC は失敗します。(そのようなデバイスが予約セットを書き込んでも LR がアクセスするバイトを書き込まない場合、SC は失敗する場合もあればしない場合もある)。プログラム順にLRとSCの間に別のSC（任意のアドレスへの）が存在する場合、SCは失敗しなければなりません。LR/SCシーケンスが成功するための原子性要件の正確な記述は、第14.1節の原子性公理で定義されています。

*プラットフォームは、予約セットの大きさと形状を決定するための手段を提供する必要があります。*

*プラットフォームの仕様は、予約セットのサイズと形状を制約することができます。例えば、Unixプラットフォームは、予約セットが固定サイズで、連続していて、自然に整列していて、仮想メモリのページサイズより大きくないことをメインメモリに要求することが期待されています。*

*メモリのスクラッチワードへの保存条件付き命令は、既存のロード予約を強制的に無効にするために使用する必要があります。*

* *プリエンプティブコンテキストスイッチの間に*
* *アクティブな予約を含む可能性のあるページを移行する場合など、仮想アドレスから物理アドレスへのマッピングを変更する場合に必要であれば*

*ハートがLRやSCを実行するときにハートの予約が無効になることは、ハートが一度に1つの予約しか保持できず、SCはプログラム順に、最新のLRとしかペアリングできず、LRは次のSCとしかペアリングできないことを暗示しています。これは、このように動作する予想される共通実装上でソフトウェアが正しく動作することを保証する、セクション14.1のアトミック性公理への制限です。*

予約を確立した LR 命令の前に SC 命令が他の RISC-V ハートから観測されることはありません。LR/SC シーケンスは、LR 命令に *aq ビット*を設定することで取得セマンティクスを付与することができます。LR/SC シーケンスは、SC 命令に *rl ビットを設定すること*で解放セマンティクスを付与することができます。LR 命令に *aq ビットを設定*し、SC 命令に *aq ビット*と *rl* ビットの両方を設定することで、LR/SC シーケンスは連続的に一貫したものとなります。

LRとSCの両方にどちらのビットもセットされていない場合、LR/SCシーケンスは、同じRISC-Vハートからの周囲のメモリ操作の前または後に発生することが観察できます。これは、LR/SCシーケンスが並列削減動作を実装するために使用される場合に適切です。

ソフトウェアは、*aqビットが設定されていない*限りLR命令に*rl*ビットを設定してはならず、また、ソフトウェアは*rl*ビットが設定されていない限りSC命令に*aq*ビットを設定してはなりません。LR. *rl* および SC. *aq* 命令は、両方のビットをクリアした場合よりも強い順序を提供することは保証されていないが、性能が低下する可能性があります。

# a0 はメモリロケーションのアドレスを保持しています

# a1は期待値を保持しています

# a2が所望の値を保持しています

# a0 は戻り値を保持し、成功した場合は 0、そうでない場合は !0 を保持します。

cas:

lr.w t0, (a0) # 元の値を読み込みます。

bne t0, a1, fail # 一致しないので失敗。

sc.w t0, a2, (a0) # 更新してみてください。

bnez t0, cas # 保存条件に失敗した場合は再試行します。

li a0, 0 # 成功へのリターンを設定します。

jr ra # 復帰します。

fail:

li a0, 1 # 失敗した場合のリターンを設定します。

jr ra # 復帰します。

図8.1.LR/SCを用いた比較と交換機能のサンプルコード。

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る。LR/SC を用いた比較・スワップ機能の実装例を図 8.1 に示します。インライン化した場合、比較・スワップ機能は4つの命令で済みます。

## 8.3　ストア条件付き命令の最終的な成功

標準A拡張では、*制約付きLR/SCループを*定義しており、以下の特性を持ちます。

* ループはLR/SCシーケンスと、失敗した場合にシーケンスを再試行するためのコードのみで構成され、メモリに連続して配置された最大16個の命令で構成されなければならなりません。
* LR/SC シーケンスは LR 命令で始まり SC 命令で終わる。LR命令とSC命令の間で実行される動的コードは基本的な”I"命令セットの命令のみを含むことができ、ロード、ストア、後方ジャンプ、後方分岐、JALR、FENCE、FENCE.I、SYSTEM命令を除きます。”C"拡張がサポートされている場合、前述のI命令の圧縮形式も許可されています。
* 失敗したLR/SCシーケンスを再試行するコードは、LR/SCシーケンスを繰り返すための後方ジャンプおよび/または分岐を含むことができるが、それ以外の場合はLRとSCの間のコードと同じ制約を持ちます。
* LRとSCのアドレスは、LR*/SCの偶数*特性を持つメモリ領域内に存在しなければならない。実行環境は、どの領域がこの特性を持つかを通信する責任があります。
* SCは、同じハートで実行される最新のLRと同じ有効アドレス、同じデータサイズでなければなりません。

制約付きLR/SCループ内にないLR/SCシーケンスは制約無し*です*。制約のないLR/SCシーケンスは、いくつかの実装では成功するかもしれませんが、他の実装では成功しないかもしれません。

*LR/SCループの長さは、ベースの64バイト以内に収まるように制限しました。*

*命令キャッシュとTLBのサイズと連想性に対する過度の制限を避けるために、ISAを使用しました。同様に、プライベートキャッシュ内の予約を追跡する単純な実装において、データキャッシュの連想性に対する制限を避けるために、ループ内の他のロードおよびストアを禁止しました。分岐とジャンプの制限は、シーケンスに費やすことができる時間を制限します。浮動小数点演算と整数の乗算／除算は、適切なハードウェアサポートがない実装での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簡素化するために禁止されました。*

*ソフトウェアは，制約のないLR/SCシーケンスを使用することを禁止されていませんが，可搬型ソフトウェアは，シーケンスが繰り返し失敗する場合を検出し， 制約のないLR/SCシーケンスに依存しない代替のコードシーケンスにフォールバックしなければなりません。実装は，どのような制約のない LR/SC シーケンスであっても無条件に失敗することが許されています。*

ハート*Hが*制約付きLR/SCループに入った場合、実行環境は、以下のイベントのいずれかが最終的に発生することを保証しなければなりません。

* *H*または他のハートは、*H*の拘束されたLR/SCループ内のLR命令の予約セットにSCを成功させます。
* *Hの*制約されたLR/SCループ内のLR命令の予約セットに対して、他の何かのハルトが無条件にストア命令やAMO命令を実行したり、システム内の他の何かのデバイスがその予約セットに書き込みを行ったりします。
* *Hは制約の*あるLR/SCループを抜ける分岐やジャンプを実行します。
* *H*トラップ。

*これらの定義は、前述の保証に違反していない場合には、何らかの理由で SC 命令に失敗することがある実装を許可していることに注意してください。*

*偶発性保証の結果として、ある実行環境の一部のハートが制約付きLR/SCループを実行していて、実行環境の他のハートやデバイスがその予約セットに対して無条件でストアやAMOを実行しない場合、少なくとも1つのハートは最終的に制約付きLR/SCループを抜けることになります。対照的に、他のハートやデバイスがその予約セットに書き込みを続けている場合、どのハートもLR/SCループを抜けることは保証されません。*

*ロードやロード予約命令は、それ自体が他のハーツのLR/SCシーケンスの進行を妨げるものではありません。この制約は、他のハーツ(同じコア内の場合もある)が実行したロード命令やロード予約命令がLR/SCの進行を無期限に妨げることができないことを意味していることに注意してください。例えば、キャッシュを共有している他のハーツがキャッシュを退避させても、LR/SCの進行を無期限に妨げることはできません。一般的に、これは予約が共有キャッシュからの退避とは無関係に追跡されることを意味します。同様に、ハート内での投機的な実行によるキャッシュミスは、LR/SCの進行を無期限に妨げることはできません。*

*これらの定義は、最終的に進歩があることを条件に、実装上の理由からSC命令が不自然に失敗する可能性があることを認めています。*

*CASの利点の一つは、システムによってはLR/SC原子配列が無期限にライブロックする可能性があるのに対し、ある種のハートが最終的に進歩することを保証することです。この懸念を回避するために、我々は特定のLR/SC原子配列に対してライブロックの自由度を保証するアーキテクチャ上の保証を追加しました。*

*この仕様の初期のバージョンでは、より強力な枯渇自由度保証が課されていました。しかし、弱いlivelockの自由度保証は、C11およびC++11言語を実装するのに十分であり、いくつかのマイクロアーキテクチャースタイルで提供するのが実質的に容易です。*

## 8.4　原子メモリの操作

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | aq | r1 | rs2 | rs1 | funct3 | rd | オペコード |

5 1 1 ５ 5 ３ 5 7

AMOSWAP.W/D 順序 0 アドレス 幅 dest AMO

AMOADD.W/D 順序 0 アドレス 幅 dest AMO

AMOAND.W/D 順序 0 アドレス 幅 dest AMO

AMOOR.W/D 順序 0 アドレス 幅 dest AMO

AMOXOR.W/D 順序 0 アドレス 幅 dest AMO

AMOMAX[U].W/D 順序 0 アドレス 幅 dest AMO

AMOMIN[U].W/D 順序 0 アドレス 幅 dest AMO

アトミック・メモリ・オペレーション(AMO)命令は、マルチプロセッサ同期化のためのリード・モディファイ・ライト操作を行う命令で、R型の命令フォーマットでエンコードされています。これらのAMO命令は、*rs1*のアドレスからデータ値をアトミックにロードし、その値をレジスタ*rd*に格納し、ロードされた値とrs*2*の元の値にバイナリ演算子を適用し、その結果をrs*1*のアドレスに格納します。AMO は、メモリ内の64 ビッ ト(RV64 のみ) と32 ビッ トのワードで動作します。RV64 の場合、32 ビット AMO は常に *rd* に格納された値を符号拡張します。

AMOの場合、A拡張子は、*rs1*に保持されているアドレスがオペランドのサイズに自然に整列している必要があります(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)。アドレスが自然アラインメントされていない場合は、アドレス・ミスアラインド例外またはアクセス・フォルト例外が生成されます。アクセスフォールト例外は、アドレスがずれたアクセスをエミュレートしてはいけない場合に、 アドレスがずれていなければメモリアクセスを完了させることができた場合に生成されます。22章で説明されている”Zam"拡張はこの要件を緩和し、不整合AMOのセマンティクスを規定しています。

サポートされている演算は、スワップ、整数加算、ビット単位の AND、ビット単位の OR、ビット単位の XOR、符号付きおよび符号なしの整数の最大値と最小値です。順序制約がなければ、これらのAMOを使用して並列削減演算を実装することができ、通常はx0に書き込むことで戻り値が破棄されます。

*我々は、LR/SCやCASよりも優れた高並列システムへの拡張性を持つ、取り出し操作スタイルのアトミックプリミティブを提供しています。単純なマイクロアーキテクチャであれば、AMOが最終的に完了することを保証できる実装であれば、LR/SCプリミティブを用いてAMOを実装することができます。より複雑な実装では、メモリコントローラでAMOを実装することも可能であり、目的地が*x0の*ときに元の値をフェッチしないように最適化することができます。*

*AMO のセットは、C11/C++11 のアトミックメモリ操作を効率的にサポートし、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。AMOのもう一つの用途は、I/O空間のメモリマップされたデバイスレジスタ（例えば、ビットの設定、クリア、トグルなど）への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る*場合、この RISC-V ハートでは AMO の前に後のメモリ操作が観測されることはありません。逆に、*rl ビットがセットされている*場合、他の RISC-V ハートは、この RISC-V ハート内の AMO に先行するメモリアクセスの前に AMO を観測しません。AMO に *aq* と *rl の*両方のビットを設定すると、シーケンスは連続的に一貫したものとなり、同じハートからのメモリアクセスでは、それ以前のメモリアクセスやそれ以降のメモリアクセスでは順序を変えることができません。

*AMO は、C11 およびC++11 のメモリ モデルを効率的にインプリメントできるように設計されています。FENCE R, RW 命令は*取得*動作を実装するのに十分であり、FENCE RW, W 命令は*解放動作*を実装するのに十分であるが、どちらも対応する* aq *または* rl *ビットセットを持つ AMO と比較して、不要な順序付けが追加されることを意味しています。*

図 8.2 に、テストとテストとセット スピンロックで保護されたクリティカル セクションのコード・シーケンスの例を示します。最初の AMO には *aq と*マークされており、クリティカル セクションの前にロックの取得を指示し、2 番目の AMO には *rl と*マークされており、ロックの放棄の前にクリティカル セクションを指示していることに注意してください。

*投機的ロック・エリシオンの実装を単純化するために、ロックの取得と解放の両方に上記のAMO Swapイディオムを使用することを推奨します[16]。*

“A"拡張子の命令は、逐次的に一貫性のあるロードとストアを提供するためにも使用できる。逐次的に一貫性のあるロードは、*aq*と*rl*の両方を設定したLRとして実装することができる。逐次的に一貫性のあるストアは、古い値を x0 に書き込む AMOSWAP として実装することができ、 *aq* と *rl の両方が*設定されています。

li t0, 1 # スワップ値を初期化します。

again:

lw t1, (a0) # ロックが保持されているかどうかを確認します。

bnex t1, again # 保持されている場合は再試行してください。

amoswap.w.aq t1, t0, (a0) # ロック取得を試みます。

bnez t1, again # 保持されている場合は再試行してください。

# ．．．

#

# ．．．

amoswap.w.r1 x0, x0, (a0) # 0 を格納してロックを解除します。

図 8.2: 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含まれています。

**第9章**

# “Zicsr", 制御およびステータスレジスタ(CSR)命令, バージョン2.0

RISC-Vでは、各ハートに関連付けられた4096個の制御レジスタとステータスレジスタのアドレス空間が定義されています。この章では、これらのレジスタで動作するCSR命令の全セットを定義します。

*CSR は主に特権アーキテクチャで使用されていますが、非特権コードでは、カウンタやタイマ、浮動小数点ステータスなど、いくつかの用途があります。*

*カウンタとタイマは、標準のベース ISA の必須部分ではなくなったため、それらにアクセスするために必要な CSR 命令は、ベース ISA の章からこの別の章に移動されました。*

## 9.1　CSR命令

すべてのCSR命令は、原子的に単一のCSRを読み書きします。そのCSR指定子は、ビット31～20で保持される命令の12ビットの*csr*フィールドにエンコードされています。即時形は、*rs1*フィールドにエンコードされた5ビットのゼロ拡張即時形を使用します。

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| csr | rs1 | funct3 | rd | オペコード |

１２ ５ ３ ５ ７

source/dest source CSRRW dest システム

source/dest source CSRRS dest システム

source/dest source CSRRC dest システム

source/dest uimm[4:0] CSRRWI dest システム

source/dest uimm[4:0] CSRRSI dest システム

source/dest uimm[4:0] CSRRCI dest システム

CSRRW (アトミック リード/ライト CSR) 命令は、CSR と整数レジスタの値を原子的に入れ替えます。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てから、整数レジスタ*rd*に書き込みます。*rs1*の初期値がCSRに書き込まれます。

55

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | レジスタオペランド | | |  |
| 命令 | ｒｄ | rs1 | CSRを読む？ | CSRを書く？ |
| ＣＳＲＲＲＷ | x0 | - | いいえ | はい |
| ＣＳＲＲＲＷ | !x0 | - | はい | はい |
| CSRRS/C | - | x0 | はい | いいえ |
| CSRRS/C | - | !x0 | はい | はい |
|  | 即時オペランド | | |  |
| 命令 | rd | uimm | CSRを読む？ | CSRを書く？ |
| ＣＳＲＲＲＷＩ | x0 | - | いいえ | はい |
| ＣＳＲＲＲＷＩ | !x0 | - | はい | はい |
| CSRRS/CI | - | 0 | はい | いいえ |
| CSRRS/CI | - | !0 | はい | はい |

表9.1：CSR命令が与えられたCSRを読み込むか書き込むかを示す表。CSRRS命令とCSRRC命令は同じ動作をするので、表ではCSRRS/Cとして示しています。

この命令は、CSRを読み込んではならず、CSRを読み込んだときに発生する可能性のある副作用を引き起こしてはなりません。

CSRRS(CSRにおけるアトミックリードおよびセットビット)命令は、CSRの値を読み取り、その値をXLENビットにゼロ拡張し、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。r*s1* のハイビットは、CSR ビットが書き込み可能であれば、対応するビットが CSR にセットされます。CSR内の他のビットは影響を受けません（ただし、CSRは書き込まれたときに副作用があるかもしれません）。

CSRRC(CSRにおけるアトミックリードおよびクリアビット)命令は、CSRの値を読み取り、その値をXLENビットにゼロエクステンションして、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。r*s1* のハイビットは、CSR ビットが書き込み可能であれば、対応するビットが CSR でクリアされます。CSR の他のビットは影響を受けません。

CSRRS と CSRRC の両方で、*rs1*=x0 の場合、命令は CSR への書き込みを一切行わないため、読み取り専用の CSR へのアクセスで不正な命令例外を発生させるなど、CSR への書き込みで発生する可能性のある副作用は発生しません。CSRRSとCSRRCの両方とも、常にアドレス指定されたCSRを読み取り、rs*1*と*rd*フィールドに関係なく読み取りの副作用を引き起こします。rs*1が*x0以外のゼロ値を保持するレジスタを指定した場合、命令は変更されていない値をCSRに書き戻そうとし、関連する副作用が発生することに注意してください。rs1=x0 の CSRRW は、宛先 CSR にゼロを書き込もうとします。

CSRRWI、CSRRSI、およびCSRRCIは、それぞれCSRRW、CSRRS、およびCSRRCに似ていますが、整数レジスタからの値ではなく、*rs1*フィールドにエンコードされた5ビット符号なし即時(uimm[4:0])フィールドをゼロ拡張して得られたXLENビット値を使用してCSRを更新します。CSRRSIおよびCSRRCIでは、uimm[4:0]フィールドがゼロの場合、これらの命令はCSRに書き込みません。CSRRWIの場合、*rd*=x0の場合、この命令はCSRを読み込んではならず、CSRの読み込み時に発生する可能性のある副作用を発生させてはなりません。CSRRSIとCSRRCIの両方とも、*rd*と*rs1*フィールドに関係なく、常にCSRを読み取り、読み取り時の副作用を発生させます。

表9.1は、CSR命令がCSRを読み書きするかどうかについての動作をまとめたものです。

*これまでに定義された CSR は、許可されていないアクセスに対して不正な命令例外を発生させる以外には、読み込みに対するアーキテクチャ上の副作用はありません。カスタム拡張では、読み込み時に副作用のある CSR を追加するかもしれません。*

命令退避カウンタinstretのようなCSRの中には、命令実行の副作用として変更されるものがあります。このような場合、CSRアクセス命令がCSRを読み込むと、命令実行前の値を読み込みます。CSRアクセス命令がそのようなCSRを書き込む場合は、インクリメントの代わりに書き込みを行います。特に、ある命令でインストレットに書き込まれた値は、次の命令で読み込まれた値になります。

CSR, CSRR *rd, csr,* を読み込むためのアセンブラ擬似命令は CSRRS *rd, csr, x0 としてエンコードされています*。CSRW *csr, rs1 を*書き込むためのアセンブラ擬似命令は CSRRW *x0, csr, rs1* としてエンコードされ、CSRWI *csr, uimm* は CSRRWI *x0, csr, uimm としてエンコードされます*。

古い値が不要な場合に CSR のビットをセットしたりクリアしたりするためのアセンブラ擬似命令がさらに定義されています。CSRS/CSRC *csr, rs1*; CSRSI/CSRCI *csr, uimm*.

### CSRアクセス順序

あるハート上で、明示的および暗黙的なCSRアクセスは、アクセスされたCSRの状態によって実行動作が影響を受ける命令に関して、プログラムの順序で実行されます。特に、CSRアクセスは、動作がCSRの状態によって変更されたり変更されたりするプログラム順の先行命令の実行後、動作がCSRの状態によって変更されたり変更されたりするプログラム順の後続命令の実行前に実行されます。さらに、CSRリードアクセス命令は命令実行前にアクセスされたCSR状態を返し、CSRライトアクセス命令は命令実行後にアクセスされたCSR状態を更新します。

上記のプログラム順序が保持されない場合、CSRアクセスの順序は弱く、ローカルハートや他のハートがプログラム順序とは異なる順序でCSRアクセスを観測することがあります。さらに、CSRアクセスが明示的なメモリアクセスを実行する命令の実行動作を変更しない限り、あるいは、CSRアクセスと明示的なメモリアクセスが、メモリモデルによって定義された構文依存性、あるいは、本マニュアル第2巻のメモリオーダリングPMAのセクションによって定義された順序付け要件のいずれかによって順序付けられない限り、CSRアクセスは明示的なメモリアクセスに関して順序付けられません。他のすべてのケースで順序付けを強制するために、ソフトウェアは関連するアクセスの間で FENCE 命令を実行しなければなりません。FENCE 命令の目的のために、CSR リードアクセスはデバイス入力 (I) として分類され、CSR ライトアクセスはデバイス出力 (O) として分類されます。

*非公式には、CSR空間は、このマニュアルの第2巻の「メモリー 順序 PMAs」のセクションで定義されているように、弱順序のメモリマップドI/O領域として機能します。その結果、他のすべてのアクセスに対するCSRアクセスの順序は、そのような領域に対するメモリマップされたI/Oアクセスの順序を制約するのと同じメカニズムによって制約されます。*

*これらのCSR順序付け制約は、主にメインメモリとメモリマップされたI/Oアクセスの順序付けをサポートするために、***time** *CSRのリードに関連して課されます。*

本仕様の第 1 巻および第 2 巻でこれまでに定義された CSR は、**time**、**cycle**、および **mcycle** CSR を除いて、他のハートや装置から直接アクセスできず、他のハートや装置から見える副作用も発生しません。*。このように、前記３つ以外のＣＳＲへのアクセスは、本明細書に違反することなく、ＦＥＮＣＥ命令に関して自由に並び替えることができます。*

副作用を引き起こすCSRアクセスについては、上記の順序制約は、副作用の開始順序には適用されますが、副作用の完了順序には必ずしも適用されません。

ハードウェア・プラットフォームは、本マニュアル第2巻の「メモリ 順序PMAs」のセクションで定義されているように、特定のCSRへのアクセスが強く順序付けられていることを定義することができます。強い順序のCSRへのアクセスは、弱い順序のCSRへのアクセスとメモリマップされたI/O領域へのアクセスの両方に対して、より強い順序制約を持っています。

**第10章**

# カウンター

RISC-V ISAには、最大32×64ビットのパフォーマンス・カウンタとタイマのセットが用意されており、非特権のXLENリードオンリーCSRレジスタ0ｘC00～0ｘC1F(上位32ビットはRV32のCSRレジスタ0ｘC80～0ｘC9F)を介してアクセスできます。これらのうち最初の3つのレジスタ(CYCLE、TIME、INSTRET)は、それぞれ専用の機能(サイクルカウント、リアルタイムクロック、命令終了)を持ち、残りのカウンタは、もし実装されていれば、プログラム可能なイベントカウントを提供します。

## 10.1 ベースカウンタとタイマ

拡張即時形を使用します。

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| csr | rs1 | funct3 | rd | オペコード |

１２ ５ ３ ５ ７

RDCYCLE[H] 0 CSRRS dest システム

RDTIME[H] 0 CSRRS dest システム

RDINSTRET[H] 0 CSRRS dest システム

RV32I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のユーザーレベル・カウンタを多数提供します。RV64Iでは、CSR命令は64ビットのCSRを操作することができます。特に、RDCYCLE、RDTIME、および RDINSTRET 擬似命令は、cycle、time、instret カウンタの 64 ビットのフルビットを読み出す。したがって、RDCYCLEH、RDTIMEH、およびRDINSTRETH命令はRV64Iでは必要ありません。

*実行環境によっては、タイミング・サイドチャネル攻撃を防ぐために、カウンタへのアクセスを禁止する場合があります。*

RDCYCLE疑似命令は、ハートが過去の任意の開始時刻から実行されているプロセッサコアによって実行されたクロックサイクル数のカウントを保持するサイクルCSRの下位XLENビットを読み出します。RDCYCLEHは、同じサイクルカウンタの63～32ビットを読み出すRV32I命令です。基礎となる64ビットカウンタは、実際には決してオーバーフローしないはずです。サイクル・カウンタが進む速度は、実装と動作環境に依存します。実行環境は、サイクル・カウンタがインクリメントしている現在のレート（サイクル/秒）を決定する手段を提供しなければなりません。

*RDCYCLE はプロセッサコアが実行したサイクル数を返すことを意図しており、ハートではありません。何が「コア」なのかを正確に定義するのは、いくつかの実装の選択(例えば、AMD Bulldozer)を考えると難しいです。何が「クロックサイクル」なのかを正確に定義するのは、（ソフトウェアエミュレーションを含む）さまざまな実装を考えると難しいですが、RDCYCLEは他のパフォーマンスカウンターと一緒にパフォーマンスの監視に使われることを意図しています。特に1つのハート/コアは、1つのハートのためのCPIを測定するためにサイクルカウント/命令が廃止されることが予想されます。。*

*コアはソフトウェアに全くさらされる必要はなく、実装者は、1つの物理コア上の複数のハートが1つのハート/コアで別々のコア上で動作しているように装い、各ハートに別々のサイクルカウンタを提供することを選択することができるかもしれません。これは、単純なバレルプロセッサ（例えば、CDC 6600 ペリフェラルプロセッサ）では、ハート間のタイミングの相互作用が存在しないか、または最小限に抑えられている場合には意味があるかもしれません。*

*複数のハート/コアがあり、動的なマルチスレッドがある場合、一般的にハートごとにサイクルを分離することはできません（特にSMTで）。特定のハートが実行しているサイクル数を捕捉しようとする別個のパフォーマンスカウンタを定義することは可能かもしれませんが、この定義は、すべての可能なスレッディング実装をカバーするために非常に曖昧でなければなりません。例えば、このハートの実行のために命令が発行されたサイクル、および/または命令がリタイアしたサイクルのみをカウントするか、このハートがマシンリソースを占有しているが、他のハートが実行に移っている間にストールして実行できなかったサイクルを含めるべきか？おそらく、理解しやすいパフォーマンス統計を得るためには、「上記のすべて」が必要になるでしょう。この複雑なハート・サイクル・カウントの定義と、マルチスレッド・コードのチューニング時のコアごとのサイクル・カウントの合計の必要性は、コア・サイクル・カウンター当たりの標準化につながり、共通の単一のハート/コア・ケースでもうまく機能します。*

*スリープ中に何が起こるかを標準化することは、「スリープ」が何を意味するかが実行環境全体で標準化されていないことを考えると現実的ではありませんが、もしコア全体が一時停止している（完全にクロックゲーティングされているか、または深いスリープ状態でパワーダウンしている）場合、それはクロックサイクルを実行していないことになり、サイクルカウントは仕様通りに増加するべきではありません。例えば、パワーダウンイベントからウェイクアップした後にプロセッサをリセットするのに必要なクロックサイクルをカウントすべきかどうかなど、多くの詳細がありますが、これらは実行環境固有の詳細と考えられています。*

*すべてのプラットフォームで動作する正確な定義はないにしても、これはほとんどのプラットフォームで有用な機能であり、ここでの不正確で、一般的な、”通常は正しい''標準は、標準がないよりも良いです。RDCYCLE の意図は主にパフォーマンスのモニタリング/チューニングであり、この仕様はその目標を念頭に置いて書かれています。*

RDTIME疑似命令は、任意の開始時刻から過去に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時間CSRの下位XLENビットを読み出す。RDTIMEHは、同じリアルタイムカウンタの63～32ビットを読み出すRV32I専用命令です。基礎となる64ビットカウンタは、実際には決してオーバーフローしないようにしなければなりません。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しなければなりません。周期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全てのハートのリアルタイムクロックは、リアルタイムクロックの1ティック以内に同期されなければなりません。環境は、クロックの精度を決定する手段を提供しなければなりません。

*いくつかの単純なプラットフォームでは、サイクルカウントは RDTIME の有効な実装を表すかもしれませんが、この場合、プラットフォームは RDCYCLE を使ってウォールクロック時間を測定するのではなく、コードをよりポータブルにするために RDCYCLE のエイリアスとして RDTIME 命令を実装すべきです。*

RDINSTRET疑似命令は、過去の任意の開始点からこのハートによってリタイアされた命令の数をカウントする**instret** CSRの下位XLENビットを読み出します。RDINSTRETHは、同じ命令カウンタの63～32ビットを読み出すRV32I専用命令です。実際には、基礎となる64ビットカウンタは決してオーバーフローしないはずです。

以下のコードシーケンスは、カウンタの上半分と下半分を読み出す間に下半分が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の値をx3:x2に読み出します。

again:

rdcycleh x3

rdcycle x2

rdcycleh x4

bne x3, x4, again

図10.1.RV32の64ビット・サイクル・カウンタを読み出すためのサンプル・コード。

*これらの基本的なカウンタは、基本的な性能分析、適応的および動的最適化、およびリアルタイムストリームを使用したアプリケーションの動作を可能にするために不可欠であるため、実装においてこれらの基本的なカウンタを提供することを推奨します。パフォーマンスの問題を診断するのに役立つ追加のカウンタを提供すべきであり、これら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べきです。*

*そうしないと、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のが非常に困難になるため、RV32でもカウンタは64ビット幅であることを要求しました。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することができます。上述のサンプル コードでは、個々の32 ビッ ト命令を使用して64 ビッ ト幅の値を安全に読み出す方法を示しています。*

*いくつかのアプリケーションでは、複数のカウンタを同じ瞬間に読み込めることが重要です。マルチタスク環境下で実行されている場合、ユーザースレッドがカウンタを読み込もうとしている間にコンテキストスイッチが発生する可能性があります。一つの解決策は、ユーザースレッドが他のカウンタを読み出す前と読み出した後にリアルタイムカウンタを読み出すことで、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。ユーザースレッドがアトミックにカウンタの値を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も検討しましたが、これはユーザコンテキストのサイズを大きくすることになり、特にカウンタのセットが豊富な実装の場合はそうなります。*

## 10.2　ハードウェアパフォーマンスカウンタ

29個の非特権64ビット・ハードウェア・パフォーマンス・カウンタhpmcounter3--hpmcounter31用にCSRスペースが割り当てられています。RV32の場合、これらのパフォーマンスカウンタの上位32ビットは、追加のCSR hpmcounter3h--hpmcounter31hを介してアクセスできます。これらのカウンタは、プラットフォーム固有のイベントをカウントし、追加の特権レジスタを使用して構成されます。これらの追加カウンタの数と幅、およびカウントするイベントのセットは、プラットフォーム固有のものです。

*特権アーキテクチャマニュアルでは、これらのカウンタへのアクセスを制御する特権CSRと、カウントするイベントを設定するための特権CSRについて説明しています。*

*最終的には、例えば浮動小数点命令の実行数などのISAレベルのメトリクスをカウントするためのイベント設定を標準化することが有用であり、また、「L1命令キャッシュのミス」のような一般的なマイクロアーキテクチャ上のメトリクスをカウントすることができるかもしれません。*

**第11章**

# “F"単精度浮動小数点の標準拡張機能、 バージョン2.2

この章では，単精度浮動小数点のための標準的な命令セット拡張について説明します．この拡張は “F"と呼ばれ，IEEE 754-2008の算術標準規格[7]に準拠した単精度浮動小数点演算命令を追加します．F拡張は、制御およびステータス・レジスタ・アクセスのための "Zicsr "拡張に依存しています。

## 11.1　F レジスタ状態

F拡張では、各32ビット幅の32個の浮動小数点レジスタf0～f31と、浮動小数点ユニットの動作モードと例外ステータスを格納する浮動小数点制御・ステータスレジスタfcsrが追加されます。この追加状態を図11.1に示します。RISC-V ISAの浮動小数点レジスタの幅を表すためにFLENという用語を使用し、F単精度浮動小数点拡張ではFLEN=32としています。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の値を操作します。浮動小数点ロードおよびストア命令は、レジスタとメモリ間で浮動小数点値を転送します。整数レジスタファイルとの間で値を転送する命令も提供されています。

*我々は、整数値と浮動小数点値の両方について統一されたレジスタファイルを考慮しましたが、これはソフトウェアのレジスタ割り当てと呼び出し規則を簡素化し、ユーザの総状態を削減するためです。しかし、分割編成は、与えられた命令幅でアクセス可能なレジスタの総数を増やし、広いスーパースカラ問題に十分なレジスタファイルポートの提供を簡素化し、デカップリングされた浮動小数点単位のアーキテクチャをサポートし、内部浮動小数点エンコーディング技術の使用を簡素化します。コンパイラのサポートと分割レジスタファイルアーキテクチャの呼び出し規則はよく理解されており、浮動小数点レジスタファイルの状態でダーティビットを使用することで、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。*

FLEN-1 ０

|  |
| --- |
| f0 |
| f1 |
| f2 |
| f3 |
| f4 |
| f5 |
| f6 |
| f7 |
| f8 |
| f9 |
| f10 |
| f11 |
| f12 |
| f13 |
| f14 |
| f15 |
| f16 |
| f17 |
| f18 |
| f19 |
| f20 |
| f21 |
| f22 |
| f23 |
| f24 |
| f25 |
| f26 |
| f27 |
| f28 |
| f29 |
| f30 |
| f31 |

FLEN

31 0

fcsr

32

図11.1.RISC-V標準F拡張単精度浮動小数点状態。

## 11.2　F 浮動小数点制御と状態レジスタ

浮動小数点制御・状態レジスタ(fcsr)はRISC-Vの制御・状態レジスタ(CSR)です。図11.2に示すように、浮動小数点演算の動的丸め込みモードを選択し、発生した例外フラグを保持する32ビットのリード/ライトレジスタです。

318 75 43210

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *予約済み* | 丸めモード(frm) | 未処理の例外 (フラグ) | | | | |
|  |  | NV | DZ | OF | UF | NX |

24 3 1 1 1 1 1

図11.2.浮動小数点制御およびステータスレジスタ。

fcsr レジスタは、基礎となる CSR アクセス命令の上に構築されたアセンブラ擬似命令である FRCSR 命令と FSCSR 命令で読み書きすることができます。FRCSR は fcsr を整数レジスタ *rd* にコピーして読み込みます。FSCSRは、元の値を整数レジスタ*rd*にコピーし、整数レジスタ*rs1*から取得した新しい値をfcsrに書き込むことで、fcsr内の値を入れ替えます。

fcsr内のフィールドは、異なるCSRアドレスから個別にアクセスすることもでき、これらのアクセスのために別個のアセンブラ擬似命令が定義されています。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。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ビットから得た新しい値をfrmに書き込むことにより、frmの値を入れ替えます。FRFLAGS と FSFLAGS は、未払例外フラグフィールド fflags に対してアナログ的に定義される。

fcsrの31～8ビットは他の標準拡張のために予約されており、10進浮動小数点のための “L"標準拡張を含みます。もし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、 読み込まれたときにはゼロの値を供給しなければならなりません。標準ソフトウェアはこれらのビットの内容を保持しなければなりません。

浮動小数点演算では、命令内にエンコードされたスタティックな丸めモード、または frm に保持されたダイナミックな丸めモードのいずれかを使用します。丸め込みモードは、表11.1に示すようにエンコードされています。命令の *rm* フィールドに 111 の値を指定すると、 frm に保持されているダイナミック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、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、不正な命令例外が発生します。ワイド化変換を含むいくつかの命令は、*rm*フィールドを持っていますが、それにもかかわらず、丸めモードの影響を受けません; ソフトウェアは、それらの*rm*フィールドをRNE (000)に設定する必要があります。

*C99 言語標準では、ダイナミック ラウンディング モード レジスタの提供が事実上義務付けられています。典型的な実装では、動的丸め込みモードの CSR ステートへの書き込みはパイプラインをシリアライズします。*

*静的丸めモードは、異なる丸めモードを頻繁に切り替える必要がある特殊な演算を実装するために使用されます。*

|  |  |  |
| --- | --- | --- |
| 丸めモード | ニーモニック | 意味 |
| 000 | RNE | 四捨五入して最も近く、偶数に縛る |
| 001 | RTZ | ゼロに向かって丸める |
| 010 | RDN | 切り捨て（-∞に向かって） |
| 011 | RUP | 切り上げ（+∞に向かって） |
| 100 | RMM | 四捨五入して最大振幅に縛る |
| 101 |  | 無効です。将来の使用のために予約されています。 |
| 110 |  | 無効です。将来の使用のために予約されています。 |
| 111 | DYN | 命令の rm フィールドでダイナミック丸めモードを選択します。  丸めモードレジスタでは、無効。 |

表 11.1.丸めモードの符号化。

発生した例外フラグは、表11.2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てから浮動小数点演算命令で発生した例外条件を示します。ベースとなるRISC-V ISAでは、浮動小数点例外フラグの設定によるトラップの発生をサポートしていません。

|  |  |
| --- | --- |
| フラグの表意記号 | フラグの意味 |
| NV | 無効な操作 |
| DZ | ゼロで割る |
| OF | オーバーフロー |
| UF | アンダーフロー |
| NX | 不正確 |

表11.2.未払例外フラグのエンコーディング

*標準で認められているように、基本ISAでは浮動小数点例外のトラップをサポートせず、代わりにソフトウェアでフラグの明示的なチェックを要求しています。浮動小数点未払例外フラグの内容によって直接制御されるブランチを追加することも検討しましたが、最終的にはISAをシンプルに保つためにこれらの命令を省略することにしました。*

## 11.3　NaNの生成と伝播

別段の記載がある場合を除き、浮動小数点演算の結果が NaN の場合、それは正規 NaN となります。正規 NaN は正の符号を持ち、MSB（別名クワイエットビット）以外のすべてのビットはクリアされます。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。

*標準規格で推奨されているように、NaN ペイロードを伝播することを検討したが、この決定はハードウェアコストを増加させることになります。さらに、この機能は標準規格ではオプションであるため、ポータブルコードでは使用できません。*

*実装者は、非標準の動作モードによって有効化された非標準の拡張として、ＮａＮペイロード伝搬スキームを自由に提供することができます。しかし、上述の正準ＮａＮスキームは常にサポートされなければならず、デフォルトモードであるべきです。*

*私たちは、例外的な条件の場合には、ユーザレベルのソフトウェア側でそれ以上の介入を行わずに（アルファISAの浮動小数点トラップ障壁とは異なり）、標準で義務付けられたデフォルト値を返すように実装することを要求します。*

*例外的なケースの完全なハードウェア処理が一般的になると考えているので、他のアプローチを最適化するためにユーザレベルのISAを複雑にすることは避けたい。実装では、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。*

## 11.4　準正規の算術

準正規数の操作は、IEEE 754-2008規格に準拠しています。

IEEE規格の用語では、丸め後に小さな値が検出されます。

*丸め後に小さな値を検出すると、スプリアスアンダーフロー信号が少なくなります。*

## 11.5　単精度ロードとストアの命令

浮動小数点のロードおよびストアは、整数ベースの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、ベース・アドレスはレジスタ*rs1、*12ビットの符号付きバイト・オフセットとなります。FLW命令は、メモリから単精度浮動小数点値を浮動小数点レジスタ*rd*にロードします。FSWは、浮動小数点レジスタ*rs2*から単精度の値をメモリに格納します。

義します。

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | オペコード |

１２ ５ ３ ５ ７

オフセット[11:0] base 幅 dest ロード-FP

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | オペコード |

7 5 ５ ３ ５ ７

オフセット[11:5] ｓｒｃ base 幅 オフセット[4:0] ストア-FP

FLWとFSWは、有効アドレスが自然に揃う場合のみ、アトミックに実行されることが保証されています。

FLWおよびFSWは、転送されるビットを変更しない；特に、非正規のNaNのペイロードは保存されます。

## １１.６　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持つ浮動小数点演算命令は、OP-FPメジャーオペコードのR型フォーマットを使用します。FADD.S、FMUL.Sは、rs*1*とrs*2の*間の単精度浮動小数点加算、乗算をそれぞれ行います。

FSUB.S は rs*1* から r*s2 の単精度*浮動小数点減算を行います。FDIV.Sはrs*1をrs2*で除算する単精度浮動小数点除算を行う。FSQRT.Sはrs*1の*平方根を計算します。いずれの場合も結果は*rd*に書き込まれます。

2 ビット浮動小数点フォーマットのフィールド *fmt* は、表 11.3 に示すようにエンコードされます。これは、F拡張のすべての命令に対して*S*(00)に設定されます。

|  |  |  |
| --- | --- | --- |
| ｆｍｔ領域 | 表意記号 | 意味 |
| 00 | S | 32ビット単精度 |
| 01 | D | 64ビット倍精度 |
| 10 | H | 16ビット半精度 |
| 11 | Q | 128ビット四倍精度 |

表 11.3：フォーマット領域のエンコーディング

丸めを行うすべての浮動小数点演算は、表11.1に示すエンコーディングを持つ *rm* フィールドを使用して丸めモードを選択することができます。

浮動小数点最小数命令 FMIN.S と最大数命令 FMAX.S は、それぞれ *rs1* と *rs2 の*小さい方または大きい方を *rd* に書き込みます。これらの命令の目的のためだけに、値-0*.0は*値+0*.*0よりも小さいとみなされます。両方の入力がNaNである場合、結果はカノニカルNaNとなる。一方のオペランドのみがNaNである場合、結果は非NaNオペランドとなります。シグナリングNaN入力は、結果がNaNでない場合でも、無効な操作例外フラグを設定します。

*F拡張のバージョン2.2では、FMIN.SおよびFMAX.S命令は、IEEE 754-2008のminNumおよびmaxNum演算ではなく、提案されたIEEE 754-201xのminimumNumberおよびmaximumNumber演算を実装するように修正されていることに注意してください。これらの演算は、シグナリングNaNの取り扱いが異なります。*

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | オペコード |

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

FSQRT S 0 src RM dest OP-FP

FMIN-MAX S src2 src1 MIN/MAX dest OP-FP

浮動小数点融合乗算加算命令には、新しい標準命令フォーマットが必要です。R4 タイプの命令では、3 つのソース レジスタ (*rs1*、rs*2*、およびrs*3*) とデスティネーション レジスタ (*rd*) が指定されます。このフォーマットは、浮動小数点フュージョン乗算加算命令でのみ使用されます。

FMADD.S は r*s1* と rs*2 の*値を掛け合わせ、rs*3 の*値を加算し、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 r*s1* と *rs2 の*値を掛け合わせ、rs*3 の*値を引き、最終的な結果を *rd* に書き込みます。FMSUB.S は *(rs1* × *rs2)-rs3 を計算します*。

FNMSUB.Sは、rs*1*とrs*2の*値を掛け合わせ、その積を否定し、rs*3*の値を加算し、最終結果を*rd*に書き込みます。FNMSUB.S は、*-(rs1 × rs2)+rs3 を*計算します。

FNMADD.Sはrs*1*とrs*2の*値を掛け合わせ、その積を否定し、rs*3の*値を引き算し、最終結果を*rd*に書き込みます。FNMADD.Sは*-(rs1 ×* rs*2)-rs3を*計算します。

*FNMSUB命令とFNMADD命令は、MIPS-IVの対応する命令の名前に由来しています。MIPS命令はRISC-V命令のように積を否定するのではなく、和を否定するように定義されていたため、当時の命名法はより合理的でした。2つの定義は、符号付きゼロの結果に関して異なります。RISC-Vの定義はx86とARMの融合乗算加算命令の動作と一致していますが、残念ながらRISC-VのFNMSUBとFNMADDの命令名はx86とARMに比べて入れ替わっています。*

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| rs3 | fmt | rs2 | rs1 | rm | rd | オペコード |

5 2 ５ 5 ３ 5 7

ｓｒｃ３ S src2 src1 RM dest F[N]MADD/F[N]MSUB

*融合乗算加算(FMA)命令は32ビット命令のエンコーディング空間の大部分を消費します。いくつかの代替案が検討されましたが、FMA は動的な丸め込みモードのみを使用するように制限されますが、静的な丸め込みモードは積丸め込みの欠如を利用するコードでは有用です。別の代替案としては、rs3 を提供するために rd を使用することも考えられますが、これはいくつかの一般的なシーケンスで追加の移動命令を必要とします。現在の設計では、FMA が非直交であることを避けつつ、32 ビットのエンコーディング空間の大部分を残しています。*

融合された乗算加算命令は、乗算が静かなＮａＮであっても、乗算値が∞と０である場合には、無効な演算例外フラグをセットしなければなりません。

*IEEE 754-2008標準は、∞ × 0 + qNaN 操作のための無効な例外を上げることを許可しますが、要求しません。*

## 11.7　単精度浮動小数点変換と移動命令

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エンコードされています。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 ビット整数または 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、FCVT.S.LUの各バリアントは、符号なし整数値への変換、または符号なし整数値からの変換を行います。XLEN*>32 の*場合、FCVT.W[U].S 符号は、32 ビットの結果をデスティネーション・レジスタ幅に拡張します。FCVT.L[U].S と FCVT.S.L[U] は RV64 専用命令です。丸められた結果がデスティネーション形式で表現できない場合は、最も近い値にクリッピングされ、無効フラグが設定されます。表11.4にFCVT.*int*.Sの有効な入力範囲と、無効な入力に対する動作を示します。

すべての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、 *rm* フィールドに従って丸められます。浮動小数点レジスタは、FCVT.S.W *rd*, x0を使用して浮動小数点正ゼロに初期化することができ、これにより例外フラグが設定されることはありません。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ＦＣＶＴ．W.S | FCVT.WU.S | FCVT.L.S | FCVT.LU.S |
| 最小有効入力(丸め後) | - 231 | 0 | - 263 | 0 |
| 最大有効入力（丸め込み後 | 231  - 1 | 232  - 1 | 263  - 1 | 264  - 1 |
| 範囲外マイナス入力時の出力 | - 231 | 0 | - 263 | 0 |
| Output for - \infty | - 231 | 0 | - 263 | 0 |
| 範囲外正入力時の出力 | 231  - 1 | 232  - 1 | 263  - 1 | 264  - 1 |
| 出力は＋＋の場合とNaNの場合 | 231  - 1 | 232  - 1 | 263  - 1 | 264  - 1 |

表 11.4.浮動小数点数から整数への変換の領域と、無効な入力に対する動作。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | オペコード |

5 2 ５ 5 ３ 5 7

FCVT.*int.fmt* S W[U]/L[U] src RM dest OP-FP

FCVT.*fmt.int* S W[U]/L[U] src 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, FSGNJX.S は、rs*1 の*符号ビットを除くすべてのビットを取る結果を生成します。FSGNJ の場合、結果の符号ビットは r*s2 の符号ビット*であり、FSGNJN の場合、結果の符号ビットは rs*2 の符号ビット*の逆であり、FSGNJX の場合、符号ビットは r*s1* と *rs2 の*符号ビットの XOR です。符号注入命令は浮動小数点例外フラグを設定したり、NaNを正規化したりしません。注意： FSGNJ.S *rx, ry, ry は、ry を rx* に移動させます（アセンブラ擬似命令 FMV.S *rx, ry*）； FSGNJN.S *rx, ry, ry は、ry* の否定を *rx* に移動させます（アセンブラ擬似命令 FNEG.S rx, ry）； FSGNJN.S *rx, ry, ry は、ry* の否定を *rx に*移動させます（アセンブラ擬似命令 FNeg.S *rx, ry*)、FSGNJX.S *rx, ry, ry は、ry* の絶対値を *rx* に移動させます (アセンブラ疑似命令 FABS.S *rx, ry*)。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | オペコード |

5 2 ５ 5 ３ 5 7

FSGNJ S src2 src1 J[N]/JX dest OP-FP

*符号注入命令は、浮動小数点の MV、ABS、および NEG を提供し、IEEE の 符号複写演算や超越的数学関数ライブラリの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 1 つのレジスタオペランドしか必要としませんが、FSGNJ 命令は 2 つのレジスタを必要としますが、ほとんどのマイクロアーキテクチャでは、これらの比較的頻度の低い命令のレジスタリードの数を減らすことで利益を得るために最適化を追加する可能性は低いでしょう。この場合でも、マイクロアーキテクチャは、FSGNJ命令の両方のソース・レジスタが同じである場合に、単純に検出して、1つのコピーだけを読み出すことができます。*

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させる命令が用意されています。

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。この転送ではビットは変更されず、特に非正規ＮａＮのペイロードは保存されます。ＲＶ６４の場合、転送先レジスタの上位３２ビットは、浮動小数点数の符号ビットのコピーで満たされます。

FMV.W.Xは、IEEE 754-2008標準符号化で符号化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。この転送ではビットは変更されず、特に非正規ＮａＮのペイロードは保存されます。

*FMV.W.XとFMV.X.W命令は、以前はFMV.S.XとFMV.X.Sと呼ばれていましたが、Wの使用は、32ビットを解釈せずに移動する命令としての意味合いに合致しています。これは、NaN-boxingを定義した後に明らかになりました。既存のコードの邪魔にならないように、W版とS版の両方がツールでサポートされることになりました。*

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | オペコード |

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

*基本の浮動小数点 ISA は、レジスタ内の浮動小数点フォーマットの内部リコーディングを使用して、正規値以下の値の処理を簡素化し、機能単位のレイテンシを削減することができるように定義されています。この目的のために、基本ISAでは、整数レジスタファイルを直接読み書きする変換・比較演算を定義することで、浮動小数点レジスタで整数値を表現することを避けています。これにより、整数レジスタと浮動小数点レジスタの間で明示的に移動する必要がある多くの一般的なケースを排除し、一般的な混在フォーマットのコードシーケンスの命令数とクリティカルパスを削減します。*

## 11.8　単精度浮動小数点比較命令

浮動小数点比較命令（FEQ.S, FLT.S, FLE.S）は、浮動小数点レジスタ（rs*1* = *rs2*, rs*1 < rs2, rs*1 *≦ rs2））*間で指定された比較を行い、条件が成立した*場合は*整数レジスタ rd に 1 を書き込み、そうでない場合は 0 を書き込みます。

FLT.SとFLE.Sは、IEEE 754-2008規格で*シグナリング*比較と呼ばれているものを実行します：つまり、どちらかの入力がNaNの場合に無効な動作例外フラグを設定します。FEQ.Sは、どちらかの入力がシグナリングNaNである場合にのみ無効な動作例外フラグを設定します。3つの命令すべてにおいて、どちらかのオペランドがNaNであれば、結果は0になります。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | オペコード |

5 2 ５ 5 ３ 5 7

FCMP S src2 src1 EQ/LT/LE dest OP-FP

*F 拡張では a≦比較が提供されるのに対し、ベースの ISA では、a≧分岐比較が提供されます。≦*は≧*から合成することができ、またその逆もできるので、この矛盾には性能上の意味はありませんが、それにもかかわらず、ISAにおける不幸な不調和です。*

## 11.9　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を表11.5に示します。*rd* の対応するビットは、プロパティが真であればセットされ、そうでなければクリアされます。*rd の*他のすべてのビットはクリアされます。*rd の*中の 1 ビットだけが設定されることに注意してください。FCLASS.Sは浮動小数点例外フラグを設定しません。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | オペコード |

5 2 ５ 5 ３ 5 7

FCLASS S 0 src 001 dest OP-FP

|  |  |
| --- | --- |
| ｒ*ｄ*ビット | 意味 |
| 0 | *rs1 は* - ∞ です。 |
| 1 | *rs1は*負の正規数です。 |
| 2 | *rs1は*負の亜正規数です。 |
| 3 | *rs1は*-0です。 |
| 4 | *rs1は*+0です。 |
| 5 | *rs1は*正の非正規数です。 |
| 6 | *rs1は*正の正規数である。 |
| 7 | *rs1 は + ∞ です。* |
| 8 | *rs1は*シグナリングNaNである。 |
| 9 | *rs1は*静かなNaNです。 |

表 11.5：FCLASS 命令の結果のフォーマット

**第12章**

# “D"倍精度浮動小数点の標準拡張機能、 バージョン2.2

この章では、IEEE 754-2008 算術標準に準拠した倍精度浮動小数点演算命令を追加した標準の倍精度浮動小数点命令セット拡張 “D"について説明します。D拡張は基本の単精度命令セットFに依存しています。

## 12.1　D レジスタ状態

D拡張により、32個の浮動小数点レジスタf0～f31が64ビットに拡張されました(図11.1のFLEN=64)。fレジスタは、以下のセクション12.2で説明するように、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、Q の拡張子がサポートされているかに応じて、32、64、128 のいずれかにすることができます。H、F、D、Q を含む最大 4 つの異なる浮動小数点精度をサポートしています。*

## 12.2　NaN 狭義値の箱詰め

複数の浮動小数点精度がサポートされている場合、*n <* FLEN の狭い *n* ビット型の有効な値は、FLEN ビットの NaN 値の下位 n ビットで表現され、NaN-箱詰め と呼ばれる処理が行われます。有効なNaN-箱詰めされた値の上位ビットはすべて1でなければなりません。したがって、有効な NaN-箱詰めされた *n-*bit 値は、より広い *m*-bit 値、*n < m ≦* FLEN として見た場合、負の静かな NaN (qNaNs) として表示されます。より狭い結果をfレジスタに書き込む操作は、合法的なNaN-箱詰めされた値を得るために、最上位のFLEN - *n*ビットにすべての1を書き込まなければなりません。

*ソフトウェアは浮動小数点レジスタに保存されているデータの現在のタイプを知らないかもしれませんが、レジスタの値を保存したり復元したりすることができなければならないので、より狭い値を転送するためにより広い演算を使用する結果を定義しなければなりません。一般的なケースは、呼び出し先が保存したレジスタの場合ですが、可変個引数、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、デバッグなどの機能についても、標準の規則が望ましいです。*

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レジスタに出し入れし、浮動小数点ロード/ストア(*FLn*/*FSn*)と浮動小数点移動命令(FMV.*n.*X/FMV.X.*n*)で構成されています。f レジスタへの n *<* FLEN の狭い *n* ビット転送は、有効な NaN-箱詰めされた値を生成します。浮動小数点レジスタからの狭い*nビットの転送は*、上位の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。

前段落で説明した転送演算とは別に、*n <* FLEN の狭い *n* ビット演算に対する他のすべての浮動小数点演算は、入力オペランドが正しく NaN-箱詰めされた であるかどうか、すなわち、FLEN - n の上位ビットがすべて 1 であるかどうかをチェックします。もしそうであれば、入力の*n個の*最下位ビットが入力値として使用され、そうでなければ入力値は*n*ビットの正準NaNとして扱われます。

*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や幅の広いオペランドの結果を演算に与える動作は定義されていませんでしたが、幅の広い保存や復元が幅の狭いオペランドの値を保持することを要求する以外は、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を削除し、浮動小数点単位の非コード化およびコード化されていない実装の両方に対応しています。新しい定義はまた、値が誤って使用された場合にNaNを伝播することで、ソフトウェアエラーを検出するのに役立ちます。*

*非符号化実装では、各浮動小数点演算の入出力でオペランドをIEEE標準フォーマットにアンパックしてパックします。非復号化実装に対するNaN-箱詰めのコストは、主に、絞り込み演算の上位ビットが正当なNaN-箱詰めされた値を表すかどうかをチェックすることと、結果の上位ビットにすべての1を書き込むことにあります。*

*再コード化された実装では、浮動小数点値を表現するために便利な内部フォーマットを使用し、指数ビットを追加してすべての値を正規化して保持できるようにします。再コード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、これは指数フィールドにNaNを内部的に再コードすることで、新しいステートビットを追加することなく行うことができます。再符号化されたフォーマットの値を内外に転送するために使用されるパイプラインに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に抑えられます。再符号化プロセスは、いずれの場合もワイドオペランドの入力亜正規値のシフトを処理しなければならず、NaN箱詰めされた値の抽出は、先頭0ビットをスキップする代わりに先頭1ビットをスキップすることを除いて正規化と同様のプロセスであり、データパスの多重化を共有することを可能にしています。*

## 12.3　倍精度のロードとストアの命令

FLD命令は、メモリから倍精度の浮動小数点値を浮動小数点レジスタ*RD*にロードします。FSDは、浮動小数点レジスタからメモリに倍精度の値を格納します。

*倍精度値は、ＮａＮ箱詰めされた単精度値であってもよい。*

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | width | rd | オペコード |

１２ ５ ３ ５ ７

オフセット[11:0] base D dest ロード-FP

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:5] | rs2 | rs1 | width | imm[4:0] | オペコード |

7 5 ５ ３ ５ ７

オフセット[11:5] ｓｒｃ base D オフセット[4:0] ストア-FP

FLDとFSDは、有効アドレスが自然に整列され、XLEN≧64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。

FLDおよびFSDは、転送されるビットを変更しない；特に、非正規のNaNのペイロードは保存される。

## 12.4　倍精度浮動小数点計算命令

倍精度浮動小数点演算命令は、単精度浮動小数点演算命令と同様に定義されていますが、倍精度のオペランドで動作し、倍精度の結果が得られます。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | オペコード |

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

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| rs3 | fmt | rs2 | rs1 | rm | rd | オペコード |

5 2 ５ 5 ３ 5 7

ｓｒｃ３ D src2 src1 RM dest F[N]MADD/F[N]MSUB

## 12.5　倍精度浮動小数点変換と動作命令

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 主要なオペコード空間でエンコードされています。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 ビット整数または 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 ビット符号付き整数または 64 ビット符号付き整数を、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、FCVT.D.LUの各バリアントは、符号なし整数値への変換、または符号なし整数値からの変換を行います。

RV64 では、FCVT.W[U].D 符号は32 ビットの結果を拡張します。FCVT.L[U].D と FCVT.D.L[U] は RV64 専用命令です。FCVT.int.Dの有効入力範囲と無効入力時の動作は、FCVT.int.Sと同じです。

すべての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、 *rm* フィールドに従って丸められます。注意 FCVT.D.W.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 | オペコード |

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、単精度から倍精度変換命令 FCVT.S.D、FCVT.D.S は、OP-FP 主要なオペコード空間で符号化されており、送信元、送信先ともに浮動小数点レジスタとなっています。*rs2*フィールドはソースのデータ型をコードし、*fmt*フィールドはデスティネーションのデータ型をコードします。FCVT.D.S.D は RM フィールドに従ってラウンドし、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 | オペコード |

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注入命令 FSGNJ.D, FSGNJN.D, FSGNJX.D は、単精度符号注入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 | オペコード |

5 2 ５ 5 ３ 5 7

FSGNJ D src2 src1 J[N]/JX dest OP-FP

XLEN≧64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させる命令が用意されています。FMV.X.Dは、浮動小数点レジスタ*rs1*の倍精度の値を、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の値を整数レジスタ*rs1*から浮動小数点レジスタ*rdに移動します*。

FMV.X.DとFMV.D.Xは転送されるビットを変更せず、特に非正規のNaNのペイロードは保存されます。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | オペコード |

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

*RISC-V ISAの初期バージョンには、RV32システムが64ビット浮動小数点レジスタと整数レジスタの上位と下位の間を転送できるようにする命令が追加されていました。しかし、これらの命令はレジスタの部分的な書き込みが可能な唯一の命令であり、再コード化された浮動小数点やレジスタの名前を変更する実装では複雑さが増し、パイプラインでの読み出し・変更・書き込み・手順が必要になります。RV32とRV64の4倍精度を扱うようにスケールアップするには、このパターンを踏襲するために追加の命令が必要になります。ISAは、変換と比較の結果を適切なレジスタファイルに書き込むことで、明示的な整数-浮動小数レジスタの移動の数を減らすように定義されているため、これらの命令の恩恵は他のISAよりも少ないと予想されます。*

*融合乗算のサポートと 64 ビット浮動小数点ロードとストアを含む 64 ビット浮動小数点ユニットを実装するシステムでは、32 ビットから 64 ビット整数データパスへの移行にかかる限界ハードウェアコストが低く、32 ビット幅のアドレス空間とポインタを使用することで、静的データや動的なメモリトラフィックの増加を避けることができます。*

## 12.6　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、単精度浮動小数点比較命令と同様に定義されていますが、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 | オペコード |

5 2 ５ 5 ３ 5 7

FCMP D src2 src1 EQ/LT/LE dest OP-FP

## 12.7　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、単精度と同様に定義されていますが、倍精度の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 | オペコード |

5 2 ５ 5 ３ 5 7

FCLASS D 0 src 001 dest OP-FP

**第13章**

# “Q"4倍精度浮動小数点の標準拡張機能、 バージョン2.2

この章では、IEEE 754-2008 算術標準規格に準拠した128ビットの2進数浮動小数点命令のQ規格拡張について説明します。この拡張は倍精度浮動小数点拡張Dに依存します。浮動小数点レジスタは単精度浮動小数点値、倍精度浮動小数点値、または倍精度浮動小数点値のいずれかを保持するように拡張されました(FLEN=128)。12.2節で説明したNaN-箱詰めスキームが再帰的に拡張され、単精度の値が倍精度の値の中にNaN-箱詰めされ、それ自体が4倍精度の値の中にNaN-箱詰めされるようになりました。

## 13.1　四角精度のロードおよびストア命令

LOAD-FPとSTORE-FP命令の新しい128ビットのバリエーションが追加され、funct3幅フィールドの新しい値でエンコードされました。

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | width | rd | オペコード |

１２ ５ ３ ５ ７

オフセット[11:0] base D dest ロード-FP

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:5] | rs2 | rs1 | width | imm[4:0] | オペコード |

7 5 ５ ３ ５ ７

オフセット[11:5] ｓｒｃ base D オフセット[4:0] ストア-FP

FLQとFSQは、有効アドレスが自然に整列されていて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ます。

FLQとFSQは、転送されるビットを変更しない。特に、非正規のNaNのペイロードは保存されます。

## 13.2　四倍精度計算命令

表13.1に示すように、ほとんどの命令のフォーマットフィールドに新しいサポートされるフォーマットが追加されました。

|  |  |  |
| --- | --- | --- |
| *fmt*領域 | ニーモニック | 意味 |
| 00 | S | 32ビット単精度 |
| 01 | D | 64ビット倍精度 |
| 10 | H | 16ビット半精度 |
| 11 | Q | 128ビット四倍精度 |

表13.1.フォーマットフィールドのエンコーディング。

四倍精度浮動小数点演算命令は、倍精度浮動小数点演算命令と同様に定義されていますが、四倍精度のオペランドで演算を行い、四倍精度の結果を生成します。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | オペコード |

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

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| rs3 | fmt | rs2 | rs1 | rm | rd | オペコード |

5 2 ５ 5 ３ 5 7

ｓｒｃ３ Q src2 src1 RM dest F[N]MADD/F[N]MSUB

## 13.3 四倍精度変換と移動命令

浮動小数点から整数への変換命令と整数から浮動小数点への変換命令を追加しました。これらの命令は、倍精度から整数への変換命令、整数から倍精度への変換命令と同様に定義されています。FCVT.W.Q または FCVT.L.Q は、それぞれ 4 倍精度浮動小数点数を符号付き 32 ビットまたは 64 ビット整数に変換します。FCVT.Q.WまたはFCVT.Q.Lは，それぞれ32ビットまたは64ビットの符号付き整数を，4倍精度の浮動小数点数に変換します。FCVT.WU.Q、FCVT.LU.Q、FCVT.Q.WU、FCVT.Q.LUの各バリアントは、符号なし整数値への変換、または符号なし整数値からの変換を行います。FCVT.L[U].Q および FCVT.Q.L[U] は RV64 専用命令です。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | オペコード |

5 2 ５ 5 ３ 5 7

FCVT.*int.Q* Q W[U]/L[U] src RM dest OP-FP

FCVT.*Q.int* Q W[U]/L[U] src RM dest OP-FP

新しい浮動小数点-浮動小数点変換命令が追加されました。これらの命令は、倍精度浮動小数点から浮動小数点への変換命令と同様に定義されています。FCVT.S.Q または FCVT.Q.S は、それぞれ 4 倍精度浮動小数点数を単精度浮動小数点数に、またはその逆に変換します。FCVT.D.Q、FCVT.Q.Dは、それぞれ4倍精度浮動小数点数を倍精度浮動小数点数に変換します。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | オペコード |

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注入命令 FSGNJ.Q, FSGNJN.Q, FSGNJX.Q は、倍精度符号注入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 | オペコード |

5 2 ５ 5 ３ 5 7

FSGNJ Q src2 src1 J[N]/JX dest OP-FP

FMV.X.QとFMV.Q.X命令はRV32やRV64には用意されていないので、メモリを介して整数レジスタに移す必要があります。

*RV128では、Q拡張でFMV.X.QとFMV.Q.Xに対応します。*

## 13.4　四倍精度浮動小数点比較命令

四倍精度浮動小数点比較命令は、その倍精度カウンターパートと類似して定義されていますが、四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 | オペコード |

5 2 ５ 5 ３ 5 7

FCMP Q src2 src1 EQ/LT/LE dest OP-FP

## 13.5　四倍精度浮動小数点分類命令

四倍精度浮動小数点分類命令FCLASS.Qは、倍精度と同様に定義されていますが、四倍精度の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 | オペコード |

5 2 ５ 5 ３ 5 7

FCLASS Q 0 src 001 dest OP-FP

**第14章**

# RVWMOメモリ一貫性モデル、 バージョン0.1

この章では、RISC-Vのメモリ一貫性モデルを定義します。メモリ一貫性モデルとは、メモリのロードによって返される値を指定する規則の集合です。RISC-Vでは ”RVWMO"と呼ばれるメモリモデルを使用しています。(RISC-V 弱いメモリ順 Weak Memory Ordering)は、アーキテクトが高性能でスケーラブルなデザインを構築するための柔軟性を提供すると同時に、扱いやすいプログラミングモデルをサポートするように設計されています。

RVWMOの下では、単一のハート上で実行されているコードは、同じハート内の他のメモリ命令から見れば順番通りに実行されているように見えますが、別のハートからのメモリ命令は、最初のハートからのメモリ命令が異なる順番で実行されていることを観察するかもしれません。そのため、マルチスレッドコードでは、異なるハートからのメモリ命令間の順序を保証するために、明示的な同期が必要になる場合があります。ベースとなるRISC-V ISAはこの目的のためのFENCE命令を提供しており、セクション2.7で説明しましたが、アトミック拡張 “A"はロードリザーブ/ストア条件付き命令とアトミックリード・モディファイ・ライト命令を追加で定義しています。

アトミックの位置ずれのための標準ISA拡張 “Zam" (第22章)と、ストア全体の順序付けのための標準ISA拡張 “Ztso"(第23章)では、RVWMOを拡張して、これらの拡張に固有の規則を追加しています。

本仕様書の付録は、メモリ一貫性モデルの公理的形式化と操作的形式化の両方と、追加の説明資料を提供します。

*この章では、通常のメインメモリ操作のためのメモリモデルを定義します。I/O メモリ、命令フェッチ、FENCE.I、ページテーブルウォーク、SFENCE.VMA との相互作用は(まだ)正式化されていません。上記のいくつかまたはすべては、将来のこの仕様のリビジョンで正式化されるかもしれません。RV128ベースのISAと将来のISA拡張機能である”V"ベクトル、”T"トランザクショナルメモリ、”J"JITの拡張機能は、今後のリビジョンにも取り入れる必要があります。*

*異なる幅のメモリアクセスの重複を同時にサポートするメモリ一貫性モデルは、学術研究の活発な分野であり、まだ完全には理解されていません。RVWMOの下で異なるサイズのメモリアクセスがどのように相互作用するかについての詳細は、現在の能力の範囲内で規定されていますが、新たな問題が発見された場合には修正される可能性があります。*

## 14.1 RVWMOメモリモデルの定義

RVWMOメモリモデルは、*広域・メモリ・順序*、つまり、すべてのハートで生成されるメモリ操作の合計順で定義されています。一般的に、マルチスレッド・プログラムは多くの異なる実行が可能で、各実行はそれぞれに対応する広域・メモリ・順序を持っています。

グローバルメモリ順序は、メモリ命令によって生成されたプリミティブのロードおよびストア操作にわたって定義されます。そして、この章の残りの部分で定義されている制約に従います。メモリモデルの制約をすべて満たす実行は、（メモリモデルに関する限り）合法的な実行です。

### メモリモデルのプリミティブ

メモリ操作に対する*プログラムの順序は*、各ロードおよびストアを生成する命令が、そのハートの動的命令ストリームに論理的に配置されている順序、すなわち、単純な順序のプロセッサがそのハートの命令を実行する順序を反映しています。

メモリにアクセスする命令は、*メモリ操作を発生させます*。メモリ操作は、*ロード操作*、*ストア操作、*またはその両方を同時に行うことができます。すべてのメモリ操作は単一コピーの原子的なものであり、部分的に完全な状態では観察できません。

RV32GCとRV64GCの命令のうち、アラインメントされた各メモリ命令では、2つの例外を除いて正確に1つのメモリ操作が発生します。第一に、SC命令が失敗してもメモリ操作は発生しません。次に、FLD命令とFSD命令は、123項で述べたように、XLEN<64の場合、それぞれ複数のメモリ操作を生じさせます。アラインドされたAMOは、ロードとストアの両方を同時に行う1つのメモリ操作となります。

*RV128ベースの命令セットや、V(ベクトル)やP(SIMD)のような将来のISA拡張命令では、複数のメモリ演算が発生する可能性があります。しかし、これらの拡張命令のメモリモデルはまだ正式には定義されていません。*

ロード命令またはストア命令は、任意の粒度のコンポーネント・メモリ操作のセットに分解することができます。XLEN*<64の*FLD命令またはFSD命令もまた、任意の粒度のコンポーネント・メモリ操作のセットに分解されてもよい。このような命令によって生成されたメモリ操作は、プログラムの順序では互いに順番が合わないが、プログラムの順序では、前後の命令によって生成されたメモリ操作に対しては、正常に順番が合わされます。アトミック拡張 “A"では、実行環境が不揃いなアトミック命令をサポートする必要はありませんが、もし不揃いなアトミックがZam拡張でサポートされている場合、第22章で定義されている不揃いなアトミックのための原子性公理の制約のもとでLR、SC、AMOを分解することができます。

*ミスアラインメントメモリ操作をバイト単位に分解することで、ミスアラインメントアクセスをネイティブにサポートしていない実装でのエミュレーションが容易になります。そのような実装では、例えば、単にミスアラインドアクセスのバイトを1つずつ反復処理することができます。*

LR命令とSC命令は、プログラム順にLRがSCに先行し、その間に他のLR命令やSC命令が存在しない場合、対になっているといい、対応するメモリ操作も対になっているといいます（SCが失敗した場合、ストア操作が発生しない場合を除く）。SCが成功しなければならないか、成功してもよいか、失敗しなければならないかを決定する条件の完全なリストは8.2節で定義されています。

また、ロードおよびストア操作には、以下のような1つまたは複数の順序付けアノテーションを付けることができます。"acquire-RCpc"、"acquire-RCsc"、"release-RCpc"、"release-RCsc"。aqが設定されたAMOまたはLR命令は、"acquire-RCsc "アノテーションを持ちます。rlが設定されているAMOまたはSC命令には、"release-RCsc "が設定されています。AMO、LR、SCのうち、aqとrlの両方が設定されている命令は、"acquire-RCsc "と "release-RCsc "の両方のアノテーションを持ちます。

ここでは便宜上、アクワイア・アノテーションをアクワイア-RCpcアノテーションまたはアクワイア-RCscアノテーションと呼ぶことにします。同様に、releaseアノテーションはrelease-RCpcアノテーションまたはrelease-RCscアノテーションを指します。RCpcアノテーション」とは、Acquisition-RCpcアノテーションまたはrelease-RCpcアノテーションを指します。RCscアノテーション」とは、RCscアノテーションの取得またはRCscアノテーションの解放を意味します。

*メモリモデルの文献では、 “RCpc"という用語は、プロセッサと一貫した同期操作のリリースの一貫性を意味し、 “RCsc"という用語は、逐次的に一貫した同期操作のリリースの一貫性を意味します[5]。*

*アノテーションの取得と公開については、文献には多くの異なる定義があるが、RVWMOの文脈では、これらの用語は、保存プログラム・オーダー規則5～7で簡潔かつ完全に定義されています。*

*“RCpc"アノテーションは現在のところ、標準の拡張機能 “Ztso"第(23章)のように全てのメモリアクセスに暗黙的に割り当てられている場合にのみ使用されます。さらに、ISAには現在のところネイティブのロードアクワイア命令やストアリリース命令、またRCpcの変種は含まれていませんが、RVWMOモデル自体は、将来の拡張で上記のいずれかまたはすべてがISAに追加される可能性があるため、前方互換性があるように設計されています。*

### 構文的な依存関係

RVWMOメモリモデルの定義は、以下のように定義される構文依存性の概念に部分的に依存しています。

依存関係を定義する文脈では、”register''は汎用レジスタ全体、CSRの一部、またはCSR全体を指します。CSRを通して依存関係を追跡する粒度はそれぞれのCSRに固有であり、セクション14.2で定義されています。

構文依存関係は、命令の*ソースレジスタ*、命令の*デスティネーションレジスタ*、および命令がソースレジスタからデスティネーションレジスタに*依存関係を運ぶ*方法の観点から定義されます。このセクションでは、これらすべての用語の一般的な定義を提供しますが、セクション14.3では、各命令の仕様の完全なリストを提供します。

一般的に、x0以外のレジスタ*rは、*以下のいずれかがホールドされている場合、命令*iのソースレジスタ*となります。

* *iの*オペコードでは、*rs1*、rs*2*、またはrs*3がr*に設定されています。
* *i は* CSR 命令であり、*i* のオペコードでは、*i が* CSRRW または CSRRWI で *rd が* x0 に設定されていない限り、*csr は r* に設定されます。
* *rは*CSRであり、セクション14.3で定義されているように、*iに対する暗*黙のソースレジスタです。
* *rは*、*i*に対する別のソースレジスタとエイリアスするCSRです。

また、メモリ命令は、どのソースレジスタが*アドレスソースレジスタ*であり、どのソースレジスタが*データソースレジスタ*であるかをさらに特定します。

一般的に、x0以外のレジスタ*rは、*以下のいずれかがホールドされていれば、命令*iのデスティネーションレジスタ*となります。

* *iのオペコードではrdがrに設定されています。*
* *ii は CSR 命令であり、i のオペコードでは、csr が r に設定されます。ただし、i が CSRRS または CSRRC で rs1 が x0 に設定されている場合、または i が CSRRSI または CSRRCI で uimm[4:0] がゼロに設定されている場合は除きます。*
* rはCSRであり、14.3項で定義されているように、iの暗黙のデスティネーションレジスタです。
* rは、iの別のデスティネーション・レジスタとエイリアスするCSRです。

ほとんどの非メモリ命令は、それぞれのソースレジスタからそれぞれのデスティネーションレジスタへの*依存関係を持ちます。*しかし、この規則には例外があります。

命令*jは、*以下のいずれかが保持されている場合、*i*のデスティネーションレジスタ*s*と*j*のソースレジスタ*rを*介して、命令*i*に*構文的に依存してい*ます。

* *s は r と同じで*あり，*i* と *j の*間で順序付けされた命令プログラムで *r を*宛先レジスタとして持つものはありません。
* *i* と *j の*間にプログラム順に並べられた命令 *m が*あり、以下のすべてが成立する。

1. *j は、*デスティネーションレジスタ *q* とソースレジスタ *r を*介して *m* に構文的に依存しています。
2. *m は、*宛先レジスタ *s* とソースレジスタ *p を*介して *i* に構文的に依存しています。
3. *mはp*から*q*への依存関係を持っています。

最後に、以下の定義において、*a*と*bを*2つのメモリ操作とし、*i*と*jを*それぞれ*a*と*bを*生成する命令とします。

*bは、rがjのアドレス・ソース・レジスタであり、jがソース・レジスタrを介してiに構文上の依存関係を持っている場合、aに構文上のアドレス依存関係を持っています*。

*bがストア操作であり、rがjのデータ・ソース・レジスタであり、jがソース・レジスタrを介してiに構文上の依存関係を有する場合、bはaに構文上のデータ依存関係をもっています*。

*bは、iとjの間にプログラムオーダーされた命令mが存在し、mが分岐または間接ジャンプであり、mがiに構文上の依存性を持つ場合、aに構文上の制御依存性を持っています*。

*一般的に、非AMOロード命令はデータソースレジスタを持たず、無条件に非AMOストア命令はデスティネーションレジスタを持ちません。しかし、成功したSC命令は、*rd*で指定されたレジスタを宛先レジスタとして持っていると考えられ、したがって、ある命令は、プログラム順序でそれより前にある成功したSC命令に構文的に依存性を持つことが可能です。*

### プリザーブドプログラムオーダー

プログラムの任意の実行のためのグローバルメモリ順序は、各ハートのプログラム順序の一部を尊重しますが、すべてを尊重するわけではありません。グローバルメモリ順序によって尊重されなければならないプログラム順序のサブセットは、*保存されたプログラム順序*として知られています。

保存されたプログラム順序の完全な定義は次のとおりです（なお、AMOはロードとストアの両方を同時に行う）：プログラム順でaがbに先行し、aとbの両方が（I/O領域ではなく）通常のメインメモリにアクセスし、かつ以下のいずれかが成立する場合、保存されたプログラム順（したがってグローバルメモリ順）でメモリ操作aがメモリ操作bに先行します：

* 重複するアドレスの順序：

1. *b は*ストアであり、*a* と *b は*重複するメモリアドレスにアクセスします
2. *a* と *b は*ロードであり、*x は a* と *b の*両方によって読み込まれたバイトであり、プログラムの順序で *a* と *b の*間に *x* へのストアはなく、*a* と b *は*異なるメモリ操作によって書き込まれた *x の*値を返します
3. *aはAMOまたはSC命令によって生成され、bはロードであり、bはaによって書き込まれた値を返します*

* 明示的同期化

1. *bの*前に*aを*命令するFENCE命令があります
2. *ａには*取得注釈があります
3. *b は*リリースアノテーションを持っています
4. *a* と *b は*共に RCsc アノテーションを持ちます
5. *ａ*と*ｂが*対になっています

* 構文的な依存関係

1. *b は*、構文的なアドレス依存性を持ちます
2. *b は*、構文データの依存関係を持ちます
3. *b は*ストア*で*あり、*b は*構文制御依存性を持ちます

* パイプラインの依存関係

1. *b は*ロードであり、プログラムの順序で *a* と *b の*間には、*m が a* にアドレスまたはデータの依存関係を持ち、*b は m* によって書かれた値を返すようなストア *m が存在します*
2. *b は*ストアであり、プログラムの順序で *a* と *b の*間に、m *が a* にアドレス依存性を持つような命令 *m が存在します*

### メモリモデルの公理

RISC-Vプログラムの実行は、保存されたプログラム順序に準拠した大域的なメモリ順序が存在し、*負荷値公理*、*原子性公理*、*進行公理を満たす*場合にのみ、RVWMOメモリ一貫性モデルに従います。

**荷重値公理** 各ロード*iの*各バイトは、以下のストアのうち、グローバルメモリ順の最新のストアがそのバイトに書き込んだ値を返します：

1. そのバイトを書き込み、グローバルメモリの順序で *i* より前のバイトを格納します
2. そのバイトを書き込み、プログラム順に *i* より前のバイトを格納します

**原子性公理** rとwがhというハートの中でLR命令とSC命令がアラインされて生成されたロードとストアのペア操作であり、sがバイトxへのストアであり、rがsによって書き込まれた値を返す場合、グローバルメモリの順序ではsがwに先行しなければならず、h以外のハートからグローバルメモリの順序でsに続いてwに先行するバイトxへのストアは存在し得なません。

*原子性公理は理論的には、異なる幅のLR/SCペアや不一致アドレスをサポートしており、実装ではそのような場合でもSC演算を成功させることが許されています。しかし、実際にはそのようなパターンは稀であり、使用は推奨されません。*

**進捗公理** どのメモリ操作も、グローバルメモリの順序では、他のメモリ操作の無限のシーケンスに先行してはなりません。

## 14.2 CSR 依存性追跡の粒度

|  |  |  |
| --- | --- | --- |
| 名前 | 独立した単位として計上されている部分 | エイリアス |
| fflags | ビット 4, 3, 2, 1, 0 | fcsr |
| ｆｒｍ | ＣＳＲ全体 | fcsr |
| fcsr | ビット 7-5, 4, 3, 2, 1, 0 | Fflags, frm |

表14.1.構文依存関係が CSR を通して追跡される粒度

注意: 読み取り専用の CSR は、構文依存関係の定義に参加しないため、リストには記載されていません。

## 14.3 ソースおよびデスティネーションレジスタのリスト

このセクションでは、各命令のソースとデスティネーションレジスタの具体的なリストを提供する。これらのリストは、セクション14.1の構文依存性の定義で使用される。

用語 “accumulating CSR"は、ソースレジスタとデスティネーションレジスタの両方を持ちますが、それ自身からそれ自身への依存関係だけを運ぶCSRを説明するために使用されます。

命令は、注釈を付けている場合を除き、“ソースレジスタ”の各ソースから“デスティネーションレジスタ”の各デスティネーションレジスタへ、“ソースレジスタ”の各ソースから“累積CSR”の各CSRへ、“累積CSR”の各CSRからそれ自身へと依存関係を持っています。

キー：

Ａ アドレスソースレジスタ

Ｄ データソースレジスタ

† この命令は、ソース・レジスタからデスティネーション・レジスタへの依存関係はありません

‡ この命令は、ソース・レジスタからデスティネーション・レジスタへの依存関係を、指定されたとおりに運びます

**RV32I 基本整数命セット**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ソースレジスタ | デスティネーションレジスタ | 蓄積 CSRs |  |
| LUI |  | *rd* |  |  |
| AUIPC |  | *rd* |  |  |
| JAL |  | *rd* |  |  |
| JALR † | *rs1* | *rd* |  |  |
| BEQ | *rs1, rs2* |  |  |  |
| BNE | *rs1, rs2* |  |  |  |
| BLT | *rs1, rs2* |  |  |  |
| BGE | *rs1, rs2* |  |  |  |
| BLTU | *rs1, rs2* |  |  |  |
| BGEU | *rs1, rs2* |  |  |  |
| LB † | *rs1 A* | *rd* |  |  |
| LH † | *rs1 A* | *rd* |  |  |
| LW † | *rs1 A* | *rd* |  |  |
| LBU † | *rs1 A* | *rd* |  |  |
| LHU † | *rs1 A* | *rd* |  |  |
| SB | *rs1 A, rs2 D* |  |  |  |
| SH | *rs1 A, rs2 D* |  |  |  |
| SW | *rs1 A, rs2 D* |  |  |  |
| ADDI | *rs1* | *rd* |  |  |
| SLTI | *rs1* | *rd* |  |  |
| SLTIU | *rs1* | *rd* |  |  |
| XORI | *rs1* | *rd* |  |  |
| ORI | *rs1* | *rd* |  |  |
| ANDI | *rs1* | *rd* |  |  |
| SLLI | *rs1* | *rd* |  |  |
| SRLI | *rs1* | *rd* |  |  |
| SRAI | *rs1* | *rd* |  |  |
| ADD | *rs1, rs2* | *rd* |  |  |
| SUB | *rs1, rs2* | *rd* |  |  |
| SLL | *rs1, rs2* | *rd* |  |  |
| SLT | *rs1, rs2* | *rd* |  |  |
| SLTU | *rs1, rs2* | *rd* |  |  |
| XOR | *rs1, rs2* | *rd* |  |  |
| SRL | *rs1, rs2* | *rd* |  |  |
| SRA | *rs1, rs2* | *rd* |  |  |
| OR | *rs1, rs2* | *rd* |  |  |
| AND | *rs1, rs2* | *rd* |  |  |
| FENCE |  |  |  |  |
| FENCE.I |  |  |  |  |
| ECALL |  |  |  |  |
| EBREAK |  |  |  |  |

**RV32I 基本整数命令セット (続き)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ソースレジスタ | デスティネーションレジスタ | 蓄積 CSRs |  |
| CSRRW ‡ | *rs1, ｃｓｒ \** | *rd, csr* |  | \* unless rd==x0 |
| CSRRS ‡ | *rs1, ｃｓｒ* | *rd \*, csr* |  | \* unless rd==x0 |
| CSRRC ‡ | *rs1, ｃｓｒ* | *rd \*, csr* |  | \* unless rd==x0 |

‡ rs1からcsrへ、csrからrdへの依存関係があります  
  
**RV32I 基本整数命令セット (続き)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ソースレジスタ | デスティネーションレジスタ | 蓄積 CSRs |  |
| CSRRWI ‡ | *ｃｓｒ \** | *rd, csr* |  | \* unless rd==x0 |
| CSRRSI ‡ | *ｃｓｒ* | *rd \*, csｒ \** |  | \* unless uimm[4:0]==0 |
| CSRRCI ‡ | *ｃｓｒ* | *rd \*, csｒ \** |  | \* unless uimm[4:0]==0 |

‡ csrからrdへの依存関係があります  
  
**RV64I 基本整数命令セット**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ソースレジスタ | デスティネーションレジスタ | 蓄積 CSRs |  |
| LWU | *rs1 A* | *rd* |  |  |
| LD | *rs1 A* | *rd* |  |  |
| SD | *rs1 A, rs2 D* |  |  |  |
| SLLI | *rs1* | *rd* |  |  |
| SRLI | *rs1* | *rd* |  |  |
| SRAI | *rs1* | *rd* |  |  |
| ADDIW | *rs1* | *rd* |  |  |
| SLIW | *rs1* | *rd* |  |  |
| SRLIW | *rs1* | *rd* |  |  |
| SRAIW | *rs1* | *rd* |  |  |
| ADDW | *rs1, rs2* | *rd* |  |  |
| SUBW | *rs1, rs2* | *rd* |  |  |
| SLLW | *rs1, rs2* | *rd* |  |  |
| SRLW | *rs1, rs2* | *rd* |  |  |
| SRAW | *rs1, rs2* | *rd* |  |  |

**RV32M標準拡張**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ソースレジスタ | デスティネーションレジスタ | 蓄積 CSRs |  |
| MUL | *rs1, rs2* | *rd* |  |  |
| MULH | *rs1, rs2* | *rd* |  |  |
| MULHSU | *rs1, rs2* | *rd* |  |  |
| MULHU | *rs1, rs2* | *rd* |  |  |
| DIV | *rs1, rs2* | *rd* |  |  |
| DIVU | *rs1, rs2* | *rd* |  |  |
| REM | *rs1, rs2* | *rd* |  |  |
| REMU | *rs1, rs2* | *rd* |  |  |

**RV64M 基本整数命令セット**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ソースレジスタ | デスティネーションレジスタ | 蓄積 CSRs |  |
| MULW | *rs1, rs2* | *rd* |  |  |
| DIVW | *rs1, rs2* | *rd* |  |  |
| DIVUW | *rs1, rs2* | *rd* |  |  |
| REMW | *rs1, rs2* | *rd* |  |  |
| REMUW | *rs1, rs2* | *rd* |  |  |

**RV32A 標準拡張**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ソースレジスタ | デスティネーションレジスタ | 蓄積 CSRs |  |
| LR.W † | *rs1 A* | *rd* |  |  |
| SC.W † | *rs1 A, rs2 D* | *rd* |  | \* 成功時 |
| AMOSWAP.W † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOADD.W † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOXOR.W † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOAND.W † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOOR.W † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOMIN.W † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOMAX.W † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOMINU.W † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOMAXU.W † | *rs1 A, rs2 D* | *rd* |  |  |

**RV64A 標準拡張**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ソースレジスタ | デスティネーションレジスタ | 蓄積 CSRs |  |
| LR.D † | *rs1 A* | *rd* |  |  |
| SC.D † | *rs1 A, rs2 D* | *rd* |  | \* 成功時 |
| AMOSWAP.D † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOADD.D † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOXOR.D † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOAND.D † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOOR.D † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOMIN.D † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOMAX.D † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOMINU.D † | *rs1 A, rs2 D* | *rd* |  |  |
| AMOMAXU.D † | *rs1 A, rs2 D* | *rd* |  |  |

**RV32F 標準拡張**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ソースレジスタ | デスティネーションレジスタ | 蓄積 CSRs |  |
| FLW † | *rs1 A* | *rd* |  |  |
| FSW | *rs1 A, rs2 D* |  |  |  |
| FMADD.S | *rs1, rs2, rs3, frm \** | *rd* | NV, OF, UF, NX | \* rm==111の時 |
| FMSUB.S | *rs1, rs2, rs3, frm \** | *rd* | NV, OF, UF, NX | \* rm==111の時 |
| FNMSUB.S | *rs1, rs2, rs3, frm \** | *rd* | NV, OF, UF, NX | \* rm==111の時 |
| FNMADD.S | *rs1, rs2, rs3, frm \** | *rd* | NV, OF, UF, NX | \* rm==111の時 |
| FADD.S | *rs1, rs2, frm \** | *rd* | NV, OF, UF, NX | \* rm==111の時 |
| FSUB.S | *rs1, rs2, frm \** | *rd* | NV, OF, UF, NX | \* rm==111の時 |
| FMUL.S | *rs1, rs2, frm \** | *rd* | NV, OF, UF, NX | \* rm==111の時 |
| FDIV.S | *rs1, rs2, frm \** | *rd* | NV, DZ, OF, UF, NX | \* rm==111の時 |
| FSQRT.S | *rs1, frm \** | *rd* | NV, NX | \* rm==111の時 |
| FSGNJ.S | *rs1, rs2* | *rd* |  |  |
| FSGNJN.S | *rs1, rs2* | *rd* |  |  |
| FSGNJX.S | *rs1, rs2* | *rd* |  |  |
| FMIN.S | *rs1, rs2* | *rd* | NV |  |
| FMAX.S | *rs1, rs2* | *rd* | NV |  |
| FCVT.W.S | *rs1, frm \** | *rd* | NV, NX | \* rm==111の時 |
| FCVT.WU.S | *rs1, frm \** | *rd* | NV, NX | \* rm==111の時 |
| FMV.X.W | *rs1* | *rd* |  |  |
| FEQ.S | *rs1, rs2* | *rd* | NV |  |
| FLT.S | *rs1, rs2* | *rd* | NV |  |
| FLE.S | *rs1, rs2* | *rd* | NV |  |
| FCLASS.S | *rs1* | *rd* |  |  |
| FCVT.S.W | *rs1, frm \** | *rd* | NX | \* rm==111の時 |
| FCVT.S.WU | *rs1, frm \** | *rd* | NX | \* rm==111の時 |
| FMV.W.X | *rs1* | *rd* |  |  |

**RV64F 標準拡張**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ソースレジスタ | デスティネーションレジスタ | 蓄積 CSRs |  |
| FCVT.L.S | *rs1, frm \** | *rd* | NV, NX | \* rm==111の時 |
| FCVT.LU.S | *rs1, frm \** | *rd* | NV, NX | \* rm==111の時 |
| FCVT.S.L | *rs1, frm \** | *rd* | NX | \* rm==111の時 |
| FCVT.S.LU | *rs1, frm \*D* | *rd* | NX | \* rm==111の時 |

**RV32D 標準拡張**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ソースレジスタ | デスティネーションレジスタ | 蓄積 CSRs |  |
| FLD † | *rs1 A* | *rd* |  |  |
| FSD | *rs1 A, rs2 D* |  |  |  |
| FMADD.D | *rs1, rs2, rs3, frm \** | *rd* | NV, OF, UF, NX | \* rm==111の時 |
| FMSUB.D | *rs1, rs2, rs3, frm \** | *rd* | NV, OF, UF, NX | \* rm==111の時 |
| FNMSUB.D | *rs1, rs2, rs3, frm \** | *rd* | NV, OF, UF, NX | \* rm==111の時 |
| FNMADD.D | *rs1, rs2, rs3, frm \** | *rd* | NV, OF, UF, NX | \* rm==111の時 |
| FADD.D | *rs1, rs2, frm \** | *rd* | NV, OF, UF, NX | \* rm==111の時 |
| FSUB.D | *rs1, rs2, frm \** | *rd* | NV, OF, UF, NX | \* rm==111の時 |
| FMUL.D | *rs1, rs2, frm \** | *rd* | NV, OF, UF, NX | \* rm==111の時 |
| FDIV.D | *rs1, rs2, frm \** | *rd* | NV, DZ, OF, UF, NX | \* rm==111の時 |
| FSQRT.D | *rs1, frm \** | *rd* | NV, NX | \* rm==111の時 |
| FSGNJ.D | *rs1, rs2* | *rd* |  |  |
| FSGNJN.D | *rs1, rs2* | *rd* |  |  |
| FSGNJX.D | *rs1, rs2* | *rd* |  |  |
| FMIN.D | *rs1, rs2* | *rd* | NV |  |
| FMAX.D | *rs1, rs2* | *rd* | NV |  |
| FCVT.S.D | *rs1, frm \** | *rd* | NX | \* rm==111の時 |
| FCVT.D.S | *rs1, frm \** | *rd* | NX | \* rm==111の時 |
| FEQ.D | *rs1, rs2* | *rd* | NV |  |
| FLT.D | *rs1, rs2* | *rd* | NV |  |
| FLE.D | *rs1, rs2* | *rd* | NV |  |
| FCLASS.D | *rs1* | *rd* |  |  |
| FCVT.W.D | *rs1, frm \** | *rd* | NV, NX | \* rm==111の時 |
| FCVT.WU.D | *rs1, frm \** | *rd* | NV, NX | \* rm==111の時 |
| FCVT.D.W | *rs1* | *rd* |  |  |
| FCVT.D.WU | *rs1* | *rd* |  |  |

**RV64D 標準拡張**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ソースレジスタ | デスティネーションレジスタ | 蓄積 CSRs |  |
| FCVT.L.D | *rs1, frm \** | *rd* | NV, NX | \* rm==111の時 |
| FCVT.LU.D | *rs1, frm \** | *rd* | NV, NX | \* rm==111の時 |
| FMV.X.D | *rs1* | *rd* |  |  |
| FCVT.D.L | *rs1, frm \** | *rd* | NX | \* rm==111の時 |
| FCVT.D.LU | *rs1, frm \*D* | *rd* | NX | \* rm==111の時 |
| FMV.D.X | *rs1* | *rd* |  |  |

**第15章**

# “L"浮動小数点10進数用の標準拡張機能、 バージョン0.0

**本章は、財団の批准を受けていない草案です。**

この章はIEEE 754-2008規格で定義されている10進浮動小数点演算をサポートするために設計された標準拡張機能 “L"の仕様のためのプレースホルダです。

## 15.1 浮動小数点10進レジスタ

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への値の移動およびメモリからの値の移動に使用されます。

*融合乗算加算命令で必要とされる大きなオペコード空間のため、10進浮動小数点命令の拡張では、30ビットのエンコーディング空間に5つの25ビットのメジャーオペコードが必要になります。*

**第16章**

# “C"圧縮命令のための標準拡張機能、 バージョン2.0

この章では、RISC-V標準の圧縮命令セット拡張機能「C」の現在の提案について説明します。C拡張はベースとなるISA(RV32, RV64, RV128)に追加することができますが、ここではこれらのISAをカバーするために “RVC"という総称を使用します。通常、プログラム中のRISC-V命令の50%～60%をRVC命令に置き換えることで、25%～30%のコードサイズの削減が可能です。

## 16.1 概要

RVCは、一般的な32ビットRISC-V命令の16ビット版をより短くしたシンプルな圧縮方式を採用しています。

* 即時またはアドレスオフセットが小さいか、
* ゼロレジスタ(x0)、ABIリンクレジスタ(x1)、ABIスタックポインタ(x2)、または
* 宛先レジスタと最初のソースレジスタが同一であるか、
* 使用されるレジスタが最も一般的な8つのレジスタであるかのいずれかです。

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拡張では、16ビット命令と32ビット命令を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、すなわちIALIGN=16から開始することができるようになりました。C拡張が追加されたことで、どの命令でも命令アドレス・ミスアライメント例外を発生させることができなくなりました。

*元の32ビット命令の32ビットアライメント制約を取り除くことで、コード密度を大幅に向上させることができます。*

圧縮命令のエンコーディングはRV32C、RV64C、RV128Cでほぼ共通ですが、表16.4に示すように、いくつかのオペコードはベースISA幅に応じて異なる目的で使用されています。

たとえば、アドレス空間の広いRV64CとRV128Cでは、64ビット整数値のロードとストアを圧縮するために追加のオペコードが必要ですが、RV32Cでは、単精度浮動小数点値のロードとストアを圧縮するために同じオペコードを使用します。同様に、RV128Cでは128ビット整数値のロードとストア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では倍精度浮動小数点値のロードとストアに同じオペコードが使用されます。C 拡張機能が実装されている場合、関連する標準浮動小数点拡張機能 (F または D) が実装されている場合は、適切な圧縮浮動小数点ロード/ストア命令を提供する必要があります。さらに、RV32Cには、短距離のサブルーチン呼び出しを圧縮するための圧縮されたジャンプとリンク命令が含まれており、RV64CとRV128CのADDIWを圧縮するために同じオペコードが使用されています。

*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割合を占めているため、RV32CとRV64Cのエンコーディングに含める動機となっています。*

*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スタティック圧縮またはダイナミック圧縮の重要なソースにはなりませんが、ハードウェアの単精度浮動小数点単位のみを提供し、単精度浮動小数点数のみをサポートするABIを持つマイクロコントローラでは、測定されたベンチマークでは、単精度ロードおよびストアが倍精度ロードおよびストアと同じくらいの頻度で使用されることになります。そのため、RV32Cではこれらを圧縮してサポートすることにしました。*

*短距離のサブルーチン呼び出しは、マイクロコントローラ用の小さなバイナリに多く含まれている可能性が高いため、RV32Cにこれらを含める動機となっています。*

*異なるベース レジスタ幅で異なる目的のためにオペコードを再利用すると、ドキュメントに多少の複雑さが加わりますが、複数のベースのISA レジスタ幅をサポートするデザインの場合でも、インプリメンテーションの複雑さへの影響は小さくなります。圧縮された浮動小数点ロードおよびストアのバリアントは、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持つ同じ命令フォーマットを使用します。*

RVCは、各RVC命令がベースのISA(RV32I/E、RV64I、またはRV128I)またはFおよびD標準エクステンションのいずれかで1つの32ビット命令に拡張されるという制約の下で設計されています。この制約を採用すると 、主に2つの利点があ り ます。

* ハードウェア設計では、デコード中にRVC命令を単純に拡張して検証を簡素化し、既存のマイクロアーキテクチャへ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拡張を意識せず、コードの圧縮はアセンブラとリンカに任せることができますが、圧縮を意識したコンパイラの方が一般的にはより良い結果を得ることができます。

*C言語とベースのIFD命令を1対1で対応させることで、複数の複雑さを軽減することができましたが、C言語の拡張機能でしかサポートされていない命令を追加したり、1つのC言語命令で複数のIFD命令をエンコードできるようにするなど、多少高密度なエンコーディングを行うことで得られる潜在的な利益をはるかに上回ると考えました。*

C 拡張はスタンドアロン ISA として設計されておらず、ベースの ISA と一緒に使用することを意図していることに注意してください。

*コード密度を向上させるために、可変長の命令セットが使われてきました。例えば、1950年代後半に開発されたIBMのStretch[4]は、32ビットと64ビットの命令を持つISAを持ち、32ビットの命令の一部は64ビットの完全な命令を圧縮したものでした。ストレッチ社では、短い命令形式の中でアドレス可能なレジスタを限定するというコンセプトを採用し、インデックスレジスタのいずれか1つのみを参照できる短い分岐命令を採用していました。*

*後のIBM 360アーキテクチャ[3]では、16ビット、32ビット、48ビットの命令フォーマットを持つシンプルな可変長の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 [18] を発表しましたが、これは 15 ビットと 30 ビットの 2 つの命令長を持つレジスタリッチなロードストアアーキテクチャを導入したものです。後の Cray-1 デザインでは、16 ビットと 32 ビットの命令長を持つ、非常に類似した命令フォーマットが使用されました。*

*1980年代の初期のRISC ISAは、すべてコードサイズよりも性能を重視しており、ワークステーション環境には適していましたが、組み込みシステムには適していませんでした。そのため、ARMとMIPSはその後、標準的な32ビット幅の命令ではなく、16ビット幅の命令セットを提供することで、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、出発点からの相対的なコードサイズを約25～30%削減し、80x86よりも大幅に*小さい*コードを実現しました。この結果は、16ビットと32ビットの形式しか提供しないRISC ISAよりも可変長のCISC ISAの方が小さいはずだと直感していたため、一部の人たちを驚かせました。*

*オリジナルのRISC ISAでは、これらの無計画な圧縮命令を含むための十分なオペコードスペースが確保されていなかったため、代わりに完全な新しいISAとして開発されました。これは、コンパイラが別々の圧縮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ます。最初の圧縮RISC ISA拡張（ARM ThumbやMIPS16など）では、固定の16ビット命令サイズのみを使用しており、スタティックコードサイズの削減には優れていましたが、ダイナミック命令数が増加し、元の固定幅32ビット命令サイズと比較して性能が低下しました。このため、16 ビッ ト命令と32 ビッ ト命令を混在させた第2 世代の圧縮RISC ISA デザイン（ARM Thumb2、microMIPS、PowerPC VLE など）が開発され、性能は純粋な32 ビッ ト命令と同等ですが、コードサイズを大幅に削減することができました。残念ながら、これらの異なる世代の圧縮ISAは、互いに互換性がなく、オリジナルの非圧縮ISAとの互換性がないため、ドキュメント、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っています。*

*一般的に使用されている64ビットISAのうち、現在圧縮命令フォーマットをサポートしているのはPowerPCとmicroMIPSだけです。モバイルプラットフォーム向けの最も一般的な64ビットISA（ARM v8）には圧縮命令フォーマットが含まれていませんが、スタティックコードサイズとダイナミック命令フェッチ帯域幅が重要なメトリクスであることを考えると、驚くべきことに、圧縮命令フォーマットが含まれていません。大規模なシステムでは、スタティックコードサイズは大きな問題ではありませんが、命令フェッチの帯域幅は、大規模な命令ワークセットを持つことが多い商用ワークロードを実行するサーバーでは、大きなボトルネックになる可能性があります。*

*RISC-Vは25年の時を経て、最初から圧縮命令をサポートするように設計されており、基本ISAの上に（他の多くの拡張機能と共に）シンプルな拡張機能としてRVCを追加できるだけの十分なオペコードスペースを残しています。RVCの理念は、組み込みアプリケーションのコードサイズを縮小し*、*命令キャッシュのミスが少ないため、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 のフェッチにより命令ビットが 25%%-30%減り、命令キャッシュのミスが 20%-25%減り、または命令キャッシュサイズを 2 倍にした場合とほぼ同じパフォーマンスの影響があることを示しています[22]。*

## 16.2 圧縮された命令形式

表16.1 に、9 つの圧縮命令形式を示します。CR、CI、CSSは32個のRVIレジスタのいずれかを使用できますが、CIW、CL、CS、CA、CBはそのうちの8個だけに制限されています。表16.2に、レジスタx8～x15に対応する、これらの一般的なレジスタの一覧を示します。スタックへの保存とスタックからの復元が非常に一般的なので、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とストア命令の別バージョンがあり、32個のデータレジスタすべてにアクセスできるようにCIとCSSフォーマットを使用することに注意してください。CIWはADDI4SPN命令に8ビットのイミディエイトを供給します。

*RISC-VのABIが変更され、頻繁に使用されるレジスタがレジスタ*x8*--*x15*にマップされるようになりました。これは、連続した自然に整列されたレジスタ番号のセットを持つことで、伸長デコーダを簡素化し、16個の整数レジスタしか持たないRV32EベースのISAとの互換性もあります。*

圧縮レジスタ ベース の浮動小数点 ロード と ストアも それぞれCL とCS 形式を使用し 、8つのレジスタ はf8～f15 にマッピング されます。

*標準的なRISC-Vの呼び出し規約では、最も頻繁に使用される浮動小数点レジスタをレジスタ*f8*から*f15に*マップしており、整数レジスタ番号の場合と同様のレジスタ伸長デコードが可能です。*

このフォーマットは、2つのレジスタ ソース指定子のビットをすべての命令で同じ場所に保持し、デスティネーション レジスタ フィールドが移動できるように設計されています。5 ビッ トの完全なデスティネーション レジスタ指定子が存在する場合は、32 ビッ トのRISC-V エンコーディングと同じ場所にあります。即時フィールドが符号拡張されている場合、符号拡張は常にビット12からです。イミディエイトフィールドは、基本仕様と同様にスクランブル化されており、必要とされるイミディエイトの数を減らすことができます。

*即時フィールドは、命令フォーマットでは逐次的な順序ではなくスクランブルされているため、すべての命令で可能な限り多くのビットが同じ位置にあることになり、それによって実装が簡素化されます。例えば、即時ﾋﾞｯﾄ17--10は常に同じ命令ﾋﾞｯﾄ位置から供給されます。他の5つの即時ビット(5、4、3、1、および0)は2つのソース命令ビットだけを持ち、4つ(9、7、6、および2)は3つのソースを持ち、1つ(8)は4つのソースを持ちます。*

多くの RVC 命令では、ゼロ値の即時値は禁止されており、x0 は有効な 5 ビットのレジスタ指定子ではありません。これらの制限は、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エンコーディングスペースを解放します。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| フォーマット | 意味 | 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | | | | | | |
| CR | レジスタ | funct4 | | rd/rs1 | | rs2 | | op |
| CI | 即値 | funct3 | imm | rd/rs1 | | imm | | op |
| CSS | 相対的スタックストア | funct3 | imm | | | rs2 | | op |
| CIW | ワイド即値 | funct3 | imm | | | | rd’ | op |
| CL | ロード | funct3 | imm | | rs1’ | imm | rd’ | op |
| CS | ストア | funct3 | imm | | rs1’ | imm | rs2’ | op |
| CA | 算術 | funct6 | | | rd’/rs’ | funct2 | rs2’ | op |
| CB | 分岐 | funct3 | offset | | rs1’ | offset | | op |
| CJ | ジャンプ | funct3 | jump target | | | | | op |

表16.1: 圧縮された16bit RVC命令のフォーマット

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RVCレジスタ番号 | 000 | 001 | 010 | 0111 | 100 | 101 | 110 | 111 |
| 整数レジスタ番号 | x8 | x9 | x10 | x11 | x12 | x13 | x14 | x15 |
| 整数レジスタABI名 | s0 | s1 | a0 | a1 | a2 | a3 | a4 | a5 |
| 浮動小数点レジスタ番号 | f8 | f9 | f10 | f11 | f12 | f13 | f14 | f15 |
| 浮動小数点レジスタABI名 | fs0 | fs1 | fa0 | fa1 | fa2 | fa3 | fa4 | fa5 |

表16.2：CIW，CL，CS，CA，CBフォーマットの3ビットのrs1 ′，rs2 ′，rd ′フィールドで指定されるレジスタ

## 16.3 ロードと保存の手順

16ビット命令の到達範囲を広げるために、データ転送命令では、データのサイズをバイト単位でスケーリングしたゼロ拡張イミディエイトを使用します。ワードは×4、ダブルワードは×8、クアッドワードは×16となっています。

RVC はロードとストアの 2 つのバリエーションを提供します。1つは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。もう一つは、8つのベースアドレスレジスタのうちの1つと8つのデータレジスタのうちの1つを参照することができます。

### 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 | オペコード |

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フォーマットを使用しています。

C.LWSP はメモリから 32 ビット値をレジスタ *rd* にロードします。これは、4 でスケーリングされた*ゼロ*拡張オフセットをスタック・ポインタ x2 に加算することで有効なアドレスを計算します。lw rd, offset[7:2](x2) に展開されます。C.LWSP は *rd ≠* x0 の時にのみ有効である; *rd* = x0 のコードポイントは予約されている。

C.LDSPは、メモリから64ビット値をレジスタ*RD*にロードするRV64C/RV128C専用命令です。この命令は、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で有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.LDSP は、*rd ≠* x0 の時にのみ有効です; *rd* = x0 のコードポイントは予約されています。

C.LQSPは、メモリから128ビットの値をレジスタ*RD*にロードするRV128C専用命令です。この命令は、スタック・ポインタ x2 に、16 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で、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.LQSP は *rd ≠* 0 の時にのみ有効である; *rd*=x0 のコードポイントは予約されています。

C.FLWSPは、メモリから単精度浮動小数点値を浮動小数点レジスタ*RD*にロードするRV32FC専用命令です。この命令は、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することで、その有効アドレスを計算します。これは、flw rd, offset[7:2](x2) に展開されます。

C.FLDSPはRV32DC/RV64DCのみの命令で、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。この命令は、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で有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 | オペコード |

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

これらの命令は、CSS形式を使用しています。

C.SWSP はレジスタ *rs2* に 32 ビットの値をメモリに格納します。これは、4 でスケーリングされた*ゼロ*拡張オフセットをスタック・ポインタ x2 に加算することで有効な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専用命令です。この命令は、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で有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 専用命令です。この命令は、スタック・ポインタ x2 に、16 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で、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。*ゼロ*拡張されたオフセットを4でスケーリングしたものをスタック・ポインタx2に加算することで、有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専用命令です。実効アドレスは、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で計算されます。fsd rs2, offset[8:3](x2) に展開されます。

*関数のエントリ/エキジット時のレジスタのセーブ/リストアコードは、スタティックコードサイズのかなりの部分を占めています。RVC のスタックポインタベースの圧縮ロードとストアは、動的命令の帯域幅を削減することで性能を向上させながら、セーブ/リストアのスタティックコードサイズを2分の1に削減するのに効果的です。*

*他のISAで使用されている一般的なメカニズムは、セーブ/リストアコードサイズをさらに削減するためのロードマルチプル命令とストアマルチプル命令です。RISC-Vへの採用を検討しましたが、これらの命令には以下のような欠点があります。*

* *これらの命令は、プロセッサの実装を複雑にする。*
* *仮想メモリシステムでは、一部のデータアクセスは物理メモリに常駐することができ、一部のデータアクセスは物理メモリに常駐しないことができるため、部分的に実行された命令のための新しい再起動メカニズムが必要となります。*
* *RVC の他の命令とは異なり、ロード 複数 と ストア 複数 に相当する IFD はありません。*
* *RVCの他の命令とは異なり、保存と復元が連続して行われるため、コンパイラはこれらの命令を意識して命令を生成し、保存と復元の機会を最大化するような順序でレジスタを割り当てる必要があります。*
* *単純なマイクロアーキテクチャの実装では、複数のロード/ストア命令の周りに他の命令をどのようにスケジューリングするかが制限され、性能低下の原因となります。*
* *直列的なレジスタ割り当ての要求は、CIW、CL、CS、CA、CBフォーマットで選択された特徴的な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のコードを共通のプロローグとエピローグのコードへのサブルーチン呼び出しで置き換えることで、多くの利益をソフトウェアで実現することができます。[23]の５．6で説明。*

*合理的なアーキテクトは異なる結論に至るかもしれませんが、私たちは、複数のロードとストアを省略して、その代わりに、最大のコードサイズの削減を達成するために、ミリコードの保存/復元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とストア

１５ 13 12 10 9 7 6 5 4 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct3 | imm | rs1′ | imm | ｒｄ′ | オペコード |

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 フォーマットを使用します。

C.LW は、メモリから 32 ビットの値をレジスタ *rd‘*にロードします。それは、レジスタ *rs1’*のベースアドレスに、４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。それは、lw rd‘, offset[6:2](rs1’)に展開します。

C.LD は、RV64C/RV128C のみの命令で、メモリから 64 ビットの値をレジスタ *rd ‘*にロードします。レジスタ *rs1* のベースアドレスに、*ゼロ*拡張オフセットを 8 でスケーリングした値を加算して、実効アドレスを計算します。それは、ld rd’, offset[7:3](rs1 ‘)に展開します。.

C.LQ は、メモリから 128 ビットの値をレジスタ *rd ‘*にロードする RV128C 専用命令です。これは、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することにより、実効アドレスを計算します。それは、lq rd ‘, offset[8:4](rs1 ’)に展開します。

C.FLW は、メモリから単精度浮動小数点値を浮動小数点レジスタ *rd* にロードする RV32FC 専用命令です。*ｒｓ１‘*のベース・アドレスに*ゼロ*拡張オフセットを４でスケーリングした値を加算して有効アドレスを計算します。それは、flw rd ’, offset[6:2](rs1 ‘)に展開します。

C.FLDは、RV32DC/RV64DC専用の命令で、メモリから浮動小数点レジスタ*ｒｄ ‘*に倍精度の浮動小数点値をロードします。を加算して有効アドレスを計算します。

*レジスタrs1′のベースアドレスに，8でスケーリングされたゼロ拡張オフセットを加えて，実効アドレスを計算します。*それは、fld rd ‘, offset[7:3](rs1 ’)に展開します。

１５ 13 12 10 9 7 6 5 4 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct3 | imm | rs1′ | imm | ｒs2′ | オペコード |

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ています。

C.SW は、32 ビットの値をレジスタ *rs2 ‘*に記憶します。それは、レジスタ r*s1 ’*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することにより、実効アドレスを計算します。それは sw rs2 ‘, offset[6:2](rs1) に展開します。

C.SD は、RV64C/RV128C 専用命令で、レジスタ *rs2 ‘*の 64 ビット値をメモリに格納します。レジスタ r*s1 ’*のベースアドレスに、*ゼロ拡張オフセット* を 8 でスケーリングした値を加算して、実効アドレスを計算します。それは、sd rs2 ‘, offset[7:3](rs1 ’)に展開します。

C.SQ は、RV128C 専用命令で、レジスタ *rs2‘*の 128 ビット値をメモリに格納します。これは、レジスタ r*s1 ’*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することにより、実効アドレスを計算します。それは sq rs2 ‘, offset[8:4](rs1 ) に展開します。

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、*ゼロ*拡張オフセットを 4 でスケーリングした値を加算して有効アドレスを算出します。それは fsw rs2 ’, offset[6:2](rs1 ‘)に展開します。

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 専用命令です。レジスタ r*s1 ’*のベースアドレスに、*ゼロ*拡張オフセットを 8 でスケーリングした値を加算して有効アドレスを算出します。それは fsd rs2 ‘, offset[7:3](rs1 ’)に展開します。

## 16.4 制御転送命令

RVC は無条件にジャンプ命令と条件分岐命令を提供します。ベースのRVI命令と同様に、すべてのRVC制御転送命令のオフセットは2バイトの倍数です。

15 13 12 2 1 0

|  |  |  |
| --- | --- | --- |
| funct3 | imm | オペコード |

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形式を使用しています。

C.Jは無条件に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。したがって、C.Jは、± 2KiBの範囲をターゲットにすることができます。C.Jはjal x0, offset[11:1]に展開します。

C.JALはRV32C専用命令で、C.Jと同じ動作をしますが、ジャンプ(pc+2)後の命令のアドレスをリンクレジスタx1に書き込みます。C.JAL は jal x1, offset[11:1] に展開します。

15 12 11 7 6 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| funct4 | rs1 | rs2 | オペコード |

4 5 ５ 2

C.JR ｓｒｃ≠0 0 C2

C.JＡＬR ｓｒｃ≠0 0 C2

これらの命令は、CR フォーマットを使用しています。

C.JR(ジャンプレジスタ)はレジスタ*rs1*のアドレスに無条件に制御転送を行います。C.JRは、jalr x0, 0(rs1)に展開します。C.JR は *rs1≠*x0 の時のみ有効で、rs*1*=x0 のコードポイントは予約されています。

C.JALR(ジャンプ と リンクレジスタ)は、C.JRと同じ動作をしますが、ジャンプ(pc+2)に続く命令のアドレスをリンクレジスタx1に追加で書き込みます。C.JALRはjalr x1, 0(rs1)に展開します。C.JALRはrs*1≠*x0の時のみ有効で、rs*1*=x0のコードポイントがC.EBREAK命令に対応します。

*厳密に言えば、C.JALRはリンクアドレスを形成するためにPCに加えられる値がベースISAのように4ではなく2であるため、ベースRVI命令に正確に展開されませんが、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、ベースマイクロアーキテクチャへの非常にマイナーな変更に過ぎません。*

15 13 12 10 9 7 6 2 1 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| funct3 | imm | rs1‘ | imm | オペコード |

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フォーマットを使用しています。

C.BEQZは条件付き制御転送を実行します。オフセットは符号拡張され、分岐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、そのため、±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0であれば分岐します。beq rs1‘ , x0, offset[8:1]に展開します。

C.BNEZも同様に定義されていますが、r*s1 ‘*に0以外の値が含まれている場合に分岐します。それは、bne rs1’, x0, offset[8:1]に展開します。

## 16.5 整数演算命令

RVCは整数演算と定数生成のためのいくつかの命令を提供しています。

### 整数定数生成命令

2つの定数生成命令はどちらもCI命令フォーマットを使用し、任意の整数レジスタをターゲットに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] | オペコード |

3 1 5 ５ 2

C.LI imm[5] dest≠0 imm[4:0] C1

C.LUI nzimm[17] dest≠0 nzimm[16:12] C1

C.LI は符号拡張された 6 ビットのイミディエイト *imm* をレジスタ *rd に*ロードします。C.LI は addi rd, x0, imm[5:0]に展開します。C.LIはrd≠x0の時のみ有効です。rd=x0のコードポイントはHINTsをエンコードしています。

C.LUIは、デスティネーション・レジスタのビット17--12に0以外の6ビットの即時フィールドをロードし、最下位12ビットをクリアし、ビット17をデスティネーションの上位ビットすべてに符号拡張します。C.LUI は lui rd, nzimm[17:12] に展開します。C.LUIはrd={x0, x2}のとき、即物的に0にならないときにのみ有効です。nzimm=0のコードポイントは予約、rd=x0の残りのコードポイントはHINTs、rd=x2の残りのコードポイントはC.ADDI16SP命令に対応しています。

### 整数レジスタ-初歩的な演算

これらの整数レジスタ即値演算はCIフォーマットでエンコードされており、整数レジスタと6ビットの即値に対する演算を実行します。

15 13 12 11 7 6 2 1 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| funct3 | imm[5] | rd/rs1 | imm[4:0] | オペコード |

3 1 5 ５ 2

C.ADDI nzimm[5] dest≠0 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

C.ADDI はレジスタ *rd* の値に非ゼロ符号拡張 6 ビット即時を加算し、その結果を *rd* に書き込みます。C.ADDI は addi rd, rd, nzimm[5:0] に展開します。C.ADDI は *rd が≠*x0 と nzimm≠0 の時だけ有効です。 *rd*=x0 のコードポイントは C.NOP 命令をエンコードし、*nzimm*=0 の残りのコードポイントは HINTs をエンコードします。

C.ADDIW は RV64C/RV128C 専用命令で、同じ計算を実行して 32 ビットの結果を生成し、結果を 64 ビットに拡張します。C.ADDIW は addiw rd, rd, imm[5:0] に展開します。即値は C.ADDIW に対してはゼロであることができ、これは sext.w rd に対応します。C.ADDIWは、rdが≠0の時にのみ有効である；*rd*=x0のコード・ポイントは予約されています。

C.ADDI16SPは、CLUIとオペコードを共有していますが、デスティネーションフィールドがx2となっています。C.ADDI16SPは、非ゼロの符号拡張された6ビットの即値をスタックポインタ(sp=x2)の値に追加します。即値は(-512,496)の範囲で16の倍数を表すようにスケーリングされます。C.ADDI16SPは、プロローグとエピローグのスタックポインタの調整に使用されます。これは addi x2, x2, nzimm[9:4] に展開されます。C.ADDI16SPはnzimm≠0のときのみ有効で、nzimm=0のコードポイントは予約されています。。

*標準的なRISC-Vの呼び出し規約では、スタックポインタ*spは*常に16バイトアラインメントされています。*

15 13 12 5 4 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| funct3 | imm | rd‘ | オペコード |

3 8 3 2

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

C.ADDI4SPN は、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非ゼロ即時を追加し、その結果を rd ‘に書き込む CIW 形式の命令です。この命令はスタック割り当てられた変数へのポインタを生成するために使用され、addi rd ’, x2, nzuimm[9:2]に展開します。C.ADDI4SPN は *nzuimm≠*0 のときのみ有効です。

15 13 12 11 7 6 2 1 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| funct3 | shamt[5] | rd/rs1 | shamt[4:0] | オペコード |

3 1 5 ５ 2

C.SLLI shamt[5] dest≠0 shamt[4:0] C2

C.SLLI は、レジスタ *rd* の値を論理的に左シフトし、その結果を *rd* に書き込む CI 形式の命令です。シフト量は*shamt*フィールドでエンコードされます。RV128Cでは、ゼロのシフト量が64のシフト量をエンコードするために使用されます。C.SLLIは、shamt=0のRV128Cを除き、slli rd, rd, shamt[5:0]に展開し、slli rd, rd, 64に展開します。

RV32Cの場合*、shamt[5]*は0でなければならない。*shamt[5]=1*のコードポイントは、カスタム拡張用に予約されています。RV32CとRV64Cでは、シフト量は0以外でなければなりません。*shamt=0*のコードポイントはHINTsです。すべての基本ISAでは、RV32Cで*shamt[5]=1*のコードポイントを除き、*rd=x0*のコードポイントはHINTsとなります。

１５ 13 12 １１ 10 9 7 6 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct3 | shamt[5] | funct2 | rd‘/rs1’ | shamt[4:0] | オペコード |

3 1 2 3 5 2

C.SRLI shamt[5] C.SRLI dest shamt[4:0] C1

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

C.SRLI は、レジスタ *rd ‘* の値を論理的に右シフトし、その結果を *rd ’*に書き込む CB フォーマットの命令です。シフト量は *shamt* フィールドに符号化されます。RV128Cの場合は、シフト量0を使用して64のシフトを符号化しています。さらに、RV128Cではシフト量は符号拡張されているため、正規シフト量は1～31、64、96～127となります。C.SRLI は srli rd ‘, rd ’, shamt[5:0] に展開しますが、RV128C の shamt=0 の場合は srli rd ‘, rd ’, , 64 に展開します。

RV32Cの場合、sh*amt[5]は*ゼロでなければならず、*shamt[5*]=1のコードポイントはカスタム拡張用に予約されています。RV32CとRV64Cでは、シフト量は0以外でなければならず、*shamt*=0のコードポイントはHINTsです。

C.SRAIは、C.SRLIと類似して定義されていますが、代わりに算術的右シフトを行います。C.SRAIは、srai rd ‘, rd ’, shamt[5:0]に展開します。

*左シフトは通常、アドレス値のスケーリングに頻繁に使用されるため、左シフトは右シフトよりも頻繁に使用されます。そのため、右シフトはエンコーディング領域が狭くなり、他のすべての即時値が符号拡張されるエンコーディング四分位に配置されます。RV128では、6 ビッ トのシフト量の即時値も符号拡張することにしました。デコードの複雑さを減らすことは別として、64-95よりも96-127の右シフト量の方が、128ビットのアドレスポインタの高い部分にあるタグの抽出を可能にするために有用であると考えています。128ビットのアドレス空間コードの典型的な使用法を評価できるように、RV128CはRV32CやRV64Cと同じポイントではフリーズしないことに注意してください。*

１５ 13 12 １１ 10 9 7 6 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct3 | imm[5] | funct2 | rd′/rs1′ | imm[4:0] | オペコード |

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 6 ビット即時のビット単位の AND を計算し、その結果を r*d ’*に書き込む CB 形式の命令です。C.ANDIは、andi rd ‘, rd ’, imm[5:0]に展開します。

### 整数レジスタの操作

15 12 11 7 6 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| funct4 | rd/rs1 | rs2 | オペコード |

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 フォーマットを使用しています。

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 に展開されます。C.MV は *rs2≠*x0 の時のみ有効であり、rs*2*=x0 のコードポイントは C.JR 命令に対応する。*rs2≠*x0 と *rd*=x0 のコードポイントは、HINTs です。

*C.MV は、ADDI を使用する標準的な MV 疑似命令とは異なる命令に展開します。例えば、レジスタリネームハードウェアを使用してMVを特別に扱う実装では、わずかなハードウェアコストの追加で、ADDの代わりにC.MVをMVに展開した方が便利だと感じるかもしれません。*

C.ADD はレジスタ *rd*, *rs2 の*値を加算し、その結果をレジスタ *rd* に書き込みます。C.ADD は add rd, rd, rs2 に展開します。C.ADD は *rs2≠*x0 の時のみ有効で、rs*2*=x0 のコードポイントは C.JALR 命令と C.EBREAK 命令に対応しています。*rs2≠*x0 と *rd*=x0 のコードポイントは HINTs です。

15 10 9 7 6 ５ 4 2 1 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| funct6 | rd‘/rs1’ | funct2 | rs2‘ | オペコード |

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.ADDW dest C.ADDW src C1

C.SUBW dest C.SUBW src C1

これらの命令はCAフォーマットを使用しています。

C.AND は、レジスタ *rd ’ とrs2 ’*のビット毎の 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 and rd ’, rd ‘, rs2 ’に展開します。

C.OR は、レジスタ *rd ‘* と *rs2 ’* のビット毎の OR を計算し、その結果をレジスタ *rd ‘* に書き込みます。C.OR は or rd ’, rd ‘, rs2 ’に展開します。

C.XOR は、レジスタ *rd ‘* と *rs2 ’* のビット毎の XOR を計算し、その結果をレジスタ *rd ‘* に書き込みます。C.XORは、xor rd ’, rd ‘, rs2 ’ に展開します。

C.SUB は、レジスタ *rs1‘ 値から rs2 ’* を減算し、その結果をレジスタ *rd* ‘ に書き込みます。C.SUB は sub rd ’, rd ‘, rs2 ’に展開します。

C.ADDW は、RV64C/RV128C のみの命令で、レジスタ *rd ‘*と *rs2 ’* の値を加算し、加算結果をレジスタ *rd‘* に書き込む前に、和の下位 32 ビットを符号拡張します。C.ADDWは addw rd ’, rd ‘, rs2 ’ に展開します。

C.SUBW は、RV64C/RV128C のみの命令で、レジスタ *rs2‘ を*レジスタ *rd ’*から減算し、その結果をレジスタ *rd ‘* に書き込む前に、その差の下位 32 ビットを符号拡張します。C.SUBW は subw rd ’, rd ‘, rs2 ’ に展開します。

*この 6 つの命令群は、個別には大きな節約にはなりませんが、エンコーディング スペースをあまり占有せず、実装が簡単で、グループとして静的圧縮と動的圧縮の価値ある改善を提供します。*

### 定義された不正な命令

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の16ビット命令は、不正な命令として永久に予約されます。

*我々は、メモリ空間のゼロ化された部分や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を助けるために、オールゼロ命令を不正な命令とすることを留保します。オールゼロ値は、非標準的な拡張では再定義されるべきではありません。*

*同様に、全てのビットが1に設定されている命令（RISC-Vの可変長エンコーディング方式の非常に長い命令に対応）は、非存在メモリ領域で見られるもう一つの共通の値を捕捉するための違法な命令として確保しています。*

### NOP命令

15 13 12 11 7 6 2 1 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| funct3 | imm[5] | rd/rs1 | imm[4:0] | オペコード |

3 1 5 ５ 2

C.NOP 0 0 0 C1

C.NOP は CI 形式の命令で、PC を前進させ、該当するパフォーマンス カウンタをインクリメントする以外は、ユーザから見える状態を一切変更しません。C.NOPはnopに展開されます。C.NOPは*imm*=0の時にのみ有効です。*imm≠*0のコードポイントはHINTsをエンコードします。

### ブレークポイント命令

15 12 11 2 1 0

|  |  |  |
| --- | --- | --- |
| funct4 | 0 | オペコード |

4 １0 2

C.EBREAK 0 C2

デバッガはC.EBREAK命令を使用することができます。これはebreakに展開され、制御をデバッグ環境に戻すために使用されます。C.EBREAKはC.ADD命令とオペコードを共有していますが、*rd*とrs*2の*両方がゼロであるため、CRフォーマットを使用することもできます。

## 16.6 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、第8.3節で述べたように、制約付きLR/SCシーケンス内で許可されるI命令の圧縮形式が、制約付きLR/SCシーケンス内でも許可されます。

*この意味するところは、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成することを保証しなければならないということです。*

## 16.7 HINT命令

RVC エンコーディング スペースの一部はマイクロアーキテクトHINT 用に予約されています。RV32I ベースのISA(セクション2.9 参照) のHINT と同様に、これらの命令は、PC と適用可能なパフォーマ ンス カウンターを進めることを除いて、アーキテクト状態を変更しません。HINT は、無視する イ ンプ リ メ ン テーシ ョ ンではno-op として実行 さ れます。

RVC HINTは、*rd*=x0（例：C.ADD *x0*, *t0*）、または*rdが*それ自身のコピーで上書きされる（例：C.ADDI *t0*, 0）ため、アーキテクチャ状態を変更しない計算命令としてエンコードされています。

*このHINTエンコーディングは、シンプルな実装ではHINTを完全に無視し、代わりにアーキテクチャの状態を変異させない通常の計算命令としてHINTを実行できるように選択されています。*

RVC HINTは必ずしもRVI HINTに展開されるとは限りません。例えば、C.ADD *x0*, *t0は、*ADD *x0*, *x0*, *t0*と同じHINTをエンコードしないかもしれません。

*RVC HINTがRVI HINTに展開することを要求しない主な理由は、HINT が基礎となる計算命令と同じ方法で圧縮可能である可能性が低いからです。また、RVC とRVI のHINT マッピングをデカップリングすることで、不足しているRVC HINT 空間を最もポピュラーなHINT、特にマクロオプ融合に適したHINT に割り当てることができます。*

表16.3にすべてのRVC HINTコードポイントを示します。RV32Cでは、HINT空間の78%が標準HINT用に予約されていますが、現在は何も定義されていません。残りのHINT空間はカスタムHINT用に予約されており、このサブスペースでは標準HINTは定義されません。

|  |  |  |  |
| --- | --- | --- | --- |
| 命令 | 制約 | コードポイント | 目的 |
| C.NOP | *nzimm*≠0 | 63 | *将来の標準的な使用のために予約* |
| C.ADDI | *rd*≠x0, *nzimm*=0 | 31 |
| C.LI | *rd*=x0 | 64 |
| C.LUI | *rd*=x0, *nzimm*≠0 | 63 |
| C.MV | *rd*=x0, *rs2*≠x0 | 31 |
| C.ADD | *rd*=x0, *rs2*≠x0 | 31 |
| C.SLLI | *rd*=x0, *nzimm*≠0 | 31(RV32)  63(RV64/128) | *将来の標準的な使用のために予約* |
| C.SLLI64 | *rd*=x0 | 1 |
| C.SLLI64 | *rd*≠x0, RV32 and RV64 only | 31 |
| C.SRLI64 | RV32 and RV64 only | 8 |
| C.SRAI64 | RV32 and RV64 only | 8 |

表 16.3: RVC HINT 命令。

## 16.8 RVCインストラクションセット一覧

表 16.4 に RVC の主要なオペコードのマップを示す。表の各行は、エンコーディング空間の1象限に対応しています。最後の象限は、最下位2ビットが設定されており、ベースISAのものも含めて16ビットよりも広い命令に対応しています。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。無効な場合は、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*、オペコードがカスタム拡張のために予約されていることを示す*NSE*、オペコードが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*HINTの*いずれかにマークされます(セクション16.7を参照してください)。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| inst[15:13] | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 | ＲＶ３２  ＲＶ６４  RV128 |
| inst[1:0] |
| 00 | ＡＤＤＩ４ＳＰＮ | FLD  FLD  LQ | ＬＷ | FLW  LD  LD | *予約済み* | FSD  FSD  SQ | SW | ＦＳＷ  SD  SD |
| 01 | ADDI | JAL ADDIW  ADDIW | LI | LUI/ADDI16SP | MISC-ALU | J | BEQZ | BNEZ | ＲＶ３２  ＲＶ６４  RV128 |
| 10 | ＳＬＬＩ | ＦＬＤＳＰ  ＦＬＤＳＰ  LQSP | ＬＷＳＰ | FLWSP  LDSP  LDSP | J[AL]R/MV/ADD | FSDSP  FSDSP  SQSP | SWSP | FSWSP  ＳＤＳＰ  SDSP | ＲＶ３２  ＲＶ６４  RV128 |
| 11 |  |  |  | *> 16b* | |  |  |  |  |

表 16.4.RVCオペコードマップ

表 16.5--16.7 に RVC の命令を示します。

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 000 | 0 | | | 0 | 00 |
| 000 | nzuimm[5:4| 9:6| 2| 3] | | | ｒｄ‘ | 00 |
| 001 | uimm[5:3] | rs1‘ | uimm[7:6] | ｒｄ‘ | 00 |
| 001 | uimm[5:4| 8] | rs1‘ | uimm[7:6] | ｒｄ‘ | 00 |
| 010 | uimm[5:3] | rs1‘ | uimm[2| 6] | ｒｄ‘ | 00 |
| 011 | uimm[5:3] | rs1‘ | uimm[2| 6] | ｒｄ‘ | 00 |
| 011 | uimm[5:3] | rs1‘ | uimm[7:6] | ｒｄ‘ | 00 |
| 100 | --- | | |  | 00 |
| 101 | uimm[5:3] | rs1‘ | uimm[7:6] | rs2‘ | 00 |
| 101 | uimm[5:4| 8] | rs1‘ | uimm[7:6] | rs2‘ | 00 |
| 110 | uimm[5:3] | rs1‘ | uimm[2| 6] | rs2‘ | 00 |
| 111 | uimm[5:3] | rs1‘ | uimm[2| 6] | rs2‘ | 00 |
| 111 | uimm[5:3] | rs1‘ | uimm[7:6] | rs2‘ | 00 |

*不正な命令*

C.ADDI4SPN*(RES,nzuimm=0)*

C.FLD*（RV32/64）*

C.LQ*（RV128）*

C.LW

C.FLW*（RV32）*

C.LD*（RV64/128）*

*予約済み*

C.FSD *(RV32/64)*

C.SQ*（RV128）*

C.SW

C.FSW *(RV32)* C.SD(RV64/128)

表 16.5: RVC, 象限 0 の命令リスト。

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 000 | nzimm[5] | 0 | | nzimm[4:0] | | 01 |
| 000 | nzimm[5] | rs1/rd≠0 | | nzimm[4:0] | | 01 |
| 001 | imm[11| 4| 9:8| 10| 6| 7| 3:1| 5] | | | | | 01 |
| 001 | imm[5] | rs1/rd≠0 | | imm[4:0] | | 01 |
| 010 | imm[5] | rd≠0 | | imm[4:0] | | 01 |
| 011 | nzimm[9] | 2 | | nzimm[4|6|8:7|5] | | 01 |
| 011 | nzimm[17] | rd≠{0*,*2} | | nzimm[16:12] | | 01 |
| 100 | nzuimm[5] | 00 | rs1‘/rd’ | nzuimm[4:0] | | 01 |
| 100 | 0 | 00 | rs1‘/rd’ | 0 | | 01 |
| 100 | nzuimm[5] | 01 | rs1‘/rd’ | nzuimm[4:0] | | 01 |
| 100 | 0 | 01 | rs1‘/rd’ | 0 | | 01 |
| 100 | imm[5] | 10 | rs1‘/rd’ | imm[4:0] | | 01 |
| 100 | 0 | 11 | rs1‘/rd’ | 00 | rs2‘ | 01 |
| 100 | 0 | 11 | rs1‘/rd’ | 01 | rs2‘ | 01 |
| 100 | 0 | 11 | rs1‘/rd’ | 10 | rs2‘ | 01 |
| 100 | 0 | 11 | rs1‘/rd’ | 11 | rs2‘ | 01 |
| 100 | 1 | 11 | rs1‘/rd’ | 00 | rs2‘ | 01 |
| 100 | 1 | 11 | rs1‘/rd’ | 01 | rs2‘ | 01 |
| 100 | 1 | 11 | --- | 10 | --- | 01 |
| 100 | 1 | 11 | --- | 11 | --- | 01 |
| 101 | imm[11| 4| 9:8| 10| 6| 7| 3:1| 5] | | | | | 01 |
| 110 | imm[8| 4:3] | | rs1‘/rd’ | imm[7:6| 2:1| 5] | | 01 |
| 111 | imm[8| 4:3] | | rs1‘/rd’ | imm[7:6| 2:1| 5] | | 01 |

C.NOP *(HINT, nzimm≠0)*

C.ADDI *(HINT, nzimm=0)*

C.JAL*（RV32）*

C.ADDIW *(RV64/128; RES,rd=0)*

C.LI *(HINT, rd=0)*

C.ADDI16SP *(RES, nzimm=0)*

C.LUI *(RES,nzimm=0;HINT,rd=0)*

C.SRLI*（RV32 NSE、nzuimm[5]=1*

C.SRLI64 *(RV128; RV32/64 HINT)*

C.SRAI*（RV32 NSE、nzuimm[5]=1*

C.SRAI64 *(RV128; RV32/64 HINT)*

C.ANDI

C.SUB

Ｃ.ＸＯＲ

Ｃ.ＯＲ

C.AND

C.SUBW*（RV64/128；RV32 RES）*

Ｃ.ＡＤＤＷ*（ＲＶ６４／１２８；ＲＶ３２ ＲＥＳ）*

*予約済み*

*予約済み*

C.J.

C.BEQZ

C.BNEZ

表 16.6: RVC, 象限 1 の命令リスト。

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 |
| 000 | 0 | rs1/rd≠0 | 0 | 10 |
| 001 | uimm[5] | rd | uimm[4:3| 8:6] | 10 |
| 001 | uimm[5] | rd≠0 | uimm[4| 9:6] | 10 |
| 010 | uimm[5] | rd≠0 | uimm[4:2| 7:6] | 10 |
| 011 | uimm[5] | rd | uimm[4:2| 7:6] | 10 |
| 011 | uimm[5] | rd≠0 | uimm[4:3| 8:6] | 10 |
| 100 | 0 | rs1≠0 | 0 | 10 |
| 100 | 0 | rd≠0 | rs2≠0 | 10 |
| 100 | 1 | 0 | 0 | 10 |
| 100 | 1 | rs1≠0 | 0 | 10 |
| 100 | 1 | rs1/rd≠0 | rs2≠0 | 10 |
| 101 | uimm[5:3| 8:6] | | rs２ | 10 |
| 101 | uimm[5:4| 9:6] | | rs２ | 10 |
| 110 | uimm[5:2| 7:6] | | rs２ | 10 |
| 111 | uimm[5:2| 7:6] | | rs２ | 10 |
| 111 | uimm[5:3| 8:6] | | rs２ | 10 |

C.SLLI *(HINT,rd=0; RV32 NSE, nzuimm[5]=1)*

C.SLLI64 *(RV128;RV32/64 HINT;HINT, rd=0)*

C.FLDSP *(RV32/64)*

C.LQSP *(RV128; RES, rd=0)*

C.LWSP *(RES, rd=0)*

C.FLWSP*（RV32）*

C.LDSP *(RV64/128; RES, rd=0)*

C.JR *(RES, rs1=0)*

C.MV *(HINT, rd=0)*

C.EBREAK

C.JALR

C.ADD *(HINT, rd=0)*

C.FSDSP *(RV32/64)*

C.SQSP*（RV128）*

C.SWSP

C.FSWSP *(RV32)*

C.SDSP *(RV64/128)*

表16.7.RVC、象限2の命令リスト。

**第17章**

# “B"ビット操作のための標準拡張機能、 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、回転、ファンネルシフト、ビットおよびバイトの置換などのビット操作命令を提供するための将来の標準拡張のための代用物です。

*ビ ッ ト 操作命令は、 一部のアプリケーション 領域では非常に有効ですが、 特に外部にパック さ れたデー タ 構造を扱う場合には非常に有効ですが、 すべての 領域で有用ではな く 、 必要なオペ ラ ン ド をすべて供給す る ために追加の複雑さや命令形式を追加す る 可能性があ る ため、 ベースISA から除外しました。*

*B拡張は、ベースとなる30ビットの命令空間内でのブラウンフィールドエンコーディングになると予想しています。*

**第18章**

# “J"動的翻訳された言語のための 標準拡張機能、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ための将来の標準拡張機能の代用物です。

*多くの一般的な言語は、通常、Java や Javascript などの動的翻訳を介して実装されています。これらの言語は、動的チェックやガベージコレクションのためのISAサポートを追加することで恩恵を受けることができます。*

**第19章**

# “T"トランザクションメモリの標準拡張、 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代用物です。

*過去20年以上にわたって多くの研究が行われ、最初の商用実装が行われたにもかかわらず、複数のアドレスを含むアトミックな操作をサポートする最良の方法については、まだ多くの議論があります。*

*私たちの現在の考えは、オリジナルのトランザクションメモリの提案に沿って、小さな限られた容量のトランザクションメモリバッファを含めることです。*

**第20章**

# “P"パックドSIMD命令の標準拡張、 バージョン0.2

*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演算のためのV拡張を標準化するために、浮動小数点レジスタのためのこのpacked-SIMDの提案をやめようという考えが示されました。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固定小数点演算には関心がありました。タスクグループが新しいP拡張を定義する作業を行っています。*

**第21章**

# “V"ベクトル演算のための標準拡張、 バージョン0.7

現在の作業部会草案は https://github.com/riscv/riscv-v-spec で公開されています。

*基本ベクタ拡張は、32 ビット命令エンコーディング空間内でのデータ並列実行の一般的なサポートを提供することを目的としており、それ以降のベクタ拡張では特定の領域のためのより豊かな機能をサポートします。*

**第22章**

# “Zam" ミズアラインド・アトミックのための 標準拡張機能、v0.1

この章では “Zam"拡張機能を定義します。この拡張機能は ”A"拡張機能を拡張したもので、アトミックメモリ操作(AMO)のサポートを標準化したものです。“Zam"を実装したプラットフォーム上では、不整合アトミックメモリ操作AMOは同じアドレスで同じサイズの他のアクセス(アトミックではないロードやストアを含む)に対してアトミックに実行すればよいだけです。より正確には、 ”Zam"を実装した実行環境は以下の公理に従うことになります。

アト**ミックのミスアラインメントに対するアトミック性公理** *r* と *w が同じ*アドレスで同じサイズの*ハート*からのロード命令とストア命令のペアのミスアラインメントである場合、同じアドレスで *r* と *w と同じ*サイズの*ハート以外のハート*からのストア命令 *s は*存在しないので、グローバルメモリオーダーで *r* と *w によって生成され*たメモリ操作の間に *s* によって生成されたストア操作が存在することはありえません。さらに、広域メモリオーダーにおいて、*r*または*w*によって生成された2つのメモリ操作の間に、*l*によって生成されたロード操作が存在するような、同じアドレスで*r*と*wと同じ*サイズの*hart*以外のhartからのロード命令*lは*存在しないことができます。

この制限されたアトミック性の形式は、 アトミックのずれをサポートする必要があるアプリケーションのニーズと、 実際に必要なアトミック性を提供する実装の能力のバランスをとることを目的としています。

“Zam"の下での整列命令は、RVWMOの下では通常通りに動作します。

*“Zam"の意図は、2つの方法のうちの1つを実装することができるということです。*

1. *問題のアドレスとサイズへのアトミックな不整合アクセスをネイティブにサポートしているハードウェアでは（例えば、単一のキャッシュライン内での不整合アクセスのために）、単に整列 AMO に適用されるのと同じ規則に従うだけでよいのです。*
2. *問題のアドレスとサイズへの誤ったアクセスをネイティブにサポートしていないハードウェア上では、そのアドレスとサイズのすべての命令(ロードを含む)をトラッピングし、与えられたメモリアドレスとアクセスサイズの関数であるミューテックス内でそれらを(任意の数のメモリ操作を介して)実行します。AMO はロードとストアの別々の操作に分割してエミュレートすることができますが、保存されているすべてのプログラム順序規則（例えば、入力と出力の構文依存性）は、AMO がまだ単一のメモリ操作であるかのように動作しなければなりません。*

**第23章**

# “Ztso" トータル ストア オーダリングの ための標準拡張機能, v0.1

この章では、RISC-Vトータルストアオーダリング(RVTSO)メモリ整合性モデルのための “Ztso"拡張子を定義します。RVTSOは14.1章で定義されているRVWMOからのデルタとして定義されています。

*Ztso 拡張は、もともと x86 やSPARC アーキテクチャ用に書かれたコードの移植を容易にすることを目的としており、どちらもデフォルトで TSO を使用しています。また、RVTSO の動作を本質的に提供し、その事実をソフトウェアに公開したいと考えている実装もサポートしています。*

RVTSOは、RVWMOに対して以下の調整を行っています。：

* すべてのロード操作は、あたかも acquire-RCpc アノテーションを持っているかのように動作します。
* すべてのストア操作は、release-RCpcアノテーションがあるかのように動作します。
* すべてのAMOは、Acquisition-RCscとrelease-RCscの両方のアノテーションを持っているかのように振る舞っています。

*これらの規則は4～7を除く全てのPPO規則を冗長化します。また、PWとSRの両方がセットされていないI/O以外のフェンスはすべて冗長になります。最後に、これらは、AMOを過ぎても、どちらの方向にもメモリ操作が並び替えられることはないということも意味しています。*

*RVTSOのコンテキストでは、RVWMOの場合と同様に、ストレージ順序のアノテーションはPPO規則5～7によって簡潔かつ完全に定義されています。これらのメモリモデルでは、ロードバリューの公理により、ハートがストアバッファから後続の（プログラム順の）ロードに値を転送することができます。つまり、ストアは他のハートに見える前にローカルに転送することができるのです。*

ZtsoがISAに新しい命令を追加しないという事実にもかかわらず、RVTSOを想定して書かれたコードは、Ztsoをサポートしていない実装では正しく動作しません。Ztso でのみ動作するようにコンパイルされたバイナリは、バイナリ内にフラグを立ててその旨を示すべきであり、Ztso を実装していないプラットフォームでは動作を拒否することができます。

**第24章**

# RV32/64G取扱説明書セット一覧

RISC-Vプロジェクトの目標の一つは、RISC-Vを安定したソフトウェア開発ターゲットとして使用することである。この目的のために、ベースとなるISA(RV32IまたはRV64I)と選択された標準拡張機能(IMAFD、Zicsr、Zifencei)の組み合わせを「汎用」ISAと定義し、IMAFDZicsr Zifenceiの命令セット拡張機能の組み合わせを「G」と呼ぶことにしました。この章では、RV32GとRV64Gのオペコードマップと命令セットリストを示します。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| inst[4:2] | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111  *(> 32b*)  48*b*  *64b*  48*b* |
| inst[6:5] |
| 00 | LOAD | LOAD-FP | *custom-0* | ＭＩＳＣ-ＭＥＭ | OP-IMM | AUIPC | OP-IMM-32 |
| 01 | STORE | STORE-FP | *custom-1* | AMO | OP | LUI | OP-３２ |
| 10 | MADD | MSUB | NMSUB | NMADD | ＯＰ-ＦＰ | *予約* | *custom-2/rv128* |
| 11 | BRANCH | JALR | *予約* | JAL | SYSTEM | *予約* | *custom-3/rv128* | ≧80b |

表 24.1.RISC-V ベースオペコードマップ、inst[1:0]=11

表24.1にRVGの主要オペコードのマップを示します。下位ビットが3ビット以上設定されている主要オペコードは、32ビット以上の命令長のために予約されています。*予約済み*とマークされたオペコードは、将来の標準拡張で使用される可能性があるため、カスタム命令セット拡張では使用しないでください。*custom-0*およびcustom*-1*とマークされた主要なオペコードは、将来の標準拡張では使用されず、基本32ビット命令フォーマット内でのカスタム命令セット拡張での使用が推奨されます。*custom-2/rv128*とcustom*-3/rv128のオペコード*は、将来のRV128で使用するために予約されていますが、それ以外の場合は標準拡張では使用されないため、RV32とRV64のカスタム命令セット拡張でも使用できます。

RV32G およびRV64G は、幅広い汎用コンピューティングに対応するシンプルながらも完全な命令セットを提供するものと考えています。第16章で説明したオプションの圧縮命令セットを追加することで（RV32GCとRV64GCを形成）、ハードウェアの複雑さは増しますが、性能、コードサイズ、エネルギー効率を向上させることができます。

IMAFDC を超えてさらに命令セットを拡張していくと、追加された命令はよりドメインに特化したものとなり、マルチメディアやセキュリティなどの限定されたクラスのアプリケーションにのみメリットを提供する傾向があります。多くの商用 ISA とは異なり、RISC-V ISA の設計では、ベースとなる ISA と広く適用可能な標準拡張と、これらのより専門的な追加命令とが明確に分離されています。第26章では、RISC-V ISA に拡張機能を追加する方法について、より広範な議論がなされています。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct7 | rs２ | rs１ | funct3 | rd | オペコード | R型 |
| imm[11:0] | | rs１ | funct3 | rd | オペコード | I型 |
| imm[11:5] | rs２ | rs１ | funct3 | imm[4:0] | オペコード | S型 |
| imm[12|10:5] | rs２ | rs１ | funct3 | imm[4:1|11] | オペコード | B型 |
| imm[31:12] | | | | rd | オペコード | U型 |
| imm[20|10:1|11|19:12] | | | | rd | オペコード | J型 |

**RV32I 基本命令セット**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| imm[31:12] | | | | | | rd | 0110111 | LUI |
| imm[31:12] | | | | | | rd | 0010111 | AUIPC |
| imm[20|10:1|11|19:12] | | | | | | rd | 1101111 | JAL |
| imm[11:0] | | | | rs１ | 000 | rd | 1100111 | JALR |
| imm[12|10:5] | | rs２ | | rs１ | 000 | imm[4:1|11] | 1100011 | BEQ |
| imm[12|10:5] | | rs２ | | rs１ | 001 | imm[4:1|11] | 1100011 | BNE |
| imm[12|10:5] | | rs２ | | rs１ | 100 | imm[4:1|11] | 1100011 | BLT |
| imm[12|10:5] | | rs２ | | rs１ | 101 | imm[4:1|11] | 1100011 | BGE |
| imm[12|10:5] | | rs２ | | rs１ | 110 | imm[4:1|11] | 1100011 | BLTU |
| imm[12|10:5] | | rs２ | | rs１ | 111 | imm[4:1|11] | 1100011 | BGEU |
| imm[11:0] | | | | rs１ | 000 | rd | 0000011 | LB |
| imm[11:0] | | | | rs１ | 001 | rd | 0000011 | LH |
| imm[11:0] | | | | rs１ | 010 | rd | 0000011 | LW |
| imm[11:0] | | | | rs１ | 100 | rd | 0000011 | LBU |
| imm[11:0] | | | | rs１ | 101 | rd | 0000011 | LHU |
| imm[11:5] | | rs２ | | rs１ | 000 | imm[4:0] | 0100011 | SB |
| imm[11:5] | | rs２ | | rs１ | 001 | imm[4:0] | 0100011 | SH |
| imm[11:5] | | rs２ | | rs１ | 010 | imm[4:0] | 0100011 | SW |
| imm[11:0] | | | | rs１ | 000 | rd | 0010011 | ADDI |
| imm[11:0] | | | | rs１ | 010 | rd | 0010011 | SLTI |
| imm[11:0] | | | | rs１ | 011 | rd | 0010011 | SLTIU |
| imm[11:0] | | | | rs１ | 100 | rd | 0010011 | XORI |
| imm[11:0] | | | | rs１ | 110 | rd | 0010011 | ORI |
| imm[11:0] | | | | rs１ | 111 | rd | 0010011 | ANDI |
| 0000000 | | shamt | | rs１ | 001 | rd | 0010011 | SLLI |
| 0000000 | | shamt | | rs１ | 101 | rd | 0010011 | SRLI |
| 0100000 | | shamt | | rs１ | 101 | rd | 0010011 | SRAI |
| 0000000 | | rs２ | | rs１ | 000 | rd | 0110011 | ADD |
| 0100000 | | rs２ | | rs１ | 000 | rd | 0110011 | SUB |
| 0000000 | | rs２ | | rs１ | 001 | rd | 0110011 | SLL |
| 0000000 | | rs２ | | rs１ | 010 | rd | 0110011 | SLT |
| 0000000 | | rs２ | | rs１ | 011 | rd | 0110011 | SLTU |
| 0000000 | | rs２ | | rs１ | 100 | rd | 0110011 | XOR |
| 0000000 | | rs２ | | rs１ | 101 | rd | 0110011 | SRL |
| 0100000 | | rs２ | | rs１ | 101 | rd | 0110011 | SRA |
| 0000000 | | rs２ | | rs１ | 110 | rd | 0110011 | OR |
| 0000000 | | rs２ | | rs１ | 111 | rd | 0110011 | AND |
| fm | pred | | suc | rs１ | 000 | rd | 0001111 | FENCE |
| 000000000000 | | | | 00000 | 000 | 00000 | 1110011 | ECALL |
| 000000000001 | | | | 00000 | 000 | 00000 | 1110011 | EBREAK |

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct7 |  | rs２ | rs１ | funct3 | rd | オペコード | R型 |
| imm[11:0] | |  | rs１ | funct3 | rd | オペコード | I型 |
| imm[11:5] |  | rs２ | rs１ | funct3 | imm[4:0] | オペコード | S型 |

**RV64I 基本命令セット（RV32Iに加えて）**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| imm[11:0] | | |  | rs１ | 110 | rd | 0000011 | LWU |
| imm[11:0] | | |  | rs１ | 011 | rd | 0000011 | LD |
| imm[11:5] | |  | rs２ | rs１ | 011 | imm[4:0] | 0100011 | SD |
| 000000 |  | | shamt | rs１ | 001 | rd | 0010011 | SLLI |
| 000000 |  | | shamt | rs１ | 101 | rd | 0010011 | SRLI |
| 010000 |  | | shamt | rs１ | 101 | rd | 0010011 | SRAI |
| imm[11:0] | | |  | rs１ | 000 | rd | 0011011 | ADDIW |
| 0000000 | |  | shamt | rs１ | 001 | rd | 0011011 | SLLIW |
| 0000000 | |  | shamt | rs１ | 101 | rd | 0011011 | SRLIW |
| 0100000 | |  | shamt | rs１ | 101 | rd | 0011011 | SRAIW |
| 0000000 | |  | rs２ | rs１ | 000 | rd | 0111011 | ADDW |
| 0100000 | |  | rs２ | rs１ | 000 | rd | 0111011 | SUBW |
| 0000000 | |  | rs２ | rs１ | 001 | rd | 0111011 | SLLW |
| 0000000 | |  | rs２ | rs１ | 101 | rd | 0111011 | SRLW |
| 0100000 | |  | rs２ | rs１ | 101 | rd | 0111011 | SRAW |

**RV32/RV64 *Zifencei* 標準拡張**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:0] | rs１ | 001 | rd | 0001111 | FENCE.I |

**RV32/RV64 *Zicsr*****標準拡張**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| csr | rs１ | 001 | rd | 1110011 | CSRRW |
| csr | rs１ | 010 | rd | 1110011 | CSRRS |
| csr | rs１ | 011 | rd | 1110011 | CSRRC |
| csr | uimm | 101 | rd | 1110011 | CSRRWI |
| csr | uimm | 110 | rd | 1110011 | CSRRSI |
| csr | uimm | 111 | rd | 1110011 | CSRRCI |

**RV32M標準拡張**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0000001 | rs２ | rs１ | 000 | rd | 0110011 | MUL |
| 0000001 | rs２ | rs１ | 001 | rd | 0110011 | MULH |
| 0000001 | rs２ | rs１ | 010 | rd | 0110011 | MULHSU |
| 0000001 | rs２ | rs１ | 011 | rd | 0110011 | MULHU |
| 0000001 | rs２ | rs１ | 100 | rd | 0110011 | DIV |
| 0000001 | rs２ | rs１ | 101 | rd | 0110011 | DIVU |
| 0000001 | rs２ | rs１ | 110 | rd | 0110011 | REM |
| 0000001 | rs２ | rs１ | 111 | rd | 0110011 | REMU |

**RV64M標準拡張(RV32Mに加えてRV64M)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0000001 | ＲＳ２ | ＲＳ１ | 000 | rd | 0111011 | MULW |
| 0000001 | ＲＳ２ | ＲＳ１ | 100 | rd | 0111011 | DIVW |
| 0000001 | ＲＳ２ | ＲＳ１ | 101 | rd | 0111011 | DIVUW |
| 0000001 | ＲＳ２ | ＲＳ１ | 110 | rd | 0111011 | REMW |
| 0000001 | ＲＳ２ | ＲＳ１ | 111 | rd | 0111011 | REMUW |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct7 | rs２ | rs１ | funct3 | rd | オペコード | R型 |

**RV32A標準拡張**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 00010 | aq | ｒｌ | 00000 | rs１ | 010 | rd | 0101111 | LR.W |
| 00011 | aq | rl | rs２ | rs１ | 010 | rd | 0101111 | SC.W |
| 00001 | aq | rl | rs２ | rs１ | 010 | rd | 0101111 | AMOSWAP.W |
| 00000 | aq | rl | rs２ | rs１ | 010 | rd | 0101111 | AMOADD.W |
| 00100 | aq | rl | rs２ | rs１ | 010 | rd | 0101111 | AMOXOR.W |
| 01100 | aq | rl | rs２ | rs１ | 010 | rd | 0101111 | AMOAND.W |
| 01000 | aq | rl | rs２ | rs１ | 010 | rd | 0101111 | AMOOR.W |
| 10000 | aq | rl | rs２ | rs１ | 010 | rd | 0101111 | AMOMIN.W |
| 10100 | aq | rl | rs２ | rs１ | 010 | rd | 0101111 | AMOMAX.W |
| 11000 | aq | rl | rs２ | rs１ | 010 | rd | 0101111 | AMOMINU.W |
| 11100 | aq | rl | rs２ | rs１ | 010 | rd | 0101111 | AMOMAXU.W |

**RV64A標準拡張（RV32Aに加えてRV64A標準拡張）**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 00010 | aq | rl | 00000 | rs１ | 011 | rd | 0101111 | LR.D |
| 00011 | aq | rl | rs２ | rs１ | 011 | rd | 0101111 | SC.D |
| 00001 | aq | rl | rs２ | rs１ | 011 | rd | 0101111 | AMOSWAP.D |
| 00000 | aq | rl | rs２ | rs１ | 011 | rd | 0101111 | AMOADD.D |
| 00100 | aq | rl | rs２ | rs１ | 011 | rd | 0101111 | AMOXOR.D |
| 01100 | aq | rl | rs２ | rs１ | 011 | rd | 0101111 | AMOAND.D |
| 01000 | aq | rl | rs２ | rs１ | 011 | rd | 0101111 | AMOOR.D |
| 10000 | aq | rl | rs２ | rs１ | 011 | rd | 0101111 | AMOMIN.D |
| 10100 | aq | rl | rs２ | rs１ | 011 | rd | 0101111 | AMOMAX.D |
| 11000 | aq | rl | rs２ | rs１ | 011 | rd | 0101111 | AMOMINU.D |
| 11100 | aq | rl | rs２ | rs１ | 011 | rd | 0101111 | AMOMAXU.D |

312726 2524201915141211760

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| funct7 | |  |  | ｒｓ２ | ｒｓ１ | funct3 | rd | オペコード | R型 |
| ｒｓ３ | funct2 |  |  | ｒｓ２ | ｒｓ１ | funct3 | rd | オペコード | R4型 |
| imm[11:0] | | |  |  | ｒｓ１ | funct3 | rd | オペコード | I型 |
| imm[11:5] | |  |  | ｒｓ２ | ｒｓ１ | funct3 | imm[4:0] | オペコード | S型 |

**RV32F標準拡張**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| imm[11:0 | | ] | ｒｓ１ | 010 | rd | 0000111 | FLW |
| imm[11:5] | | ｒｓ２ | ｒｓ１ | 010 | imm[4:0] | 0100111 | FSW |
| ｒｓ３ | 00 | ｒｓ２ | ｒｓ１ | rm | rd | 1000011 | FMADD.S |
| ｒｓ３ | 00 | ｒｓ２ | ｒｓ１ | rm | rd | 1000111 | FMSUB.S |
| ｒｓ３ | 00 | ｒｓ２ | ｒｓ１ | rm | rd | 1001011 | FNMSUB.S |
| ｒｓ３ | 00 | ｒｓ２ | ｒｓ１ | rm | rd | 1001111 | FNMADD.S |
| 0000000 | | ｒｓ２ | ｒｓ１ | rm | rd | 1010011 | FADD.S |
| 0000100 | | ｒｓ２ | ｒｓ１ | rm | rd | 1010011 | FSUB.S |
| 0001000 | | ｒｓ２ | ｒｓ１ | rm | rd | 1010011 | FMUL.S |
| 0001100 | | ｒｓ２ | ｒｓ１ | rm | rd | 1010011 | FDIV.S |
| 0101100 | | 00000 | ｒｓ１ | rm | rd | 1010011 | FSQRT.S |
| 0010000 | | ｒｓ２ | ｒｓ１ | 000 | rd | 1010011 | FSGNJ.S |
| 0010000 | | ｒｓ２ | ｒｓ１ | 001 | rd | 1010011 | FSGNJN.S |
| 0010000 | | ｒｓ２ | ｒｓ１ | 010 | rd | 1010011 | FSNJX.S |
| 0010100 | | ｒｓ２ | ｒｓ１ | 000 | rd | 1010011 | FMIN.S |
| 0010100 | | ｒｓ２ | ｒｓ１ | 001 | rd | 1010011 | FMAX.S |
| 1100000 | | 00000 | ｒｓ１ | rm | rd | 1010011 | FCVT.W.S |
| 1100000 | | 00001 | ｒｓ１ | rm | rd | 1010011 | FCVT.WU.S |
| 1110000 | | 00000 | ｒｓ１ | 000 | rd | 1010011 | FMV.X.W |
| 1010000 | | ｒｓ２ | ｒｓ１ | 010 | rd | 1010011 | FEQ.S |
| 1010000 | | ｒｓ２ | ｒｓ１ | 001 | rd | 1010011 | FLT.S |
| 1010000 | | ｒｓ２ | ｒｓ１ | 000 | rd | 1010011 | FLE.S |
| 1110000 | | 00000 | ｒｓ１ | 001 | rd | 1010011 | FCLASS.S |
| 1101000 | | 00000 | ｒｓ１ | rm | rd | 1010011 | FCVT.S.W |
| 1101000 | | 00001 | ｒｓ１ | rm | rd | 1010011 | FCVT.S.WU |
| 1111000 | | 00000 | ｒｓ１ | 000 | rd | 1010011 | FMV.W.X |

**RV64F スタンダードエクステンション（RV32Fに加えて）**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1100000 | 00010 | ＲＳ１ | rm | rd | 1010011 | FCVT.L.S |
| 1100000 | 00011 | ＲＳ１ | rm | rd | 1010011 | FCVT.LU.S |
| 1101000 | 00010 | ＲＳ１ | rm | rd | 1010011 | FCVT.S.L |
| 1101000 | 00011 | ＲＳ１ | rm | rd | 1010011 | FCVT.S.LU |

312726 2524201915141211760

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| funct7 | |  |  | ｒｓ２ | ｒｓ１ | funct3 | rd | オペコード | R型 |
| ｒｓ３ | funct2 |  |  | ｒｓ２ | ｒｓ１ | funct3 | rd | オペコード | R4型 |
| imm[11:0] | | |  |  | ｒｓ１ | funct3 | rd | オペコード | I型 |
| imm[11:5] | |  |  | ｒｓ２ | ｒｓ１ | funct3 | imm[4:0] | オペコード | S型 |

**RV32D標準拡張**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| imm[11:0 | | ] | ｒｓ１ | 011 | rd | 0000111 | FLD |
| imm[11:5] | | ｒｓ２ | ｒｓ１ | 011 | imm[4:0] | 0100111 | FSD |
| ｒｓ３ | 01 | ｒｓ２ | ｒｓ１ | rm | rd | 1000011 | FMADD.D |
| ｒｓ３ | 01 | ｒｓ２ | ｒｓ１ | rm | rd | 1000111 | FMSUB.D |
| ｒｓ３ | 01 | ｒｓ２ | ｒｓ１ | rm | rd | 1001011 | FNMSUB.D |
| ｒｓ３ | 01 | ｒｓ２ | ｒｓ１ | rm | rd | 1001111 | FNMADD.D |
| 0000001 | | ｒｓ２ | ｒｓ１ | rm | rd | 1010011 | FADD.D |
| 0000101 | | ｒｓ２ | ｒｓ１ | rm | rd | 1010011 | FSUB.D |
| 0001001 | | ｒｓ２ | ｒｓ１ | rm | rd | 1010011 | FMUL.D |
| 0001101 | | ｒｓ２ | ｒｓ１ | rm | rd | 1010011 | FDIV.D |
| 0101101 | | 00000 | ｒｓ１ | rm | rd | 1010011 | FSQRT.D |
| 0010001 | | ｒｓ２ | ｒｓ１ | 000 | rd | 1010011 | FSGNJ.D |
| 0010001 | | ｒｓ２ | ｒｓ１ | 001 | rd | 1010011 | FSGNJN.D |
| 0010001 | | ｒｓ２ | ｒｓ１ | 010 | rd | 1010011 | FSGNJX.D |
| 0010101 | | ｒｓ２ | ｒｓ１ | 000 | rd | 1010011 | FMIN.D |
| 0010101 | | ｒｓ２ | ｒｓ１ | 001 | rd | 1010011 | FMAX.D |
| 0100000 | | 00001 | ｒｓ１ | rm | rd | 1010011 | FCVT.S.D |
| 0100001 | | 00000 | ｒｓ１ | rm | rd | 1010011 | FCVT.D.S |
| 1010001 | | ｒｓ２ | ｒｓ１ | 010 | rd | 1010011 | FEQ.D |
| 1010001 | | ｒｓ２ | ｒｓ１ | 001 | rd | 1010011 | FLT.D |
| 1010001 | | ｒｓ２ | ｒｓ１ | 000 | rd | 1010011 | FLE.D |
| 1110001 | | 00000 | ｒｓ１ | 001 | rd | 1010011 | FCLASS.D |
| 1100001 | | 00000 | ｒｓ１ | rm | rd | 1010011 | FCVT.W.D |
| 1100001 | | 00001 | ｒｓ１ | rm | rd | 1010011 | FCVT.WU.D |
| 1101001 | | 00000 | ｒｓ１ | rm | rd | 1010011 | FCVT.D.W |
| 1101001 | | 00001 | ｒｓ１ | rm | rd | 1010011 | FCVT.D.WU |

**RV64D標準拡張（RV32Dに加えて）**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1100001 | 00010 | ｒｓ１ | rm | rd | 1010011 | FCVT.L.D |
| 1100001 | 00011 | ｒｓ１ | rm | rd | 1010011 | FCVT.LU.D |
| 1110001 | 00000 | ｒｓ１ | 000 | rd | 1010011 | FMV.X.D |
| 1101001 | 00010 | ｒｓ１ | rm | rd | 1010011 | FCVT.D.L |
| 1101001 | 00011 | ｒｓ１ | rm | rd | 1010011 | FCVT.D.LU |
| 1111001 | 00000 | ｒｓ１ | 000 | rd | 1010011 | FMV.D.X |

312726 2524201915141211760

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| funct7 | |  |  | ｒｓ２ | ｒｓ１ | funct3 | rd | オペコード | R型 |
| ｒｓ３ | funct2 |  |  | ｒｓ２ | ｒｓ１ | funct3 | rd | オペコード | R4型 |
| imm[11:0] | | |  |  | ｒｓ１ | funct3 | rd | オペコード | I型 |
| imm[11:5] | |  |  | ｒｓ２ | ｒｓ１ | funct3 | imm[4:0] | オペコード | S型 |

**RV32Q標準拡張**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| imm[11:0 | | ] | ｒｓ１ | 100 | rd | 0000111 | FLQ |
| imm[11:5] | | ｒｓ２ | ｒｓ１ | 100 | imm[4:0] | 0100111 | FSQ |
| ｒｓ３ | 11 | ｒｓ２ | ｒｓ１ | rm | rd | 1000011 | FMADD.Q |
| ｒｓ３ | 11 | ｒｓ２ | ｒｓ１ | rm | rd | 1000111 | FMSUB.Q |
| ｒｓ３ | 11 | ｒｓ２ | ｒｓ１ | rm | rd | 1001011 | FNMSUB.Q |
| ｒｓ３ | 11 | ｒｓ２ | ｒｓ１ | rm | rd | 1001111 | FNMADD.Q |
| 0000011 | | ｒｓ２ | ｒｓ１ | rm | rd | 1010011 | FADD.Q |
| 0000111 | | ｒｓ２ | ｒｓ１ | rm | rd | 1010011 | FSUB.Q |
| 0001011 | | ｒｓ２ | ｒｓ１ | rm | rd | 1010011 | FMUL.Q |
| 0001111 | | ｒｓ２ | ｒｓ１ | rm | rd | 1010011 | FDIV.Q |
| 0101111 | | 00000 | ｒｓ１ | rm | rd | 1010011 | FSQRT.Q |
| 0010011 | | ｒｓ２ | ｒｓ１ | 000 | rd | 1010011 | FSGNJ.Q |
| 0010011 | | ｒｓ２ | ｒｓ１ | 001 | rd | 1010011 | FSGNJN.Q |
| 0010011 | | ｒｓ２ | ｒｓ１ | 010 | rd | 1010011 | FSGNJX.Q |
| 0010111 | | ｒｓ２ | ｒｓ１ | 000 | rd | 1010011 | FMIN.Q |
| 0010111 | | ｒｓ２ | ｒｓ１ | 001 | rd | 1010011 | FMAX.Q |
| 0100000 | | 00011 | ｒｓ１ | rm | rd | 1010011 | FCVT.S.Q |
| 0100011 | | 00000 | ｒｓ１ | rm | rd | 1010011 | FCVT.Q.S |
| 0100001 | | 00011 | ｒｓ１ | rm | rd | 1010011 | FCVT.D.Q |
| 0100011 | | 00001 | ｒｓ１ | rm | rd | 1010011 | FCVT.Q.D |
| 1010011 | | ｒｓ２ | ｒｓ１ | 010 | rd | 1010011 | FEQ.Q |
| 1010011 | | ｒｓ２ | ｒｓ１ | 001 | rd | 1010011 | FLT.Q |
| 1010011 | | ｒｓ２ | ｒｓ１ | 000 | rd | 1010011 | FLE.Q |
| 1110011 | | 00000 | ｒｓ１ | 001 | rd | 1010011 | FCLASS.Q |
| 1100011 | | 00000 | ｒｓ１ | rm | rd | 1010011 | FCVT.W.Q |
| 1100011 | | 00001 | ｒｓ１ | rm | rd | 1010011 | FCVT.WU.Q |
| 1101011 | | 00000 | ｒｓ１ | rm | rd | 1010011 | FCVT.Q.W |
| 1101011 | | 00001 | ｒｓ１ | rm | rd | 1010011 | FCVT.Q.WU |

**RV64Q標準拡張(RV32Qに加えて)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1100011 | 00010 | ＲＳ１ | ラム | rd | 1010011 | FCVT.L.Q |
| 1100011 | 00011 | ＲＳ１ | ラム | rd | 1010011 | FCVT.LU.Q |
| 1101011 | 00010 | ＲＳ１ | ラム | rd | 1010011 | FCVT.Q.L |
| 1101011 | 00011 | ＲＳ１ | ラム | rd | 1010011 | FCVT.Q.LU |

表 24.2.RISC-V の命令リスト

表 24.3 に、現在 CSR アドレスが割り当てられている CSR を示します。タイマー、カウンタ、および浮動小数点型CSRは、この仕様で定義されている唯一のCSRです。

|  |  |  |  |
| --- | --- | --- | --- |
| 数 | 特権 | 名前 | 説明 |
| 浮動小数点制御およびステータスレジスタ | | | |
| 0x001 | 読み取り/書き込み | fflags | 浮動小数点の未処理の例外。 |
| 0x002 | 読み取り/書き込み | frm | 浮動小数点ダイナミック丸めモード。 |
| 0x003 | 読み取り/書き込み | fcsr | 浮動小数点制御およびステータスレジスタ (frm + fflags)。 |
| カウンタとタイマー | | | |
| 0xC00 | 読み取り専用 | cycle | RDCYCLE 命令のサイクルカウンタ。 |
| 0xC01 | 読み取り専用 | time | RDTIME命令のタイマー。 |
| 0xC02 | 読み取り専用 | instret | RDINSTRET命令の命令リタイアカウンター。 |
| 0xC80 | 読み取り専用 | cycleh | cycleの上位32ビット、RV32Iのみ。 |
| 0xC81 | 読み取り専用 | timeh | timeの上位32ビット、RV32Iのみ。 |
| 0xC82 | 読み取り専用 | instreth | instretの上位32ビット、RV32Iのみ。 |

表 24.3.RISC-V 制御およびステータスレジスタ(CSR)のアドレスマップ

**第25章**

# RISC-V アセンブリプログラマハンドブック

この章は、アセンブリプログラマのマニュアルのプレースホルダです。

表 25.1 に、x と f レジスタのアセンブラニーモニックと、最初の標準呼び出し規約での役割を示します。

|  |  |  |  |
| --- | --- | --- | --- |
| レジスタ | ABI名 | 説明 | セーバー |
| x0 | zero | ハードワイヤードゼロ | --- |
| x1 | ra | 戻りアドレス | 呼び出し元 |
| x2 | sp | スタックポインタ | 呼び出し先 |
| x3 | gp | グローバルポインタ | --- |
| x4 | tp | スレッドポインタ | --- |
| x5 | t0 | 一時/代理レジスタ | 呼び出し元 |
| x6-7 | t1-2 | 一時的 | 呼び出し元 |
| x8 | s0/fp | 保存されたレジスタ/フレームポインタ | 呼び出し先 |
| x9 | s1 | 保存されたレジスタ | 呼び出し先 |
| x10-11 | a0-1 | 関数の引数/戻り値 | 呼び出し元 |
| x12-17 | a2-7 | 関数の引数 | 呼び出し元 |
| x18-27 | s2-11 | 保存されたレジスタ | 呼び出し先 |
| x28-31 | t3-6 | 一時的 | 呼び出し元 |
| f0-7 | ｆｔ０-７ | FP一時的 | 呼び出し元 |
| f8-9 | fs０-１ | FP保存レジスタ | 呼び出し先 |
| f10-11 | fa０-１ | FP引数/戻り値 | 呼び出し元 |
| f12-17 | fa2-7 | ＦＰ 引数 | 呼び出し元 |
| f18-27 | fs2-11 | FP保存レジスタ | 呼び出し先 |
| f28-31 | ft8-11 | FP一時的 | 呼び出し元 |

表 25.1.RISC-V 整数レジスタおよび浮動小数点レジスタ用のアセンブラニーモニックと、最初の標準呼び出し規約におけるそれらの役割。

137

*将来的には異なる呼び方があるかもしれませんが、レジスタ*x1*、*x2*、および*x5は*標準ISAおよび/または圧縮拡張でエンコードされた特別な意味を持つことに注意してください。*

表25.2と25.3には、標準的なRISC-V疑似命令のリストが含まれています。

|  |  |  |
| --- | --- | --- |
| 疑似命令 | 基命令(s) | 意味 |
| la rd, symbol (*non-PIC*) | auipc rd, delta[31:12]+delta[11]  addi rd, rd, delta[11:0] | 絶対アドレスロード  delta = symbol – pcの時 |
| la rd, symbol *(PIC )* | auipc rd, delta[31 : 12] + delta[11]  l{w|d} rd, rd, delta[11:0] | 絶対アドレスロード  delta = GOT[symbol]–pcの時 |
| lla rd, symbol | auipc rd, delta[31 : 12] + delta[11]  addi rd, rd, delta[11:0] | 局所アドレスロード  delta = symbol – pcの時 |
| l{b|h|w|d} rd, symbol | auipc rd, delta[31 : 12] + delta[11]  l{b|h|w|d} rd, delta[11:0](rd) | 広域ロード |
| s{b|h|w|d} rd, symbol, rt | auipc rt, delta[31 : 12] + delta[11]  s{b|h|w|d} rd, delta[11:0](rt) | 広域ストア |
| fl{w|d} rd, symbol, rt | auipc rt, delta[31 : 12] + delta[11]  fl{w|d} rd, delta[11:0](rt) | 浮動小数点ロード広域 |
| fs{w|d} rd, symbol, rt | auipc rt, delta[31 : 12] + delta[11]  fs{w|d} rd, delta[11:0](rt) | 浮動小数点ストア広域 |

*ベース 命令は* pc 相対アドレッシングを*使用するため、リンカは*シンボル*から* pc を減算*して* delta を*取得します。リンカは*delta[11] を*20 ビッ トの上位部分に追加し、12 ビッ トの下位部分の符号拡張を相殺します。*

|  |  |  |
| --- | --- | --- |
| nop | addi x0, x0, 0 | 操作なし |
| li rd, immediate | *無数のシーケンス* | 即時読み込み |
| ｍｖ ｒｄ、ｒｓ | addi rd, rs, 0 | コピーレジスター |
| not ｒｄ、ｒｓ | xori rd, rs, -1 | 1の補数 |
| neg ｒｄ、ｒｓ | sub rd, x0, rs | 2の補数 |
| negw rd, rs | subｗ ｒｄ，ｘ０，ｒｓ | 2の補数ワード |
| sext.w ｒｄ、ｒｓ | addiw rd、rs、0 | 符号拡張ワード |
| seqz rd、ｒｓ | sltiu rd、ｒｓ、1 | = ゼロの時セット |
| snez rd、ｒｓ | sltu rd, x0, rs | ≠ ゼロ の時セット |
| sltz rd、ｒｓ | slt rd, rs, x0 | < ゼロ の時セット |
| ｓｇｔｚ ｒｄ、ｒｓ | slt rd, x0, rs | > ゼロの時セット |
| fmv.s.s.rd, rs | fsgnj.s rd, rs, rs | 単精度レジスタをコピー |
| fabs.ｓ ｒｄ、ｒｓ | fsgnjx.s rd, rs, rs | 単精度絶対値 |
| ｆｎｅｇ.ｓ ｒｄ、ｒｓ | fsgnjn.s rd, rs, rs, rs | 単精度否定 |
| fmv.d rd, rs | fsgnj.d rd, rs, rs | 倍精度レジスタをコピー |
| fabs.d ｒｄ、ｒｓ | fsgnjx.d rd, rs, rs | 倍精度絶対値 |
| ｆｎｅｇ.ｄ ｒｄ、ｒｓ | fsgnjn.d rd, rs, rs | 倍精度否定 |
| beqz rs, offset | beq rs, x0, offset | = ゼロの時分岐 |
| bnez rs, offset | bne rs, x0, offset | ≠ ゼロの時分岐 |
| blez rs、offset | bge x0, rs, offset | ≦ ゼロの時分岐 |
| bgez rs, offset | bge rs, x0, offset | ≧ ゼロの時分岐 |
| bltz rs、offset | blt rs, x0, offset | ＜ ゼロの時分岐 |
| bgtz rs、offset | blt x0, rs, offset | ＞ ゼロの時分岐 |
| bgt rs, rt, offset | blt rt, rs, offset | ＞ の時分岐 |
| ble ｒｓ、ｒｔ、offset | bge rt, rs, offset | ≦ の時分岐 |
| bgtu rs, rt, offset | bltu rt, rs, offset | ＞ の時分岐、符号なし |
| bleu ｒｓ、ｒｔ、offset | bgeu rt, rs, offset | ≦ の時分岐、符号なし |

表25.2.RISC-Vの疑似命令。

疑似命令 基本命令 意味

|  |  |  |
| --- | --- | --- |
| j offset | jal x0, offset | ジャンプ |
| jfal offset | ジャルx1、offset | ジャンプとリンク |
| jr rs | jalr x0, 0(rs) | ジャンプレジスタ |
| jalr rs | jalr x1, 0(rs) | ジャンプとリンクレジスタ |
| ret | jalr x0, 0(x1) | サブルーチンからの復帰 |
| call offset | auipc x1, offset[31:12] + offset[11] jalr x1, offset[11:0](x1) | 遠距離サブルーチン呼び出し |
| tail offset | auipc x6, offset[31 : 12] + offset[11] jalr x0, offset[11:0](x6) | テール遠距離サブルーチン呼び出し |
| fence | fence iorw, iorw | すべてのメモリとI/Oのフェンス |
| rdinstret[h] rd | csrrs rd, instret[h], x0 | 引退命令カウンター読み出し |
| rdcycle[h] rd | csrrs rd, cycle[h], x0 | サイクルカウンタ読み出し |
| rdtime[h] rd | csrrs rd, time[h], x0 | リアルタイムクロック読み出し |
| csrr rd, csr | csrrs rd、csr、x0 | CSRを読む |
| csrw csr, rs | csrrw x0、csr、rs | CSRを書く |
| csrs csr, rs | csrrs x0、csr、rs | CSRの設定ビット |
| csrc csr, rs | csrrc x0, csr, rs | CSRのクリアビット |
| csrwi csr, imm | csrrwi x0、csr、imm | CSRに即値書き込み |
| csrsi csr, imm | csrrsi x0、csr、imm | CSRのビットに即値を設定 |
| csrci csr, imm | csrrci x0、csr、imm | CSRのビットを即値でクリア |
| frcsr rd | csrrs rd, fcsr, x0 | FP制御/ステータスレジスタの読み出し |
| ｆｓｃsｒ ｒｄ、ｒｓ | csrrw rd、fcsr、rs | FP制御/ステータスレジスタの交換 |
| ｆｓｃsr rs | csrrw x0, fcsr, rs | FP制御・ステータスレジスタへの書き込み |
| frrm rd | csrrs rd, frm, x0 | FP丸めモードを読み出し |
| fsrm rd, rs | csrrw rd、frm、rs | FP丸めモードを交換 |
| fsrm rs | csrrw x0, frm, rs | FP丸めモードを書き込み |
| frflags rd | csrrs rd, fflags, x0 | FP例外フラグを読み出し |
| fsflags rd, rs | csrrw rd, fflags, rs | FP例外フラグを交換 |
| fsflags rs | csrrw x0, fflags, rs | FP例外フラグを書き込み |

表25.3.RISC-Vの疑似命令

**第26章**

# RISC-Vの拡張

標準的な汎用ソフトウェア開発をサポートするだけでなく、RISC-Vのもう一つの目標は、より特殊な命令セット拡張やカスタマイズされたアクセラレータの基盤を提供することです。命令エンコーディングスペースとオプションの可変長命令エンコーディングは、よりカスタマイズされたプロセッサを構築する際に、標準的なISAツールチェーンのソフトウェア開発の労力を容易に活用できるように設計されています。例えば、標準の I ベースのみを使用するインプリメンテーションに対して、標準以外の多くの命令セット拡張機能と一緒に完全なソフトウェアサポートを提供し続けることを意図しています。

この章では、ベースとなるRISC-VのISAを拡張するための様々な方法と、独立したグループで開発された命令セット拡張を管理するためのスキームについて説明します。第2巻で説明したスーパバイザレベルの拡張についても同じアプローチと用語を使用していますが、本編では特権のないISAのみを扱います。

## 26.1拡張用語

ここでは、RISC-Vの拡張機能を記述するためのいくつかの標準用語を定義します。

### 標準と非標準の拡張

RISC-Vプロセッサの実装は、基本整数ISA(RV32IまたはRV64I)をサポートしなければなりません。さらに、1つの実装で1つ以上の拡張機能をサポートしている場合もあります。拡張機能は、*標準*と*非標準の2つに大別されます*。

* 標準拡張とは、一般的に有用で、他の標準拡張と衝突しないように設計されたものを指します。現在、このマニュアルの他の章で説明されている“MAFDQLCBTPV"は完全な標準拡張か計画された標準拡張です。
* 標準以外の拡張機能は、高度に専門化されている可能性があり、他の標準拡張機能や標準以外の拡張機能と競合する可能性があります。私たちは、標準ではない拡張機能が時間をかけて開発され、最終的には標準の拡張機能に昇格するものもあると予想しています。

141

### 命令のエンコーディング空間と接頭辞

命令エンコーディング空間とは、ベースとなるISAやISAの拡張命令をエンコーディングするためのいくつかの命令ビット数のことです。RISC-Vは様々な命令長をサポートしていますが、一つの命令長の中にも様々なサイズのエンコーディング空間があります。例えば、ベースISAは30ビットのエンコーディング空間(32ビット命令の31-2ビット)で定義されており、アトミック拡張 “A"は25ビットのエンコーディング空間(31-7ビット)で定義されています。

ここでは、命令エンコーディング空間の*右側の*ビットを*プレフィックスと呼んで*います（RISC-Vの命令フェッチはリトルエンディアンなので、右側のビットは以前のメモリアドレスに格納され、命令フェッチ順にプレフィックスを形成しています）。標準的なベースISAエンコーディングのプレフィックスは32ビットワードの1～0ビットにある2ビットの “11"フィールドで、標準的なアトミック拡張 ”A"のプレフィックスはAMOのメジャーオペコードを表す32ビットワードの6～0ビットにある7ビットの “010101111"フィールドです。符号化形式の癖として、マイナーオペコードを符号化するために使用される 3 ビットの funct3 フィールドは、32 ビット命令形式のメジャーオペコードのビットとは連続していませんが、22 ビット命令空間のプレフィックスの一部と考えられています。

命令エンコーディング空間はどのようなサイズであっても構いませんが、より小さな共通サイズのセットを採用することで、独立して開発された拡張機能を単一のグローバルエンコーディングにまとめることが容易になります。表26.1にRISC-Vの推奨サイズを示します。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| サイズ | 使用方法 | # 標準的な命令長で利用可能 | | | |
| 16bit | ３２bit | 48bit | 64bit |
| 14bit | 圧縮された16ビットエンコーディングの象限 | 3 |  |  |  |
| 22bit | 基本32ビットエンコーディングの低位オペコード |  | 28 | 220 | 235 |
| 25bit | 基本32ビットエンコーディングの主要オペコード |  | 32 | 217 | 232 |
| 30bit | 基本32ビットエンコーディングの象限 |  | 1 | 212 | 227 |
| ３２bit | 48ビットエンコーディングの低位オペコード |  |  | 210 | 225 |
| 37bit | 48ビットエンコーディングの主要オペコード |  |  | 32 | 220 |
| 40bit | 48ビットエンコーディングの象限 |  |  | 4 | 217 |
| 45bit | 64 ビットエンコーディングの副低位オペコード |  |  |  | 212 |
| 48bit | 64 ビットエンコーディングの低位オペコード |  |  |  | 29 |
| 52bit | 64ビットエンコーディングの主要オペコード |  |  |  | 32 |

表26.1.提案された標準的なRISC-V命令のエンコーディングスペースサイズ。

### グリーンフィールドとブラウンフィールドの拡張

グリーンフィールド拡張という用語は、新しい命令エンコーディングスペースへの入力を開始する拡張を表すために使用します。したがって、プレフィックスレベルでのみエンコーディングの競合が発生する可能性があります。ブラウンフィールド拡張という用語を使用して、以前に定義された命令スペース内の既存のエンコーディングに適合する拡張を説明します。ブラウンフィールド拡張は必ずしも特定のグリーンフィールド親エンコーディングに結びついているわけではなく、同じグリーンフィールド親エンコーディングに対して複数のブラウンフィールド拡張が存在する場合もあります。例えば、ベースISAは30ビットの命令空間のグリーンフィールドエンコーディングであり、FDQ浮動小数点拡張は親ベースISAの30ビットエンコーディング空間に追加されたすべてのブラウンフィールド拡張です。

標準のA 拡張は、 標準のプレフィックスがベースのISA の30 ビッ ト エン コ ーデ ィ ン グ空間内に位置しているにもかかわらず、32 ビッ トのベース命令エンコーデ ィ ングの左端のビ ッ ト 内に以前は空だった25 ビッ ト のエン コ ーデ ィ ン グ空間を新たに定義しているため、 グリーンフィールド エン コ ーデ ィ ン グを持っていると考えています。7 ビ ッ ト のプ レ フ ィ ッ ク ス を変更す る だけで、A 拡張機能を別の30 ビッ ト のエン コ ーデ ィ ン グ スペース に移動 さ せ る こ と がで き ます。

|  |  |  |
| --- | --- | --- |
|  | 状態を追加 | 新しい状態で無い |
| グリーンフィールド | RV32I(30)、RV64I(30) | A(25) |
| ブラウンフィールド | F(I)、D(F)、Q(D) | M（I） |

表 26.2: 標準命令セット拡張の二次元的な特徴付け

表 26.2 は、単純な二次元のタクソノミに配置された拠点と標準的な拡張を示したものです。1つの軸は拡張がグリーンフィールドかブラウンフィールドかを表し、もう1つの軸はアーキテクチャ状態を追加するかどうかを表しています。グリーンフィールド拡張では、（ ）内に命令エンコーディング空間のサイズを示します。ブラウンフィールド拡張の場合、それが構築する拡張の名前（グリーンフィールドまたはブラウンフィールド）が括弧内に記載されています。追加のユーザレベルのアーキテクチャ状態は、通常、スーパバイザレベルのシステムへの変更、または標準の呼び出し規約への変更を意味します。

RV64I は、RV32I の拡張ではなく、別の完全なベースエンコーディングと考えられていることに注意してください。

**標準互換性のある広域エンコーディング**

実際のRISC-V実装におけるISAの完全なグローバルエンコーディングでは、すべての命令エンコーディング空間に、競合しない一意のプレフィックスが割り当てられなければなりません。ベースと標準拡張は、グローバルエンコーディングで共存できるように、それぞれに標準プレフィックスが割り当てられています。

*標準互換性のある*グローバルエンコーディングとは、 ベースとそれに含まれるすべての標準拡張子がそれぞれの標準接頭辞を持つものです。標準互換グローバルエンコーディングは、標準拡張子と競合しない非標準拡張子を含むことができます。標準互換のグローバルエンコーディングは、関連する標準拡張子がグローバルエンコーディングに含まれていない場合、非標準拡張子に標準接頭辞を使用することもできます。言い換えれば、標準と互換性のあるグローバルエンコーディングに含まれている場合、標準拡張はその標準プレフィックスを使用しなければなりませんが、それ以外の場合、そのプレフィックスは自由に再割り当てされます。これらの制約により、共通ツールチェーンは、任意のRISC-V標準互換グローバルエンコーディングの標準サブセットをターゲットにすることができます。

### 保証された非標準符号化空間

独自のカスタム拡張機能の開発をサポートするために、エンコーディングスペースの一部が標準の拡張機能では絶対に使用されないことが保証されています。

## 26.2 RISC-V拡張設計思想

私たちは、独自に開発された多数の拡張機能をサポートするために、拡張機能の開発者が命令エンコーディング空間内で動作するように奨励し、ユニークなプレフィックスを割り当てて標準互換性のあるグローバルエンコーディングにそれらをパックするツールを提供することを意図しています。エクステンションの中には、既存のエクステンションのブラウンフィールド拡張として実装する方が自然であり、親のグリーンフィールドエクステンションに割り当てられているプレフィックスを共有するものもあります。標準の拡張プレフィックスは、コア機能のエンコーディングにおける真の非互換性を回避する一方で、より難解な拡張機能のカスタムパッキングを可能にしています。

RISC-Vの拡張機能を異なる標準互換のグローバルエンコーディングにリパックするこの機能は、多くの方法で使用することができます。

一つのユースケースは、重要なアプリケーションドメインのカーネルを実行するように設計された、高度に特化したカスタムアクセラレータを開発することです。このようなケースでは、ベースの整数ISA以外のすべてを削除して、手元のタスクに必要な拡張機能のみを追加したいと考えるかもしれません。ベースのISAは、ハードウェア実装の要件を最小限に抑えるように設計されており、32ビット命令エンコーディング空間のごく一部を使用するようにエンコードされています。

もう一つのユースケースは、新しいタイプの命令セット拡張の研究用プロトタイプを構築することです。研究者は、可変長の命令フェッチユニットを実装するための労力を費やしたくないかもしれないので、単純な32ビット固定幅命令エンコーディングを使用して拡張機能をプロトタイプ化したいと考えています。しかし、この新しい拡張は、32ビット空間の標準的な拡張と共存するには大きすぎるかもしれません。研究実験で標準拡張のすべてを必要としない場合、標準互換性のあるグローバルエンコーディングは、研究プロトタイプのエンジニアリングを簡単にするために、使われていない標準拡張を落として、その接頭辞を再利用して、提案された拡張を標準ではない場所に配置するかもしれません。標準ツールは、開発時間を短縮するために、ベースと存在する標準拡張をターゲットにすることができるようになります。一旦、命令セット拡張が評価され、洗練された後は、すべての標準拡張との競合を避けるために、より大きな可変長のエンコーディング空間にパッキングするために利用できるようにすることができます。

以下のセクションでは、新しい命令セット拡張を使用したインプリメンテーションを開発するための高度な戦略について説明します。これらはRISC-V ISA開発のメインラインではなく、高度にカスタマイズされた教育用アーキテクチャや実験用アーキテクチャでの使用を目的としています。

## 26.3 固定幅32ビット命令フォーマット内での拡張機能

このセクションでは、基本の固定幅32ビット命令フォーマットのみをサポートする実装への拡張機能の追加について説明します。

*最もシンプルな固定幅32ビットエンコーディングは、多くの制限付きアクセラレータや研究用プロトタイプに普及すると予想しています。*

### 利用可能な30ビット命令エンコーディングスペース

標準エンコーディングでは、利用可能な 30 ビッ ト命令エンコーディング空間のうち 3 つ（2 ビッ ト接頭辞 00, 01, 10 を持つもの）が、オプションの圧縮命令拡張を有効にするために使用されます。しかし、圧縮命令セット拡張を必要としない場合は、これらの3つの30ビットエンコーディングスペースが利用可能になります。これにより、32 ビ ッ ト 形式で利用可能なエン コ ーデ ィ ン グ スペースは 4 倍にな り ます。

### 利用可能な25ビット命令エンコーディングスペース

25 ビット命令エンコーディング空間は、ベースおよび標準拡張エンコーディングのメジャーオペコードに対応しています。

カスタム拡張のために明示的に予約された4つの主要なオペコード（表24.1）があり、それぞれが25ビットのエンコーディング空間を表しています。これらのうち2つはRV128ベースエンコーディングでの最終的な使用のために予約されていますが（OP-IMM-64とOP-64になります）、RV32とRV64の標準または非標準の拡張に使用することができます。

RV64用に予約された2つのオペコード（OP-IMM-32とOP-32）は、RV32のみへの標準および非標準の拡張にも使用できます。

実装が浮動小数点を必要としない場合、標準浮動小数点拡張用に予約された7つのメジャー・オペコード（LOAD-FP、STORE-FP、MADD、MSUB、NMSUB、NMADD、OP-FP）は、非標準拡張用に再利用することができます。同様に、標準アトミック拡張が必要ない場合、AMO メジャーオペコードを再利用することができます。

実装が32ビットより長い命令を必要としない場合、追加の4つの主要なオペコードが利用可能です（表24.1でグレーでマークされているもの）。

ベースのRV32Iエンコーディングでは、11個のメジャーオペコードと3個の予約オペコードのみを使用し、最大18個の拡張が可能です。ベースのRV64Iエンコーディングでは、13個のメジャーオペコードと3個の予約オペコードのみを使用し、最大16個の拡張が可能です。

### 利用可能な22ビット命令エンコーディングスペース

22 ビットのエンコーディング空間は、ベースエンコーディングと標準拡張エンコーディングにおける funct3 マイナーオペコード空間に対応します。いくつかのメジャーオペコードは、完全に占有されていない funct3 フィールドのマイナーオペコードを持ち、いくつかの 22 ビットのエンコーディング空間が利用可能なままになっています。

通常、メジャー・オペコードは命令の残りのビットでオペランドをエンコードするために使用されるフォーマットを選択し、理想的には、ハードウェアのデコードを簡単にするために、拡張はメジャー・オペコードのオペランド・フォーマットに従うべきです。

### その他のスペース

特定のメジャーオペコードの下では、より小さなスペースが利用可能であり、すべてのマイナーオペコードが完全に埋まっているわけではありません。

## 26.4 64ビット命令拡張機能の追加

基本の32 ビ ッ ト 固定幅命令フ ォーマ ッ ト では大きすぎる拡張のためのスペースを提供する最も単純な方法は、自然にアラインメントされた64 ビッ ト命令を追加することです。実装は32ビット基本命令フォーマットをサポートしなければなりませんが、命令フェッチを簡単にするために64ビット命令を64ビット境界にアラインメントし、必要に応じて32ビットNOP命令をアラインメントパディングとして使用することを要求することができます。

標準ツールの使用を簡単にするために、64 ビット命令は図 1.1 に示すようにエンコードする必要があります。しかし、実装では、32ビット命令の標準エンコーディングを維持しつつ、64ビット命令のために非標準の命令長エンコーディングを選択することができます。例えば、圧縮命令が必要ない場合、64ビット命令は命令の最初の2ビットに1つ以上のゼロビットを使用してエンコードすることができます。

*私たちは、サポートされている可変長命令エンコーディングの任意の組み合わせを自動的に処理できる命令フェッチユニットを生成するプロセッサジェネレータを期待しています。*

## 26.5 VLIW エンコーディングのサポート

RISC-V は純粋な VLIW マシンのベースとして設計されていませんが、いくつかの代替アプローチを使用して VLIW エンコーディングを拡張として追加することができます。いずれの場合も、標準的なソフトウェアツールを使用できるように、ベースとなる32ビットエンコーディングがサポートされている必要があります。

### 固定サイズ命令群

最も単純なアプローチは、単一の大きな自然整列命令フォーマット(例えば128ビット)を定義し、その中でVLIW演算をエンコードすることです。従来のVLIWでは、このアプローチはNOPを保持するために命令メモリを浪費する傾向がありますが、RISC-V互換の実装では、基本32ビット命令もサポートしなければならず、VLIWコードサイズの拡張はVLIW加速関数に限定されます。

### 符号化された長さのグループ

別のアプローチは、図 1.1 の標準長エンコーディングを使用して並列命令グループをエンコードし、NOP を VLIW 命令から圧縮できるようにすることです。た と えば、64 ビッ ト命令は28 ビッ ト演算を2 つ保持し、96 ビッ ト命令は28 ビッ ト演算を3 つ保持することができます。また、48ビット命令は42ビット演算を1つ保持し、96ビット命令は42ビット演算を2つ保持することができます。

このアプローチは、単一の操作を保持する命令のためのベースのISAエンコーディングを保持するという利点がありますが、VLIW命令内の操作のために新しい28ビットまたは42ビットのエンコーディングを必要とし、より大きなグループのために命令フェッチの位置がずれてしまうという欠点があります。1 つの単純化は、VLIW 命令が特定のマイクロアーキテクト的に重要な境界 (キャッシュラインや仮想メモリページなど) をまたぐことを許可しないことです。

### 固定サイズの命令バンドル

Itanium に似た別のアプローチとして、より大きな自然整列固定命令バンドルサイズ（例えば 128 ビッ ト）を使用して並列演算グループをエンコードする方法があります。これは命令フェッチを単純化しますが、複雑さをグループ実行エンジンにシフトさせます。RISC-Vとの互換性を維持するためには、ベースとなる32ビット命令をサポートする必要があります。

### プレフィックスのグループ終了ビット

上記のいずれのアプローチも、VLIW 命令内の個々の演算のための RISC-V エンコーディングを保持していません。もう一つのアプローチは、固定幅32ビットエンコーディングの2つのプレフィックスビットを再利用することです。1つのプレフィックスビットは、セットされていれば「グループの終了」を示すのに使用でき、2つ目のビットはクリアされていれば述語の下での実行を示すのに使用できます。VLIW 拡張を知らないツールによって生成された標準的な RISC-V 32 ビット命令は、両方のプレフィックスビットがセットされている (11) ので、正しいセマンティクスを持ち、各命令はグループの最後にあり、述語はありません。

このアプローチの主な欠点は、ベースのISAには、攻撃的なVLIWシステムで通常必要とされる複雑な述語サポートが欠けており、標準の30ビットのエンコーディング空間でより多くの述語レジスタを指定するための空間を追加することが困難であることです。

**第27章**

# ISA拡張の命名規則

この章では、ハードウェア実装に存在する命令セット、またはアプリケーション・バイナリ・インターフェース(ABI)で使用される命令セットを簡潔に記述するために使用されるRISC-V ISA拡張ネーミング・スキームについて説明します。

*RISC-V ISAは、様々な実験的な命令セット拡張を伴う様々な実装をサポートするように設計されています。整理されたネーミングスキームにより、ソフトウェアツールやドキュメンテーションが簡素化されることがわかりました。*

**27.1 大文字小文字感度**

ISA 命名文字列は大文字と小文字を区別しません。

## 27.2 基本整数ISA

RISC-V ISA 文字列は、RV32I、RV32E、RV64I、RV128I のいずれかで始まり、基本整数の ISA でサポートされているアドレス空間のサイズをビット単位で示します。

## 27.3 命令セット拡張名

標準的なISA拡張子には、1文字からなる名前が付けられます。例えば、整数ベースの最初の4つの標準拡張は以下の通りです。“M"は整数の乗除算、”A"はアトミックメモリ命令、“F"は単精度浮動小数点命令、”D"は倍精度浮動小数点命令です。RISC-Vの命令セットの変形は、基底整数の接頭辞と含まれる拡張子の名前を連結することで簡潔に記述することができます。（例："RV64IMAFD"）。

また、“IMAFDZicsr Zifencei"のベースと拡張子を表すために、“G"という略語を定義しました。

RISC-V ISA の標準拡張には、他の予約文字が与えられています。例えば、Quad-precision浮動小数点の場合は “Q"、16bit圧縮命令フォーマットの場合は ”C"などです。

例えば、"D "は "F "に依存し、"F "は "Zicsr "に依存するなど、いくつかのISA拡張機能は他の拡張機能の存在に依存します。

例えば、“D”は“F”に依存し、“F”は“Zicsr”に依存します。これらの依存関係は、ISA名に暗黙的に含まれている場合があります。例えば、RV32IFはRV32IFZicsrに相当し、RV32IDはRV32IFDおよびRV32IFDZicsrに相当します。

## 27.4バージョン番号

命令セットは時間の経過とともに拡張されたり変化したりする可能性があることを考慮して、拡張子のバージョン番号を拡張子名の後につけてエンコードしています。バージョン番号はメジャーバージョンとマイナーバージョンに分かれていて、 “p"で区切られています。もしマイナーバージョンが ”0"であれば、バージョン文字列から “p0"を省略することができます。メジャーバージョン番号の変更は下位互換性を失うことを意味しますが、マイナーバージョン番号のみの変更は下位互換性を持たなければなりません。例えば、このマニュアルのリリース1.0で定義されているオリジナルの64ビット標準ISAは完全に ”RV64I1p0P0M1p0A1p0F1p0D1p0"と書くことができますが、より簡潔に言えば “RV64I1M1A1F1D1"と書くことができます。

2回目のリリースからバージョン番号方式を導入しました。そのため、標準拡張機能のデフォルトバージョンをその時点でのバージョンと定義し、例えば ``RV32I"" は ``RV32I2"" と同じです。

## 27.5 アンダースコア

アンダースコア “＿" は、可読性を向上させ、曖昧さをなくすために、ISA拡張を分離するために使われることがあります。（例："RV32I2 M2 A2"）。

Packed SIMD の拡張子 “P" はバージョン番号の小数点と混同される可能性があるため、数字の後に続く場合はアンダースコアを前につけなければなりません。例えば、”rv32i2p2"はRV32Iのバージョン2.2を意味し、“rv32i2\_p2"はRV32Iのバージョン2.0を意味し、P拡張子のバージョン2.0を意味します。

## 27.6 追加の標準拡張名

標準的な拡張機能は “Z" の後にアルファベットの名前とオプションでバージョン番号をつけることもできます。例えば、 ”Zifencei"は第3章で説明されている命令フェッチフェンス拡張機能の名前を付けます。"Zifencei2 "と "Zifencei2p0 "は同じもののバージョン2.0の名称です。

“Z"の後に続く最初の文字は、最も関連性の高いアルファベットの拡張子カテゴリ、IMAFDQLCBJTPVNを示しています。例えば、不正列ア トミックのための ”Zam" 拡張子の場合、 “a" はその拡張子が ”A" 標準拡張子に関連していることを示します。

複数の "Z "拡張子をつける場合は、最初にカテゴリー別に並べ、次にカテゴリー内のアルファベット順に並べる必要があります（例："Zicsr Zifencei Zam"）。

Z "という接頭語を持つ拡張子は、他の複数の文字を持つ拡張子とアンダースコアで区切らなければなりません（例："RV32IMACZicsr Zifencei"）。

## 27.7 スーパーバイザレベルの命令セット拡張

標準スーパバイザレベルの命令セット拡張は第2巻で定義されていますが、名前は“S"をプレフィックスとして使用し、その後にアルファベット名とオプションでバージョン番号を付けてください。スーパーバイザレベルの拡張子は他の複数文字の拡張子とはアンダースコアで区切らなければなりません。

標準スーパバイザレベルの拡張子は、標準の非特権拡張子の後にリストアップする必要があります。複数のスーパバイザレベルの拡張子がリストされている場合は、アルファベット順に並べる必要があります。

## 27.8 ハイパーバイザレベルの命令セット拡張

標準的なハイパーバイザーレベルの命令セット拡張機能はスーパーバイザーレベルの拡張機能と同じように名前が付けられますが、最初は “S"の代わりに ”H"で始まります。

標準的なハイパーバイザーレベルの拡張機能は、標準的な劣等特権拡張機能の後にリストアップする必要があります。複数のハイパーバイザーレベルの拡張機能がリストアップされている場合は、アルファベット順に並べる必要があります。

## 27.9 機械レベルの命令セット拡張

標準的な機械レベルの命令セット拡張子は、プレフィックスとして “Zxm"の3文字が付けられています。

標準的な機械レベルの拡張機能は、標準的な下位特権拡張機能の後にリストアップする必要があります。複数の機械レベル拡張機能がリストアップされている場合は、アルファベット順に並べる必要があります。

## 27.10 非標準の拡張子名

非標準の拡張機能は “X" の後にアルファベットの名前とオプションのバージョン番号を付けて名前を付けます。例えば、 ”Xhwacha" は Hwacha ベクトルフェッチ ISA 拡張の名前を付けます。

非標準の拡張子は、すべての標準拡張子の後に記載する必要があります。これらの拡張子は他の複数文字の拡張子とはアンダースコアで区切らなければなりません。例えば、非標準拡張子 Argle と Bargle を持つ ISA は “RV64IZifencei\_Xargle\_Xbargle" と名付けられます。

複数の非標準拡張子がリストアップされている場合は、アルファベット順に並べる必要があります。

## 27.11 サブセット命名規則

表27.1に標準化された拡張子名をまとめました。

|  |  |  |
| --- | --- | --- |
| サブセット | 名前 | 含む |
| ベース ISA | | |
| 整数 | I |  |
| 縮小整数 | E |  |
| 標準的な非特権拡張 | | |
| 整数の乗除 | M |  |
| アトミック | A |  |
| 単精度浮動小数点 | F | Zicsr |
| 倍精度浮動小数点 | D | F |
| 一般的な | G | IMADZifencei |
| 4倍精度浮動小数点 | Q | D |
| 十進浮動小数点 | L |  |
| 16ビット圧縮命令 | C |  |
| ビット操作 | B |  |
| 動的言語 | J |  |
| トランザクションメモリ | T |  |
| パックドSIMD拡張機能 | P |  |
| ベクトル拡張機能 | V |  |
| ユーザーレベルの割り込み | N |  |
| 制御および状態レジスタアクセス | Zicsr |  |
| 命令-フェッチフェンス | Zifencei |  |
| アトミック | Zam | A |
| トータルストアオーダー | Ztso |  |
| 標準的なスーパーバイザーレベルの拡張機能 | | |
| スーパーバイザレベル拡張 “def" | Sdef |  |
| 標準的なハイパーバイザーレベルの拡張機能 | | |
| ハイパーバイザーレベルの拡張 “ghi" | Hghi |  |
| 標準的な機械レベルの拡張機能 | | |
| 機械レベルの拡張 “jkl" | Zxmjkl |  |
| 非標準の拡張機能 | | |
| 非標準拡張子 “mno" | Ｘmno |  |

表 27.1: 標準的なISA拡張子の名前。また、この表では、拡張子の名前を名前文字列に記載する際の正規の順序を定義しており、表の上から下が名前文字列の最初から最後を意味します。例えば、RV32IMACVは合法ですが、RV32IMAVCは合法ではありません。27.1.標準ISA拡張名。この表では、拡張名が名前文字列の中で最初から最後まで表示されなければならない順序も定義しています。

**第28章**

# 歴史と謝辞

## 28.1 “なぜ新しいISAを開発するのか？"バークレーグループからの理由

私たちのグループは、研究と教育におけるニーズをサポートするためにRISC-Vを開発しました。私たちのグループは、研究アイデアの実際のハードウェア実装に特に興味を持っており（この仕様の第1版以降、11種類のRISC-Vのシリコンファブリケーションを完成させました）、学生が授業で探求するための実際の実装を提供しています（RISC-VプロセッサのRTLデザインは、バークレー大学の複数の学部および大学院のクラスで使用されています）。私たちの現在の研究では、従来のトランジスタのスケーリングの終わりによって課せられた電力制約によって、特殊化されたヘテロジニアスなアクセラレータへと向かう動きに特に興味を持っています。私たちは、私たちの研究努力を構築するために、非常に柔軟で拡張性の高いベースISAを求めていました。

私たちが何度も聞かれている質問は「なぜ新しいISAを開発するのか？」。既存の商用ISAを使用することの最大の利点は、開発ツールと移植されたアプリケーションの両方で、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あり、研究や教育に活用することができます。その他の利点としては、大量のドキュメントやチュートリアルの例が存在することが挙げられます。しかし、研究や教育のために商用のインストラクションセットを使用した私たちの経験では、これらのメリットは実際には小さく、デメリットを上回るものではありません。

* **商用ISAはプロプライエタリです。**オープンなIEEE標準であるSPARC V8 [2]を除いて、商用ISAのほとんどの所有者は知的財産を慎重に守り、自由に利用できる競争的な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的な研究や教育では問題になりませんが、実際のRTL実装を共有したいと考えているグループにとっては大きな懸念材料となっています。また、商業的なISA実装の数少ない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を禁止されている団体にとっても、これは大きな懸念材料となっています。私たちは、すべてのRISC-V実装がサードパーティの特許侵害から解放されることを保証することはできませんが、RISC-V実装者を訴えようとしないことを保証することはできます。
* **商用ISAは、特定の市場領域でのみ普及しています。**この記事を書いている時点での最も明白な例は、ARMアーキテクチャがサーバー分野で十分にサポートされていないことと、Intel x86アーキテクチャ（というか、ほとんどすべての他のアーキテクチャ）がモバイル分野で十分にサポートされていないことだが、IntelとARMの両社はお互いの市場セグメントに参入しようとしています。

また、他の例として拡張可能なコアを提供しているARCやTensilicaは、組み込み分野に特化しています。このように市場が細分化されると、特定の商用ISAをサポートするメリットが薄れてしまいます。実際には、ソフトウェアのエコシステムは特定の領域にしか存在せず、他の領域のためには構築する必要があるからです。

* **商業的なISAは行き交っている。これ**までの研究インフラは、もはや普及していない商用 ISA を中心に構築されてきた(SPARC, MIPS)か、あるいはもはや生産されていない(Alpha)。これらは、活発なソフトウェアエコシステムの利点を失い、ISAとサポートツールの周りに残る知的財産権の問題は、興味を持っているサードパーティがISAをサポートし続ける能力を妨害します。オープンなISAは人気を失うかもしれませんが、関心のある当事者はエコシステムを使い続け、開発し続けることができます。
* **人気のある商用ISAは複雑。**主要な商用 ISA（x86 と ARM）は、一般的なソフトウェアスタックとオペレーティング・システムをサポートするレベルまでハードウェアに実装するには、どちらも非常に複雑です。さらに悪いことに、ほとんどすべての複雑さの原因は、真に効率を向上させる機能よりも、ISAの設計上の決定が間違っていたり、少なくとも時代遅れであったりすることです。
* **商用ISAだけではアプリケーションが立ち上がらない。**商用ISAを実装するために労力を費やしても、そのISAのための既存のアプリケーションを動かすには十分ではありません。ほとんどのアプリケーションは、ユーザーレベルのISAだけでなく、実行するために完全なABI（アプリケーション・バイナリ・インターフェース）を必要とします。ほとんどの ABI はライブラリに依存しており、そのライブラリはオペレーティングシステムのサポートに依存しています。既存のオペレーティングシステムを実行するには、OS が期待するスーパバイザレベルの 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よりもはるかに仕様が少なく、実装がかなり複雑です。
* **ポピュラーな商用 ISA は拡張性を考慮して設計されていませんでした。**主要な商用ISAは、特に拡張性を考慮して設計されておらず、その結果、命令セットが大きくなるにつれて、かなりの命令エンコーディングの複雑さを追加してきました。Tensilica（Cadenceが買収）やARC（Synopsysが買収）などの企業は、拡張性を中心にISAやツールチェーンを構築してきましたが、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焦点を当ててきました。
* **改造された商用ISAとは、新しいISAのことです。**主な目的の一つは、主要なISAの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。小さな拡張であっても、その拡張を使用するためにコンパイラを変更したり、アプリケーションをソースコードから再構築したりしなければならないため、標準的なISAを使用するメリットが薄れてしまいます。新しいアーキテクチャ状態を導入する大規模な拡張でも、オペレーティングシステムの変更が必要になります。最終的には、変更された商用 ISA は新しい ISA になりますが、ベースの ISA のすべてのレガシーバゲージを運んでいます。

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イスであり、そのような重要なインターフェイスがプロプライエタリであるべき理由はありません。主要な商用 ISA は、30 年以上前にすでによく知られていた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、オープンスタンダードのハードウェアターゲットをターゲットにすることができ、商用プロセッサ設計者は、実装品質で競うべきです。

私たちは、ハードウェア実装に適したオープンなISA設計を考えた最初の人ではありません。

我々は他の既存のオープンなISA設計も考慮しましたが、その中で最も我々の目標に近いのはOpenRISCアーキテクチャ[12]でした。いくつかの技術的な理由から、OpenRISC ISAを採用しないことにしました。

* OpenRISCには条件コードと分岐遅延スロットがあり、より高性能な実装を複雑にしています。
* OpenRISCは固定の32ビットエンコーディングと16ビットのイミディエイトを使用しているため、より高密度の命令エンコーディングができず、ISAの後の拡張のためのスペースが制限されています。
* OpenRISCはIEEE 754浮動小数点規格の2008年版をサポートしていません。
* OpenRISCの64ビット設計を開始した時点ではまだ完成していませんでした。

もちろん、最初に計画していたよりもはるかに多くの努力が必要でしたが、ゼロから始めることで、すべての目標を満たすISAを設計することができました。現在では、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、リファレンスISAシミュレータ、FPGA実装、効率的なASIC実装、アーキテクチャテストスイート、教材など、RISC-V ISAインフラストラクチャの構築に多大な労力を費やしています。このマニュアルの最終版以来、学術界と産業界の両方でRISC-V ISAの利用がかなり進んでおり、私たちはRISC-V標準を保護し、促進するために非営利のRISC-V Foundationを設立しました。RISC-Vファウンデーションのウェブサイト（[https://riscv.org）では、](https://riscv.org/)ファウンデーションのメンバーシップや、RISC-Vを利用した様々なオープンソースプロジェクトに関する最新の情報が掲載されています。

## 28.2 ISAマニュアル改訂1.0からの経緯

RISC-V ISAと命令セットのマニュアルは、いくつかの初期のプロジェクトをベースにしています。スーパバイザレベルのマシンのいくつかの側面とマニュアルの全体的なフォーマットは、1992年に始まったUCバークレーとICSIのT0 (Torrent-0) ベクトルマイクロプロセッサプロジェクトにさかのぼります。T0 は MIPS-II ISA をベースにしたベクトルプロセッサで、 Krste Asanovi'c がメインアーキテクトと RTL 設計者、Brian Kingsbury と Bertrand Irrisou が主な VLSI 実装者でした。ICSIのDavid Johnsonは、T0 ISAのデザイン、特にスーパバイザモードとマニュアルテキストに大きな貢献をしてくれました。John HauserもT0 ISAデザインについてかなりのフィードバックを提供しました。

2000年に開始されたMITのScale (低エネルギーのためのソフトウェア制御のアーキテクチャ)プロジェクトは、T0プロジェクトのインフラストラクチャをベースに構築され、スーパーバイザレベルのインターフェイスを改良し、分岐遅延スロットを削除することでMIPSスカラーISAから離れました。Ronny KrashinskyとChristopher BattenはMITのScale Vector-Threadプロセッサの主要アーキテクトであり、Mark HamptonはGCCベースのコンパイラインフラストラクチャとツールをScaleに移植しました。

2002年秋学期のMIT 6.371 Introduction to VLSI Systemsの新バージョンの授業では、T0 MIPSスカラー・プロセッサ仕様(MIPS-6371)を軽く編集したものが使用され、Chris TermanとKrste Asanovi'cが講師を務めました。Chris Termanはこのクラスのためにほとんどの実験材料を提供しました (TAはいませんでした！)。6.371クラスは、2005年春にArvindとKrste Asanovi Asanovi'cによって教えられたMITでの試験的な6.884 Complex Digital Designクラスに発展し、春のレギュラークラス6.375になりました。6.884/6.375では、SMIPSと名付けられたScale MIPSベースのスカラーISAの縮小版が使用されました。Christopher Battenはこれらのクラスの初期のTAで、SMIPS ISAをベースにしたかなりの量のドキュメントと実験資料を作成しました。この同じ SMIPS ラボ資料は、UC バークレーの 2009 年秋の CS250 VLSI システムデザインのクラスで John Wawrzynek、Krste Asanovi'c、John Lazzaro が教えていた TA Yunsup Lee によって適応され、強化されました。

Maven (Malleable Array of Vector-thread ENgines) プロジェクトは、第二世代のベクタースレッドアーキテクチャでした。その設計を担当したのは、カリフォルニア大学バークレー校の交換奨学生であったChristopher Batten氏で、2007年夏から開始されました。日立の客員研究員である青木秀隆氏は、初期のMaven ISAとマイクロアーキテクチャの設計についてかなりのフィードバックを行いました。Maven ISAはScaleのインフラストラクチャをベースにしていましたが、Maven ISAはScaleで定義されたMIPS ISAからさらに離れ、浮動小数点と整数のレジスタファイルが統一されていました。Mavenは、代替データ並列アクセラレータの実験をサポートするように設計されていました。Yunsup Leeは様々なMavenベクトル単位の主な実装者であり、Rimas Avizienisは様々なMavenスカラー単位の主な実装者でした。Yunsup LeeとChristopher Battenは、新しいMaven ISAで動作するようにGCCを移植しました。Christopher Celioは、Mavenの伝統的なベクトル命令セット(“Flood")バリアントの初期定義を提供しました。

これらの過去のプロジェクトでの経験に基づき、2010年夏、Andrew Waterman、Yunsup Lee、Krste Asanovi'c、David Pattersonを中心としたRISC-V ISA定義が開始されました。RISC-V 32ビット命令サブセットの初期バージョンがUC Berkeley 2010年秋学期CS250 VLSIシステムデザインクラスで使用され、Yunsup LeeがTAとして参加しました。RISC-Vは、以前のMIPSにインスパイアされたデザインとは一線を画しています。John Hauserは、符号注入命令や浮動小数点値の内部リコーディングを可能にするレジスタ・エンコーディング・スキームを含む浮動小数点ISA定義に貢献しました。

## 28.3 ISAマニュアル改訂2.0からの経緯

図28.1に示すように、複数のシリコンファブリケーションを含むRISC-Vプロセッサの複数の実装が完了しています。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名前 | | テープアウト日 | プロセス | ISA |
| Raven-1 | 2011年5月29日 | | ST 28nm FDSOI | RV64G1\_Xhwacha1 |
| EOS14 | 2012年4月1日 | | IBM 45nm SOI | RV64G1p1\_Xhwacha2 |
| EOS16 | 2012年8月17日 | | IBM 45nm SOI | RV64G1p1\_Xhwacha2 |
| Raven-2 | 2012年8月22日 | | ST 28nm FDSOI | RV64G1p1\_Xhwacha2 |
| EOS18 | 2013年2月6日 | | IBM 45nm SOI | RV64G1p1\_Xhwacha2 |
| EOS20 | 2013年7月3日 | | IBM 45nm SOI | RV64G1p99\_Xhwacha2 |
| Raven-3 | 2013年9月26日 | | ST 28nm SOI | RV64G1p99\_Xhwacha2 |
| EOS22 | 2014年3月7日 | | IBM 45nm SOI | RV64G1p9999\_Xhwacha3 |

表 28.1.製造されたRISC-Vテストチップ

最初に製造されたRISC-VプロセッサはVerilogで書かれ、2011年にRaven-1テストチップとしてSTの28nm FDSOI技術で製造されました。2つのコアは、Yunsup LeeとAndrew Watermanによって開発され、Krste Asanovi'cのアドバイスを受け、一緒に製造されました。1) エラー検出フリップフロップを搭載したRV64スカラーコアと、2) 64ビット浮動小数点ベクターユニットを搭載したRV64コアです。最初のマイクロアーキテクチャは、非公式には

未熟なデザインライブラリでデザインを完成させるのに利用可能な時間が短いため、“TrainWreck”と呼ばれています。

その後、Andrew Waterman、Rimas Avizienis、Yunsup Leeの3人は、Krste Asanovi'cの助言を受けて、インオーダーデカップリングRV64コアのためのクリーンなマイクロアーキテクチャを開発し、鉄道をテーマに、George Stephensonの成功した蒸気機関車の設計にちなんで「Rocket」と名付けました。

Rocketは、カリフォルニア大学バークレー校で開発された新しいハードウェア設計言語Chiselで書かれました。Rocketで使用されているIEEE浮動小数点ユニットは、John Hauser、Andrew Waterman、Brian Richardsによって開発されました。Rocketはその後さらに改良・開発され、28nm FDSOI(Raven-2, Raven-3)で2回、IBM 45nm SOI技術でフォトニクスプロジェクトのために5回(EOS14, EOS16, EOS18, EOS20, EOS22)製造されています。Rocketデザインをパラメータ化されたRISC-Vプロセッサジェネレータとして利用できるようにするための作業が進行中です。

EOS14--EOS22チップは、Yunsup Lee、Andrew Waterman、Huy Vo、Albert Ou、Quan Nguyen、Stephen Twiggが開発した64ビットIEEE浮動小数点ベクトルユニットHwachaの初期バージョンを含み、Krste Asanovi'cの助言を受けている。EOS16--EOS22チップには、ヘンリー・クックとアンドリュー・ウォーターマンが開発したキャッシュコヒーレンスプロトコルを搭載したデュアルコアが含まれており、Krste Asanovi'cが助言している。EOS14のシリコンは、1.25GHzでの動作に成功している。EOS16シリコンはIBMパッドライブラリのバグに悩まされました。EOS18とEOS20は1.35GHzでの動作に成功しました。

Ravenテストチップの貢献者は、Yunsup Lee, Andrew Waterman, Rimas Avizienis, Brian Zimmer, Jaehwa Kwak, Ruzica Jevti'c, Milovan Blagojevi'c, Alberto Puggelli, Steven Bailey, Ben Keller, Pi-Feng Chiu, Brian Richards, Borivoje Nikoli'c, and Krste Asanovi'cです。

EOSテストチップの貢献者は、Yunsup Lee, Rimas Avi\v zienis, Andrew Waterman, Henry Cook, Huy Vo, Daiwei Li, Chen Sun, Albert Ou, Quan Nguyen, Stephen Twigg, Vladimir Stojanovi'c, and Krste Asanovi'cです。

Andrew Waterman と Yunsup Lee は C++ ISA シミュレータ ”Spike" を開発しました。これは開発中の黄金モデルとして使用され、米国大陸横断鉄道の完成を祝うために使用された黄金のスパイクにちなんで名付けられました。Spike は BSD オープンソースプロジェクトとして公開されています。

Andrew Watermanは、RISC-V圧縮命令セットの予備設計で修士論文を完成させました[22]。

RISC-Vの様々なFPGA実装が完了し、主にPar Labプロジェクトの研究会での統合デモの一部として使用されています。最大のFPGAデザインは、3つのキャッシュコヒーレントRV64IMAプロセッサを搭載し、研究用オペレーティングシステムを実行しています。FPGA実装への貢献者には、Andrew Waterman、Yunsup Lee、Rimas Avizienis、Krste Asanovi'cが含まれています。

RISC-VプロセッサはUC Berkeleyのいくつかの授業で使用されています。2011年秋に開講されたCS250では、Brian Zimmer氏がTAを務め、クラスのプロジェクトの基礎としてRocketが使用されました。クリストファー・セリオは、2012年春のCS152学部の授業で、Chiselを使って教育用RV32プロセッサのスイートを作成しました。このプロセッサは、「きかんしゃトーマス」とその仲間たちが住む島にちなんで「Sodor」と名付けられました。このスイートにはマイクロコード化されたコア、パイプライン化されていないコア、2、3、5ステージのパイプライン化されたコアが含まれており、BSDライセンスで公開されています。このスイートはその後更新され、2013年春にはYunsup LeeをTAとして、2014年春にはEric LoveをTAとして、CS152で再び使用されました。Christopher CelioはBOOM (Berkeley Out-of-Order Machine)として知られるアウトオブオーダーRV64デザインを開発し、それに伴うパイプラインの可視化を行い、CS152クラスで使用しました。CS152クラスでは、Andrew WatermanとHenry Cookによって開発されたRocketコアのキャッシュコヒーレントバージョンも使用されました。

2013年の夏には、カスタム・アクセラレータをRocketコアに簡単に追加できるように、RoCC (Rocket Custom Coprocessor) インターフェースが定義されました。RocketとRoCCインターフェースは、Jonathan Bachrach氏が教えた2013年秋のCS250 VLSIクラスで幅広く使用され、いくつかの学生アクセラレータ・プロジェクトがRoCCインターフェースにビルドされました。

Hwacha ベクトルユニットが RoCC コプロセッサとして書き換えられました。

2013年春、バークレー大学の2人の学部生、Quan NguyenとAlbert OuがLinuxをRISC-V上で動作するように移植することに成功しました。

Colin Schmidt氏は、2014年1月にRISC-V 2.0用のLLVMバックエンドを成功させました。

BluespecのDarius Radは2014年3月にGCCポートにソフトフロートABIのサポートを寄稿しています。

John Hauserは、浮動小数点分類命令の定義に貢献しました。

他にもいくつかのRISC-Vコアの実装があり、Tommy Thorn氏によるVerilogでの実装や、Rishiyur Nikhil氏によるBluespecでの実装などがあります。

## 謝辞

ISAバージョン2.0仕様の草案についてコメントをいただいたChristopher F. Batten、Preston Briggs、Christopher Celio、David Chisnall、Stefan Freudenberger、John Hauser、Ben Keller、Rishiyur Nikhil、Michael Taylor、Tommy Thorn、Robert Watsonに感謝します。

## 28.4 改訂2.1からの歴史

2014年5月に凍結バージョン2.0が導入されて以来、RISC-V ISAの取り込みは非常に急速に進んでおり、このような短い歴史のセクションに記録するにはあまりにも多くの活動がありました。おそらく最も重要な単一イベントは、2015年8月に非営利のRISC-V財団が結成されたことです。このファウンデーションは現在、公式のRISC-V ISA規格のスチュワードシップを引き継ぐことになっており、RISC-V規格に関するニュースや最新情報を入手するには、公式サイトriscv.orgが最適な場所となっています。

## 謝辞

バージョン2.0仕様に関するコメントをいただいたScott Beamer、Allen J. Baum、Christopher Celio、David Chisnall、Paul Clayton、Palmer Dabbelt、Jan Gray、Michael Hamburg、John Hauserに感謝します。

**28.5 改訂2.2からの歴史**

## 謝辞

バージョン2.1の仕様についてコメントをくださったJacob Bachmeyerさん、Alex Bradburyさん、David Hornerさん、Stefan O'Rearさん、Joseph Myersさんに感謝します。

## 28.6 改訂2.3の歴史

RISC-Vの採用が猛烈な勢いで続いています。

John Hauser氏とAndrew Waterman氏は、Paolo Bonzini氏の提案に基づくハイパーバイザーISA拡張を寄稿しました。

ダニエル・ルスティグ、Arvind、Krste Asanovi'c、Shaked Flur、Paul Loewenstein、Yatin Manerkar、Luc Maranget、Margaret Martonosi、Vijayanand Nagarajan、Rishiyur Nikhil、Jonas Oberhauser、Christopher Pulte。Jose Renau, Peter Sewell, Susmit Sarkar, Caroline Trippel, Muralidaran Vijayaraghavan, Andrew Waterman, Derek Williams, Andrew Wright, and Sizhuo Zhangがメモリ一貫性モデルに貢献しました。

## 28.7 資金調達

RISC-Vアーキテクチャと実装の開発は、以下のスポンサーから一部資金援助を受けています。

* **Par Lab：**この研究は、Microsoft (Award #024263) と Intel (Award #024894) の資金援助と U.C. Discovery (Award #DIG07-10227) のマッチング・ファンディングによって支援されています。また、Par Labの関連会社であるNokia、NVIDIA、Oracle、Samsungからも支援を受けました。
* **プロジェクトIsis：**DoE賞DE-SC0003624。
* **ASPIRE Lab：**DARPA PERFECTプログラム、HR0011-12-2-0016賞。DARPA POEMプログラム、アワードHR0011-11-C-0100。Semiconductor Research Corporationが出資するSTARnetセンター、Center for Future Architectures Research (C-FAR)。ASPIRE産業スポンサーのIntel、ASPIRE関連企業のGoogle、Hewlett Packard Enterprise、Huawei、Nokia、NVIDIA、Oracle、Samsungからの追加支援。

本稿の内容は、必ずしも米国政府の立場や政策を反映したものではなく、公式な支持を示すものではありません。

**付録A**

# RVWMO説明資料、バージョン0.1

このセクションでは、RVWMO（第14章）について、より非公式な言語と具体的な例を用いて、より多くの説明を行います。これらは、公理と保存されたプログラム順序規則の意味と意図を明確にすることを目的としています。この付録は解説として扱われるべきです。すべての規範的な資料は、第14章とISA仕様の本文の残りの部分で提供されています。現在知られているすべての矛盾は、セクションA.7に列挙されています。その他の不一致は意図的なものではありません。

**A.1 なぜRVWMOなのか？**

メモリ一貫性モデルは、弱いものから強いものまで、ゆるやかなスペクトルに沿って分類されます。弱いメモリモデルは、より複雑なプログラミングモデルを犠牲にして、ハードウェア実装の柔軟性を高め、強力なモデルよりも性能、ワットあたりの性能、消費電力、スケーラビリティ、およびハードウェア検証のオーバーヘッドを向上させることができます。強力なモデルはよりシンプルなプログラミングモデルを提供しますが、パイプラインとメモリシステムで実行できる（スペックのない）ハードウェア最適化の種類に制限を課し、その代償として、電力、面積オーバヘッド、および検証負担の点でコストがかかります。

RISC-Vでは、リリース一貫性のあるRVWMOメモリモデルを選択しました。これにより、メモリモデルのスペクトルの両極端の間に位置しています。RVWMOメモリモデルは、アーキテクトがシンプルなインプリメンテーション、アグレッシブなインプリメンテーション、より大規模なシステムに深く組み込まれたインプリメンテーション、複雑なメモリシステムの相互作用を受けるインプリメンテーション、その他多くの可能性を構築することを可能にし、同時にプログラミング言語のメモリモデルをハイパフォーマンスでサポートするのに十分な強度を持っています。

他のアーキテクチャからのコードの移植を容易にするために、一部のハードウェア・インプリメンテーションでは、Ztso拡張機能を実装することを選択することができ、デフォルトではより厳格なRVTSO命令セマンティクスを提供します。RVWMO用に書かれたコードは自動的にRVTSOと本質的に互換性がありますが、RVTSOを想定して書かれたコードはRVWMOインプリメンテーションで正しく動作することは保証されていません。実際、ほとんどのRVWMOの実装では、RVTSOのみのバイナリの実行を拒否します（そして拒否すべきです）。

そのため、各実装は、RVTSOコードとの互換性を優先するか（例えば、x86からの移植を容易にするため）、代わりにRVWMOを実装している他のRISCVコアとの互換性を優先するかを選択する必要があります。

RVWMO用に書かれたコードの中には、RVTSOでは冗長になるフェンスやメモリ順列アノテーションがあります。RVWMOのデフォルトがZtso実装に課すコストは、それらのフェンス（例えば、FENCE R,RWやFENCE RW,W）を取得するための増分オーバーヘッドです。しかし、Ztso 以外の実装との互換性が望まれる場合には、これらのフェンスはコードの中に残っていなければなりません。

## A.2 リトマス試験

この章の説明では、リト*マステスト*、つまり、記憶モデルのある特定の側面をテストしたり、強調したりするように設計された小さなプログラムを使用します。図A.1は、2つのハートを持つリトマス・テストの例を示している。この図とこの章で続くすべての図の慣例として、s0-s2 はすべてのハートで同じ値にあらかじめ設定されており、s0 は x、s1 は y、s2 は z とラベル付けされたアドレスを保持していると仮定します。各図は、左がリトマステストコード、右が有効または無効な実行の可視化を示しています。

ハート1

ハート 0

.

.

.

.

.

.

li t1,1

li t4,4

0)

a) sw t1,0(s

(

0)

e) sw t4,0(s

(

.

.

.

.

.

.

li t2,2

(

0)

b) sw t2,0(s

.

.

.

.

.

.

(

0)

c) lw a0,0(s

.

.

.

.

.

.

li t3,3

li t5,5

(

0)

d) sw t3,0(s

(

f) sw t5,0(s

0)

.

.

.

.

.

.

a:Wx=1

b:Wx=2

c:Rx=1

d:Wx=3

co

rfＦ

co

fr

fr

fr

e:Wx=4

co

f:Wx=5

co

図A.1.サンプルのリトマス・テストと1つの禁止された実行(a0=1)。

リトマステストは、特定の具体的な状況におけるメモリモデルの意味合いを理解するために使用されます。例えば、図A.1のリトマステストでは、最初のハートのa0の最終値は、実行時の各ハートからの命令ストリームの動的なインターリーブに応じて、2、4、または5のいずれかになります。しかし、この例では、ハート0のa0の最終値は1や3になることはありません。直感的には、ロードが実行された時点では値1はもはや見えず、ロードが実行された時点では値3はまだ見えません。このテストと他の多くのテストを以下で分析します。

各リトマス試験の右側にある図は、検討されている特定の実行候補を視覚的に表しています。これらの図は、メモリモデルの分野では一般的な表記法で、問題となる実行を行う可能性のあるグローバルメモリオーダーのセットを制約するために使用されています。これは、付録B.2で紹介されている群れのモデルの基礎でもあります。

|  |  |
| --- | --- |
| Edge | 全名前（および説明） |
| rf | Reads From (各ストアから、そのストアが書き込んだ値を返すロードまで) |
| co | コヒーレンス（各アドレスへのストアのトータルオーダー） |
| fr | フロムリード（各ロードから、ロードが値を返したストアの共同サクセサまで） |
| ppo | 保存されたプログラムの順序 |
| fence | FENCE命令で施行される命令 |
| addr | アドレス依存性 |
| ctrl | 制御依存性 |
| data | データ依存性 |

表A.1.この付録に描かれたリトマス試験図のキー

この表記法は表A.1で説明されています。リストアップされた関係のうち，ハート間のrfエッジ，coエッジ，frエッジ，ppoエッジは，グローバルなメモリ順序を直接制約します（ppoを介したfence，addr，data，およびいくつかのctrlエッジも同様）。その他のエッジ（ハート内のrfエッジなど）は情報を提供しますが、グローバルなメモリ順序を制約するものではありません。。

例えば、図A.1では、以下のいずれかが真である場合にのみ、a0=1が発生する可能性がある。

* (b)は、グローバルメモリ順序では(a)の前に現れる(コヒーレンス順序coでも)。しかし、これはRVWMO PPO規則1に違反する。(b)から(a)へのcoエッジは、この矛盾を浮き彫りにしています。
* (a)は、グローバルメモリ順序では(b)よりも前に現れる(コヒーレンス順序coでも)。しかし、この場合、(a)はプログラム順序では(c)より前の最新の一致ストアではないので、負荷値公理に違反することになる。(c)から(b)へのfrエッジは、この矛盾を浮き彫りにしている。

これらのシナリオはどちらもRVWMO公理を満たさないので、結果a0=1は禁止されています。

この付録に記載されている以外にも、7,000以上のリトマス試験が <https://github.com/litmus-tests/litmus-tests-riscv>で利用可能である[。](https://github.com/litmus-tests/litmus-tests-riscv)

*また、リトマステストのリポジトリでは、RISC-Vハードウェア上でのリトマステストの実行方法や、運用モデルや公理モデルとの比較方法についても説明しています。*

*将来的には、これらのメモリモデルリトマステストをRISC-Vコンプライアンステストスイートの一部として使用できるようにすることを期待しています。*

## A.3 RVWMO規則の説明

ここでは、RVWMOのすべての規則と公理についての説明と例を示します。

### A.3.1 プリザーブドプログラムオーダーとグローバルメモリオーダー

保存されたプログラム順序は、グローバルなメモリ順序の中で尊重されなければならないプログラム順序のサブセットを表します。概念的には、保存されたプログラム順序によって並べられた同じハートからのイベントは、他のハートと／もしくは観察者の視点からはその順序で表示されなければなりません。

一方、同じハートからのイベントで、保存されたプログラムの順序ではないものは、他のハートと／もしくは観察者の視点では、順序が変わって見えることがあります。

非公式には、グローバルメモリの順序は、ロードとストアが実行される順序を表します。形式的なメモリモデルの文献では、実行の概念を中心とした仕様から遠ざかっていますが、非公式な直観を構築するためには、この考え方は今でも有用です。ロードは、その戻り値が決定されたときに実行されたと言われます。ストアは、パイプライン内で実行されたときではなく、その値がグローバルに見えるメモリに伝播されたときにのみ実行されたと言われています。この意味で、グローバルメモリ順序は、コヒーレンスプロトコルおよび／またはメモリシステムの残りの部分の貢献を表し、各ハートによって発行される（再順序化される可能性のある）メモリアクセスを、すべてのハートによって合意された単一の合計順序にインターリーブすることを意味します。

ロードが実行される順番は、2 つのロードが返す値の相対的な年齢に必ずしも直接対応しているわけではありません。特に、ロード *b は*同じアドレスの別のロード *a の*前に実行されることがありますが（すなわち、*b は a の*前に実行され、グローバルメモリの順序では b *が a の*前に現れることがあります）、*a は b* よりも古い値を返すことがあります。この不一致は、コアとメモリの間に配置されたバッファリングの並び替え効果を（とりわけ）捉えています。例えば、*b は*ストアバッファ内のあるストアから値を返しているかもしれませんが、*a は*その若いストアを無視して、代わりにメモリから古い値を読み込んでいるかもしれません。これを考慮するために、各ロードが実行される時点で、それが返す値は、以下で説明するように、グローバルメモリの順序で同じアドレスへの最新のストアを厳密に決定するだけでなく、ロード値の公理によって決定されます。

### A.3.2 ロード値の公理

ロード値の公理：各ロード*iの各*バイトは、次のストアのうちグローバルメモリ順で最も新しいストアによってそのバイトに書き込まれた値を返します。

1. そのバイトを書き込み、グローバルメモリの順序で *i* より前のバイトを格納します。
2. そのバイトを書き込み、プログラム順に *i* より前のバイトを格納します。

保存されたプログラムの順序は、重複するアドレスへのロードに続くストアの順序を尊重する必要はあり*ません。この*複雑さは、ほぼすべての実装でストアバッファがユビキタスであることに起因しています。非公式には、ロードは、ストアがストアバッファに残っている間に、ストアから転送することによって実行（値を返す）することができ、したがって、ストア自体が実行（グローバルに見えるメモリに書き戻す）する前に実行することができます。したがって、他のどのハートも、ストアの前にロードが実行されていると見なします。

図A.2のリトマス・テストを考えてみます。このプログラムをストアバッファを持つ実装で実行すると、最終的にa0=1、a1=0、a2=1、a3=0という結果を以下のように得ることができます。

* (a)を実行し、最初のハートのプライベートストアバッファに入ります。
* (b)を実行し、(a)からの戻り値1をストアバッファに転送します。
* (c)を実行し、以前のすべてのロード(すなわち、(b)が完了してから実行されます。

ハート 0

ハート1

li t1, 1

li t1, 1

a) sw t1,0(s

0)

(

e) sw t1,0(s)

1)

(

b) lw a0,0(s

0)

(

f) lw a2,0(s)

1)

(

c**)** fence r,r

(

g) fence r,r

(

d) lw a1,0(s)

1)

(

h) lw a3,0(s)

0)

(

a:Wx=1

b:Rx=1

d:Ry=0

f:Ry=1

rfＦ

fence

ppo

fr

fence

ppo

frr

h:Rx=0

e:Wy=1

結果: a0=1, a1=0, a2=1, a3=0

図A.2: ストアバッファ転送リトマステスト(結果が許可されている)

* (d)を実行し、メモリから値 0 を読み込みます。
* (e)を実行し、2番目のハートのプライベートストアバッファに入ります。
* (f)を実行し、(e)からの戻り値1をストアバッファに転送します。
* (g)を実行し、以前のすべてのロード(すなわち、(f))が完了してから実行されます。
* (h)を実行し、メモリから値 0 を読み込みます。
* (a) は最初のハートのストアバッファからメモリに排出します。
* (e) は第２ハートのストアバッファからメモリに排出します

したがって、メモリモデルは、この動作を考慮することができなければなりません。

別の言い方をすれば、保存されたプログラム順の定義に次のような仮定の規則が含まれていたとします: 保存されたプログラム順ではメモリアクセス*aがメモリアクセスb*よりも優先されます (したがって、グローバルメモリ順でも優先されます) もしプログラム順で*aがb*よりも優先され、*a*と*bが*同じメモリ位置へのアクセスであれば、*aは*書き込み、*bは*読み出しとなります。これを“規則X"と呼びます。すると次のようになります。

* (a)は(b)に先行：規則Xによります
* (b)は(d)に先行：規則4によります
* (d)は(e)に先行: ロード値の公理により。そうでなければ、(e)が(d)に先行していた場合、(d)は値1を返す必要があります。(これは完全に合法的な実行であり、問題のある実行ではありません)
* (e)は(f) に先行: 規則 X によります
* (f)は(h)に先行: 規則4によります
* (h) は (a) に先行：上記のように負荷値公理によります。

グローバルメモリの順序は全体の順序でなければならず、周期的であってはなりません、 なぜなら、サイクルは、サイクル内のすべてのイベントが自分自身の前に起こることを意味し、それは不可能だからです。

したがって、上記で提案された実行は禁止されており、したがって、規則Xを追加すると、ストア・バッファ・フォワーディングを使用した実装が禁止されることになり、これは明らかに望ましくありません。

それにもかかわらず、グローバルメモリの順序で(b)が(a)および/または(f)が(e)に先行していたとしても、この例では(b)が(a)によって書かれた値を返し、(f)と(e)も同様に(a)によって書かれた値を返すことが唯一の賢明な可能性です。このような状況の組み合わせが、負荷値公理の定義における2番目の選択肢につながります。グローバルメモリの順序で(b)が(a)より先行していても、(a)は(b)が実行された時にストアバッファにあるので、(a)は(b)にはまだ見えます。したがって、(b)がグローバルメモリの順序で(a)より先行していたとしても、(a)がプログラムの順序で(b)より先行しているので、(b)は(a)に書かれた値を返すべきです。(e)と(f)についても同様です。

ハート 0

ハート1

li t1, 1

li t1, 1

(

a) sw t1,0(s)

0)

LOOP:。

b) fence w,w

(

d) lw a0,0(s)

1)

(

(

1)

c) sw t1,0(s)

BEQZ A0, LOOP

(

2)

e) sw t1,0(s)

(

2)

f) lw a1,0(s)

xor a2,a1,a1

add s0,s0,a2

g) lw a2,0(s)

(

0)

a:Wx=1

c:Wy=1

d:Ry=1

e:Wz=1

f:Rz=1

g:Rx=0

fence

ppo

rfＦ

ctrl

ppo

ctrl

ctrl

rf

addr

ppo

fr

結果：a0=1, a1=1, a2=0

図A.3: “PPOCA"ストアバッファ転送リトマステスト(結果は許可される)

ストア・バッファの動作を強調する別のテストを図A.3に示します。この例では、(d)は制御依存性のために(e)の前に順序付けられ、(f)はアドレス依存性のために(g)の前に順序付けられています。しかし、(e)は、(f)が(e)によって書かれた値を返すからといって、必ずしも(f)の前に順序付けられているわけ*ではありません。*これは、次のような順序に対応する可能性があります。

* (e)は、投機的に実行され、2 番目のハートのプライベートストアバッファに入ります (ただし、メモリに排出しません)。
* (f)は、投機的に実行され(e)からの戻り値1をストアバッファに転送します。
* (g)は、投機的に実行し、メモリから値0を読み込みます。
* (a)が実行され、最初のハートのプライベートストアバッファに入り、メモリに排出します。
* (b)が実行と退任
* (c)が実行され、最初のハートのプライベートストアバッファに入り、メモリに排出します。
* (d)を実行し、メモリから値1を読み込みます。
* (e)、(f)、(g)は、推測が正しいことが判明したのでコミットします。
* (e)ストアバッファからメモリに排出します。

### A.3.3 アトミック性の公理

アトミック性公理（アラインド・アトミックのための）：*r*と*wがハー* ト内のアラインメントされたLR命令とSC命令によって生成されたペアのロードとストア操作であり、*sが*バイト*x*へのストアであり、*rがs*によって書き込まれた値を返すとすると、グローバルメモリの順序では*sがwより前に*なければならず、グローバルメモリの順序では*hart*以外のハー トからバイト*x*へのストアは*s*に続いて*wより*前に存在してはなりません。

RISC-Vアーキテクチャは、アトミック性の概念を順序の概念から切り離します。TSOのようなアーキテクチャとは異なり、RVWMOの下でのRISC-Vアトミックは、デフォルトでは順序付けの要件を課していません。秩序化セマンティクスは、それ以外の場合は適用されるPPO規則によってのみ保証されます。

RISC-Vには2種類のアトミックが含まれています。AMOとLR/SCペアです。これらは概念的には以下のように異なる振る舞いをします。LR/SCは、古い値がコアに持ち込まれ、修正され、メモリに書き戻され、そのメモリ上に予約が保持されているかのように振る舞います。一方、AMOは概念的には、あたかもメモリ内で直接実行されているかのように振る舞います。したがって AMO は本質的にアトミックであるが、LR/SC ペアはアトミックであるが、元のハートが予約を保持している間は、問題のメモリ位置が他のハートによって変更されることはないという若干異なる意味でのアトミックです。

|  |  |  |
| --- | --- | --- |
| 1. lr.d a0, 0(s0) 2. sd t1, 0(s0) 3. sc.d t2, 0(s0) | 1. lr.d a0, 0(s0) 2. sw t1, 4(s0) 3. sc.d t2, 0(s0) | 1. lr.w a0, 0(s0) 2. sw t1, 4(s0) 3. sc.w t2, 8(s0) |

|  |
| --- |
| 1. lr.w a0, 0(s0) 2. sw t1, 4(s0) 3. sc.w t2, 0(s0) |

図A.4: 4つの(独立した)コードスニペットすべてにおいて、格納条件付き(c)は許可されていますが、成功は保証されていません。

アトミック性公理は、他のハートからのストアが、ある LR とその LR と対になっている SC の間でグローバルメモリ順にインタリーブされることを禁止します。アトミック性公理は、プログラム順またはグローバルメモリ順のペアリングされた操作の間にロードがインターリーブされることを禁じるものではなく、また、プログラム順またはグローバルメモリ順のペアリングされた操作の間に、同じハートからのストアや重ならない位置へのストアが現れることを禁じるものでもありません。例えば、図A.4のSC命令は成功する可能性がある(しかし保証されていません)。これらの成功はいずれもアトミック性公理に違反することはありません。なぜなら、間に介在する非条件ストアは、ペアになったロードリザーブ命令とストア条件付き命令と同じハートからのものだからです。これにより、キャッシュライン単位でメモリアクセスを追跡するメモリシステム（したがって、図A.4の4つのスニペットは同一のものとみなされます）は、予約で保持されているメモリロケーションと同じキャッシュラインの別の部分を（誤って）共有している保存条件付き命令を失敗させることはありません。

アトミック性公理は、LRとSCが異なるアドレスに触れたり かつ／もしくは 異なるアクセスサイズを使用したりする場合も技術的にはサポートしているが、そのような動作の使用例は実際には稀であると予想されます。

同様に、LR／SCのペアの間の同じハートからのストアが、LRやSCが参照するメモリ位置に実際に重なるシナリオは、介在するストアが単に同じキャッシュライン上に落ちる可能性があるシナリオに比べて、稀であると予想されます。

### A.3.4 進歩の公理

進捗公理：グローバルメモリの順序では、他のメモリ操作の無限のシーケンスに先行するメモリ操作はありません。

プログレス公理は、最小限の前進保証を保証します。これは、あるハートからのストアが最終的に有限の時間内にシステム内の他のハートから見えるようになり、他のハートからのロードが最終的にそれらの値(またはその後継)を読み取ることができることを保証します。この規則がなければ、例えば、スピンロックが値に対して無限にスピンすることは、他のハートからのストアがスピンロックのロックを解除するのを待っている状態であっても、合法となります。

プログレス公理は、RISC-V実装のHARTにフェアネス、レイテンシ、サービス品質の他の概念を押し付けることを意図していません。より強い公平性の概念は、ISAの残りの部分、および/またはプラットフォームやデバイスの定義と実装に委ねられています。

前進公理は、ほとんどの場合、標準的なキャッシュコヒーレンスプロトコルによって自然に満たされます。非コヒーレントキャッシュを持つ実装では、すべてのストア（またはその後継）の最終的な可視性をすべてのハートに保証するために、他のメカニズムを提供しなければならないかもしれません。

### A.3.5 重複アドレスの順序付け (規則1-3)

規則 1: *b は*ストア、*a* と b *は*重複するメモリアドレスにアクセス

規則 2: *a* と b *は*ロード、*x は a* と *b の*両方によって読み込まれたバイト、プログラムの順序で *a* と *b の*間に *x* へのストアはなく、*a* と b *は*異なるメモリ操作によって書き込まれた *x の*値を返します。

規則3: *aは*AMO命令またはSC命令によって生成され、*bは*ロードであり、*bはa*によって書き込まれた値を返します。

後者がストアである場合の同一アドレスの順序付けは簡単である: ロードやストアは、オーバーラップするメモリ・ロケーションへの後のストアと一緒に再順序付けすることはできない。マイクロアーキテクチャの観点から、一般的に言えば、推測で並べ替えられたストアを元に戻すことは、推測が無効であることが判明した場合には困難であるか、不可能であるため、そのような動作はモデルによって単純に禁止されています。一方、ストアから後の負荷への同一アドレスオーダーは、強制される必要はありません。セクションA.3.2で議論されているように、これはバッファリングされたストアから後のロードに値を転送する実装の観測可能な動作を反映しています。

同一アドレスのロード-ロードの順序付けの要件は、はるかに微妙です。基本的な要件は、若いロードが、同じハー ト内の古いロードが同じアドレスに返す値よりも古い値を返してはいけないということです。これは、「CoRR」（Coherence for Read-Read Pair）と呼ばれることが多く、また、より広い意味での「コヒーレンス」や「ロケーションごとのシーケンシャルな一貫性」の要件の一部としても知られています。過去のいくつかのアーキテクチャでは、同一アドレスのロードロード順序を緩和していましたが、今になってみると、これはプログラミングモデルを複雑にしすぎると一般的に考えられているため、RVWMOではCoRR順序を強制する必要があります。

しかし、グローバルなメモリ順序は、返される値の順序ではなく、ロードが実行される順序に対応しているため、グローバルなメモリ順序でCoRRの要件を捉えるには、少し間接的な作業が必要になります。

|  |  |
| --- | --- |
| li t1, 1 | li t2, 2 |
| (a) sw t1,0(s0) | (d) lw a0,0(s1) |
| (ｂ）fence ｗ、ｗ | (e) sw t2,0(s1) |
| (c) sw t1,0(s1) | （ｆ） lw a1,0(s1)  (g) xor t3,a1,a1  (h) add s0,s0,t3  (i) lw a2,0(s0) |

a:Wx=1

c:Wy=1

d:Ry=1

e:Wy=2

f:Ry=2

i:Rx=0

fence

ppoぺい

rfＦ

co

fr

ppo

rfＦ

addr情報

ppoぺい

fr

ハート 0 ハート 1

結果：a0=1, a1=2, a2=0

図A.5: リトマステスト MP+fence.w.w.w+fri-rfi-addr (結果許可)

図A.5のリトマステストを考えてみます。これはより一般的な “fri-rfi"パターンの特定の例です。”fri-rfi"とは、(d), (e), (f)のシーケンスを指します。(d) “from-reads"(すなわち、同じハートである(e)よりも前の書き込みから読み出すこと)、そして(f)は同じハートにある(e)から読み出すことである。

マイクロアーキテクチャの視点から見ると、結果a0=1, a1=2, a2=0は合法である（他の様々な微妙な結果と同様に）。直感的に言えば、次のようなことが問題の結果を生み出すでしょう。

* (d) ストール(何らかの理由で、他の先行する命令を待つために停滞しているのかもしれません。)
* (e) 実行してストアバッファに入る(ただしメモリに排出しない)
* (f) ストアバッファの(e)から実行して転送します。
* (g)、(h)、および(i)を実行します。
* (ａ）メモリへの実行と排出、（ｂ）実行、（ｃ）メモリへの実行とドレイン
* (d) アンストールして実行
* (e) ストアバッファからメモリにドレイン

これは、(f)、(i)、(a)、(c)、(d)、(e)のグローバルメモリ順序に対応します。(f)が(d)の前に実行しても、(f)が返す値は(d)が返す値よりも新しいことに注意してください。したがって、この実行は合法であり、CoRRの要件に違反しません。

同様に、2つの連続したロードが同じストアで書き込まれた値を返す場合、CoRRに違反することなく、グローバルメモリの順序が乱れていることもあります。2つの異なるストアが同じ値を書き込む可能性があるため、2つのロードが同じ値を返すという意味ではないことに注意してください。

図A.6のリトマス試験を考えてみましょう。a0=1, a1=v, a2=v, a3=0 (vは他のハートが書いた何らかの値)という結果は、(g)と(h)を並び替えさせることで観察できます。

ハート 0

ハート1

li t1, 1

(d) lw a0,0(s1)

(a) sw t1,0(s0)

(e) xor t2,a0,a0

(b) fence w, w

(f) add s4,s2,t2

(c) sw t1,0(s1)

(g) lw a1,0(s4)

(h) lw a2,0(s2)

(i) xor t3,a2,a2

(j) add s0,s0,t3

(k) lw a3,0(s0)

a:Wx=1

c:Wy=1

d:Ry=1

g:Rz=

*v*

h:Rz=

*v*

k:Rx=0

Wz=

*v*

fence

ppo

rf

addr

ppo

po

addr情報

ppo

fr

rfＦ

rf

結果：a0=1、a1=*v*、a2=*v*、a3=0

図A.6：リトマス試験RSW （結果許可）

これは投機的に行われる可能性があり、（g）の後に（h）を再生するととにかく同じストアによって書き込まれた値が返されるため、マイクロアーキテクチャによって（たとえば、キャッシュの無効化をスヌーピングして何も検出しないことによって）推測を正当化できます。したがって、a1とa2がいずれにせよ同じストアによって書き込まれた同じ値で終わると仮定すると、(g)と(h)は合法的に並び替えることができます。この実行に対応するグローバルメモリの順序は、(h),(k),(a),(c),(d),(g)となります。

a1がa2と等しくない図A.6のテストを実行する場合、実際にはグローバルメモリの順序で(g)が(h)の前に現れることを必要とします。(h)が(g)の前に現れることを許可すると、(h)は(g)が返す値よりも古い値を返すことになるため、CoRR違反になります。したがって、PPO規則2はこのCoRR違反の発生を禁止しています。このように、PPO規則2は、すべてのケースでCoRRを強制すると同時に、実際のマイクロアーキテクチャでよく見られる “RSW"と ”fri-rfi"パターンを許可するのに十分弱いというバランスをとっています。

もう一つオーバーラップアドレス規則があります。PPO規則3は、AMOやSCがグローバルに実行されるまで（SCの場合は正常に実行されるまで）、AMOやSCから後続のロードに値を返すことはできないと簡単に述べています。これは、AMO命令とSC命令の両方がメモリ内でアトミック的に実行されることを意味しているという概念的な見解から、多少は自然に従うことになります。しかし、注目すべきことに、PPO規則3では、他のAMOの場合のように、AMOSWAPの場合は、ストア値がメモリ内の前の値にセマンティックに依存していなくても、ハードウェアは、AMOSWAPによってストアされている値を後続のロードに転送してはならないと規定されています。これは、ペアになった LR が返す値に意味的に依存しない SC のストア値を転送する場合でも同様です。

上記の3つのPPO規則は、問題のメモリアクセスが部分的にしか重ならない場合にも適用されます。これは、例えば、異なるサイズのアクセスが同じオブジェクトにアクセスするために使用される場合などに発生します。また、2つのメモリアクセスが重なるためには、重なる2つのメモリ操作のベースアドレスが必ずしも同じである必要はないことにも注意してください。誤ったメモリアクセスが使用されている場合、オーバーラップアドレスPPO規則は、各コンポーネントのメモリアクセスに独立して適用されます。

### A.3.6 フェンス（規則4）

規則4: *bの*前に*aを*オーダーするFENCE命令があります。

デフォルトでは、FENCE命令はプログラム順にフェンスの前にある命令（“前任者セット”）からの メモリアクセスが、プログラム順にフェンスの後にある命令（“後任者セット”）からの メモリアクセスよりもグローバルメモリ順に早く現れるようにしています。しかし、フェンスはオプションで前任者セットおよび/または後任者セットをさらに制限することができ、ある程度の高速化を提供するために、より小さなメモリアクセスのセットに制限することができます。具体的には、フェンスは、前任者セットおよび／または後任者セットを制限するＰＲ、ＰＷ、ＳＲ、およびＳＷビットを有しています。前置セットは、PR（resp.PW）がセットされている場合にのみ、ロード（resp.ストア）を含みます。同様に、後続セットは、ＳＲ（resp.ＳＷ）がセットされている場合にのみ、ロード（レスポンシブストア）を含みます。

FENCEエンコーディングには現在、PR、PW、SR、SWの4つのビットの自明でない9つの組み合わせに加えて、「取得+解放」またはRVTSOセマンティクスのマッピングを容易にする1つの追加エンコーディングFENCETSOがあります。残りの7つの組み合わせは、前任者および／または後任者のセットが空であるため、no-opとなります。非自明な10のオプションのうち、実際によく使われるのは6つだけです：。

* FENCE RW,RW
* FENCE.TSO
* FENCE RW,W
* FENCE R,RW
* FENCE R,R
* FENCE W,W

PR、PW、SR、SWの他の組み合わせを使ったFENCE命令は予約済みです。私たちは、プログラマーがこの6つの選択肢にこだわることを強く推奨します。他の組み合わせでは、メモリモデルとの間に未知の、あるいは予期しない相互作用が生じる可能性があります。。

最後に、RISC-Vはマルチコピーアトミックメモリモデルを使用しているため、プログラマはスレッドローカルな方法でフェンスビットを推論することができます。マルチコピーアトミックではないメモリモデルに見られるような「フェンスの累積性」という複雑な概念はありません。

### A.3.7 明示的な同期化 (規則5～8)

規則5: *a*は取得注釈を持っています。

規則6: *b*は解放注釈があります。

規則7: *a* と *b は*共に RCsc 注釈を持ちます。

規則 8: *a は b* とペアになっています。

クリティカル・セクションの開始時に使用されるような*取得*操作では、プログラム・オーダーでの取得に続くすべてのメモリ操作が、グローバル・メモリ・オーダーでの取得にも追従する必要があります。これにより、例えば、クリティカルセクション内のすべてのロードとストアが、保護のために使用される同期変数に関して最新であることが保証されます。取得の順序は、同期変数のみを対象とした取得アノテーションと、以前のすべてのロードを対象とした FENCE R,RW の 2 つの方法のうち、いずれかの方法で行うことができます。

sd x1, (a1) # 任意の無関係なストア

ld x2, (a2) # 任意の無関係なロード

li t0, 1 # スワップ値の初期化

again:

amoswap.w.aq t0, t0, (a0) # ロック取得の試み.

bnez t0, again # 保持されている場合は再試行.

# ...

# クリティカルセクション。

# ...

amoswap.w.rl x0, x0, (a0) # 0を格納することでロックを解除.

sd x3, (a3) # 任意の無関係なストア

ld x4, (a4) # 任意の無関係なロード

図A.7: アトミックを用いたスピンロック

図 A.7 を考えてみましょう。この例では*aqを*使用しているので、クリティカルセクションのロードとストアは、ロックを取得するために使用されたAMOSWAPの後にグローバルメモリの順番で現れることが保証されています。しかし、a0, a1, a2が異なるメモリ位置を指していると仮定すると、クリティカルセクションのロードとストアは、この例の最初にある“任意の無関係なロード”の後にグローバルメモリの順序で現れるかもしれないし、現れないかもしれません。

sd x1, (a1) # 任意の無関係なストア

ld x2, (a2) # 任意の無関係なロード

li t0, 1 # スワップ値の初期化.

again:

amoswap.w t0, t0, (a0) # ロック取得の試み.

fence r, rw # 「取得」メモリオーダリングを適用する

bnez t0, again # 保持されている場合は再試行.

# ...

# Critical section.

# ...

fence rw, w # Enforce "release" memory ordering

amoswap.w x0, x0, (a0) # Release lock by storing 0.

sd x3, (a3) # 任意の無関係なストア

ld x4, (a4) # 任意の無関係なロード

図A.8: フェンスのあるスピンロック

さて、図A.8の代替案を考えてみましょう。この場合、AMOSWAPは*aq*ビットでの順序付けを強制しませんが、フェンスはそれにもかかわらず、クリティカルセクションのすべてのロードやストアよりも、グローバルメモリの順序の中で、 取得されたAMOSWAPの方が早く現れることを強制しています。しかし、この場合、フェンスは追加の順序付けも強制します。プログラムの開始時の “任意の無関係なロード"が、クリティカルセクションのロードやストアよりもグローバルメモリの順序で早く現れることも要求します。(ただし、このフェンスは断片の最初にある ”任意の無関係なストア''に関しては、いかなる順序も強制しません)。このように、フェンスで強制された秩序は、.aqで強制された秩序よりもわずかに粗くなります。

リリース命令は、取得命令と全く同じように動作しますが、逆方向に動作します。リリースセマンティクスでは、プログラム順序でリリース操作に先行するすべてのロードとストアが、グローバルメモリ順序でリリース操作に先行することを要求します。これにより、例えば、クリティカルセクションのメモリアクセスがグローバルメモリオーダーのロック解放ストアの前に現れることが保証されます。取得セマンティクスと同様に、リリースセマンティクスもリリースアノテーションを使用したり、FENCE RW,W オペレーションを使用して強制することができます。同じ例を使って、クリティカルセクションのロードとストアとコード断片の最後にある “任意の無関係なストア"の間の順序は、図A.7の*rl*ではなく、図A.8のFENCE RW,Wによってのみ強制されます。

RCpc のアノテーションだけでは、ストア-解放からロード-取得 の順序は強制されません。これにより、TSOおよび/またはRCpcメモリモデルで書かれたコードの移植が容易になります。ストア-解放からロード-取得 順序を強制するには、PPO規則7が適用されるように、コードは ストア-解放-RCscとロード-取得-RCsc操作を使用しなければなりません。RCpcだけでもC/C++の多くのユースケースでは十分ですが、いくつかの例を挙げると、C/C++、Java、Linuxの他の多くのユースケースでは不十分です；詳細はセクションA.5を参照してください。

PPO規則8では、グローバルメモリの順序では、対になっているLRの後にSCが出現しなければならないことが示されています。このことは、固有のデータ依存性により、アトミック的な読み出し変更書き込み操作を行うためにLR/SCが一般的に使用されていることから、当然のことながら従うことになる。しかし、PPO規則8は、格納される値が対になったLRが返す値に構文的に依存しない場合にも適用されます。

最後に、フェンスと同じように、プログラマーは順序付けアノテーションを解析する際に “累積性"を気にする必要はないことに注意してください。

### A.3.8 構文依存性 (規則9--11)

規則9: *bはaに構文上のアドレス依存性があります。*

規則10: *bは、aの構文データ依存性を持っています。*

規則11: *bはストアであり、bはaに構文制御の依存関係があります。*

同じハート内のロードから後のメモリ操作への依存関係は、RVWMOメモリモデルでは尊重されます。アルファ・メモリ・モデルは、このような依存関係の順序を強制し*ないこと*で注目されていましたが、最新のハードウェアおよびソフトウェアのメモリ・モデルのほとんどは、依存性のある命令の順序を変更することを許可することは、あまりにも混乱を招き、直観的ではないと考えています。さらに、現代のコードでは、特に軽量な順序を強制するメカニズムとして、このような依存関係を意図的に使用することがあります。

第14.1節の用語は以下のように動作します。各デスティネーションレジスタに書き込まれる値がソースレジスタの関数であるときはいつでも、命令はソースレジスタからデスティネーションレジスタへの依存関係を運ぶと言われています。ほとんどの命令では、これは宛先レジスタがすべてのソースレジスタからの依存関係を運ぶことを意味します。しかし、いくつかの顕著な例外があります。メモリ命令の場合、宛先レジスタに書き込まれる値は、ソース・レジスタから直接ではなく、最終的にメモリ・システムから来るので、ソース・レジスタから運ばれる依存関係の連鎖を断ち切ることになります。無条件ジャンプの場合、デスティネーションレジスタに書き込まれる値は、現在のpc（メモリモデルではソースレジスタとみなされることはありません）から来るので、同様にJALR（ソースレジスタを持つ唯一のジャンプ）ではrs1からrdへの依存関係はありません。

1. fadd f3,f1,f2
2. fadd f6,f4,f5
3. csrrs a0,fflags,x0

図A.9: (c)は、(a)と(b)の両方が暗黙のうちにfflagsに蓄積される宛先レジスタを介して、(a)と(b)の両方に構文的に依存しています。

デスティネーション・レジスタに書き込むのではなく、デスティネーション・レジスタに蓄積するという概念は、fflagsなどのCSRの動作を反映しています。特に、あるレジスタへの蓄積は、以前の書き込みや同じレジスタへの蓄積を妨害することはありません。例えば、図A.9では、(c)は(a)と(b)の両方に構文的に依存しています。

他の最新のメモリ・モデルと同様に、RVWMOメモリ・モデルは意味的な依存関係ではなく、構文的な依存関係を使用しています。言い換えれば、この定義は、異なる命令がアクセスするレジスタの同一性に依存しており、それらのレジスタの実際の内容ではありません。これは、たとえ計算が一見「最適化されていない」ように見えても、アドレス、制御、またはデータの依存関係が強制されなければならないことを意味しています。この選択により、RVWMOは、軽量な順序付けメカニズムとしてこれらの偽の構文依存関係を使用するコードとの互換性を確保しています。

ld a1,0(s0) xor a2,a1,a1 add s1,s1,a2 ld a5,0(s1)

図A.10: 構文アドレス依存性

例えば、図A.10の最初の命令によって生成されたメモリ操作から最後の命令によって生成されたメモリ操作まで、a1 XOR a1がゼロであり、それゆえに第2のロードによってアクセスされるアドレスに影響を与えないにもかかわらず、構文的なアドレス依存性が存在します。

軽量同期メカニズムとして依存性を使用する利点は、順序付けの強制要件が問題の特定の2つの命令にのみ限定されることです。他の依存性のない命令は、積極的な実装によって自由に順序を変えられるかもしれません。一つの代替案はロード取得を使用することですが、これは後続の*すべての*命令に関して最初のロードに対して順序付けを強制することになります。もう1つは、FENCE R、Rを使用することですが、これには前のすべてのロードと後続のすべてのロードが含まれるため、このオプションはより高価になります。

lw x1,0(x2)

bne x1,x0,next

sw x3,0(x4)

next:sw x5,0(x6)

図A.11: 構文制御の依存関係

制御依存性はアドレス依存性やデータ依存性とは異なり、制御依存性が常にプログラムの順序で元のターゲットに続くすべての命令に適用されるという意味で動作します。図A.11を考えてみましょう。next命令は常に実行されますが、その最後の命令によって生成されたメモリ操作は、それでも最初の命令によって生成されたメモリ操作から制御依存性を持っています。

lw x1,0(x2)

bne x1,x0,next

next: sw x3,0(x4)

図A.12: 別の構文制御依存性

同様に、図A.12を考えてみましょう。両方の分岐結果が同じターゲットを持っているにもかかわらず、この断片の最初の命令で生成されたメモリ操作から最後の命令で生成されたメモリ操作への制御依存性が残っています。この制御依存性の定義は、他の文脈（例えば C++）で見られるものよりも微妙に強いですが、文献にある制御依存性の標準的な定義に適合しています。

特筆すべきは、PPO規則9-11は、成功したストア条件付き命令の出力に由来する依存関係を尊重するように意図的に設計されていることです。通常、SC命令の後には、結果が成功したかどうかをチェックする条件分岐が続きます。これは、SC命令によって生成されたストア操作から分岐に続くメモリ操作への制御依存性があることを意味します。PPO規則11は、後続のストア操作は、SCによって生成されたストア操作よりもグローバルメモリの順番が遅くなることを意味しています。しかし、制御、アドレス、およびデータの依存性はメモリ操作の上に定義され、失敗したSCはメモリ操作を生成しないので、失敗したSCとその依存命令の間には順序が強制されません。さらに、SCはSCが成功した場合にのみソースレジスタから*rd*に依存性を運ぶように定義されているため、失敗したSCはグローバルメモリオーダーに影響を与えません。

b:Rz\*=0

c:Wz\*=0

d:Wy=0

f:Wx=0

po

data ppo

data ppo

rf

data ppoぽ

rf

a:Rx=0

e:Ry=0

|  |  |
| --- | --- |
| (a) ld a0,0(s0) | (e) ld a3,0(s2) |
| (b) lr a1,0(s1)  (c) sc a2,a0,0(s1)  (d) sd a2,0(s2) | (f) sd a3,0(s0) |

初期値： 0(s0)=1; 0(s1)=1

ハート 0 ハート1

結果：a0=0、a3=0

図A.13: LBリトマス・テストの変型(結果は禁止)

さらに、ストア条件命令からの依存性を尊重するという選択は、ある種の 空気のように薄いような動作を防ぐことを保証します。図A.13を考えてみましょう。仮説的な実装で、保存条件付き命令が成功することを早期に保証することができる場合があるとします。この場合、(c)は(実際に実行する前の)早い段階でa2に0を返し、(d)、(e)、(f)、(a)、(b)のシーケンスを実行できるようにし、(c)はその時点でのみ(成功して)実行するかもしれません。これは、(c)が自分の成功値を0(s1)に書き込むことを意味します!。幸いなことに、RVWMOは成功したSC命令によって生成されたストアに由来する依存関係を尊重するという事実によって、このような状況やその他の状況は防止されています。

また、命令間の構文的な依存関係は、構文的なアドレス、制御、データの依存関係の形をとっている場合にのみ効力を持つことにも注意してください。例えば、第14.3節の“蓄積するCSR”の一つを介した2つの“F”命令間の構文的な依存関係は、2つの“F”命令が順番に実行されなければならないことを意味するものでは*ありません。*このような依存関係は、最終的には、後から問題の CSR フラグにアクセスする CSR 命令への両方の “F"命令からの依存関係を設定するのに役立ちます。

### A.3.9 パイプラインの依存関係 (規則 12-13)

規則12: *bは*ロードであり、プログラムの順序で*a*と*bの*間に*mが存在し、mがa*にアドレスまたはデータ依存性を持ち、*bがm*によって書かれた値を返すようなストア*mが存在します。*

規則 13: *b は*ストアであり、プログラムの順序で *a* と *b の*間には、*m が a* にアドレス依存性を持つような命令 *m が存在します。*

ハート1

ハート 0

li t1, 1

(d) lw a0, 0(s1)

(a) sw t1,0(s0)

(e) sw a0, 0(s2)

(b) fence w, w

(f) lw a1, 0(s2)

(c) sw t1,0(s1)

xor a2,a1,a1

add s0,s0,a2

(g) lw a3,0(s0)

a:Wx=1

c:Wy=1

d:Ry=1

e:Wz=1

f:Rz=1

g:Rx=0

fence

ppo

rf

data

ppo

ppo

rf

addr

ppoい

fr

結果：a0=1, a3=0

図A.14: PPO規則12と(d)から(e)へのデータ依存性のため、(d)はグローバルメモリ順序でも(f)より前になければなりません(結果は禁止されている)

PPO規則12と13は、ほとんどすべての実際のプロセッサパイプライン実装の動作を反映しています。規則12は、そのストアのアドレスとデータが判明するまで、ロードはストアから転送できないと述べています。図A.14を考えてみましょう。(f)は(e)のデータが解決されるまで実行できません。なぜなら、(f)は(e)が書き込んだ値(またはグローバルメモリの順序でさらに後の値)を返さなければならず、(d)が実行する前に(e)のライトバックによって古い値が破壊されてはならないからです。したがって、(d)が実行する前に(f)が実行することはありません。

図A.15のように、(e)と(f)の間に同じアドレスへの別のストアがあれば、(f)はもはや(e)のデータが解決されることに依存しなくなり、したがって(e)のデータを生成する(d)への(f)の依存関係が崩れることになります。

規則13では、前の規則と同様の観察を行います。同じアドレスにアクセスする可能性のある以前のすべてのロードがそれ自体が実行されるまで、メモリ上でストアを実行することはできません。そのようなロードはストアの前に実行されるように見えなければなりませんが、ロードが古い値を読み取る機会を持つ前にストアがメモリ内の値を上書きしてしまうと、そうすることはできません。

ハート 0 ハート 1

a:Wx=1

c:Wy=1

d:Ry=1

e:Wz=1

f:Wz=1

g:Rz=1

h:Rx=0

rf

data

ppo

co

ppo

rf

add

ppo

fr

fence

ppo

|  |  |
| --- | --- |
| li t1, 1 | li t1, 1 |
| (a) sw t1,0(s0) | (d) lw a0, 0(s1) |
| (ｂ） fence ｗ, ｗ | (e) sw a0, 0(s2) |
| (c) sw t1,0(s1) | (f) sw t1, 0(s2)  (g) lw a1, 0(s2) xor a2,a1,a1 add s0,s0,a2  (h) lw a3,0(s0) |

結果：a0=1, a3=0

図A.15: (e)と(g)の間に余分なストアがあるため、(d)は必ずしも(g)に先行しません。

(結果許可)

同様に、先行する命令がアドレス解決の失敗による例外を引き起こさないことが判明するまでは、一般にストアを実行することはできません。この意味で、規則13は規則11のやや特殊なケースであると考えられます。

ハート1

ハート 0

li t1, 1

(a) lw a0,0(s0)

(d) lw a1, 0(s1)

(b) fence rw,rw

(e) lw a2, 0(a1)

(c) sw s2,0(s1)

(f) sw t1, 0(s0)

a:Ry=1

c:Wx=t

d:Rx=t

e:Rt=

*v*

fence

ppo

rf

addr

ppo

ppo

po

rf

結果：a0=1, a1=t f:Wy=1

図A.16: (d)から(e)へのアドレス依存性のため、(d)は(f)にも先行する(結果は禁止されている)。

図A.16を考えてみましょう。(f)は(e)のアドレスが解決されるまで実行できません。したがって、(d)が実行され、アドレスが本当に重なっているかどうかを確認する前に、(f)をメモリに送ることはできません。

## A.4 メインメモリを超えて

RVWMOは現在、FENCE.I、SFENCE.VMA、I/Oフェンス、PMAの動作を正式に記述しようとはしていません。これらの動作はすべて将来の形式化によって記述されます。一方、FENCEIの動作については2.7節、SFENCEVMAの動作についてはRISC-V命令セット特権アーキテクチャマニュアルに記載されており、I/Oフェンスの動作とPMAの効果については以下の通りです。

### A.4.1 コヒーレンスとキャッシュ性

RISC-V 特権 ISA は、アドレス空間の一部がコヒーレントであるかどうかと／やキャッシュ可能であるかどうかなどを指定する物理メモリ属性(PMA)を定義します。完全な詳細については、RISC-V 特権 ISA 仕様を参照してください。ここでは、各PMAの様々な詳細がメモリモデルとどのように関係しているかを簡単に説明します。

* + - メインメモリ対I/O、およびI/Oメモリの順序付け PMA：定義されたメモリモデルはメインメモリ領域に適用されます。I/Oの順序については後述します。
    - サポートされているアクセスタイプとアトミック性PMA: メモリモデルは、各領域がサポートしているプリミティブの上に単純に適用されます。
    - キャッシュ可能性PMA: 一般的にキャッシュ可能性PMAはメモリモデルに影響を与えません。非キャッシュ可能な領域はキャッシュ可能な領域よりも制限的な動作をするかもしれませんが、許可される動作のセットは関係なく変更されません。しかし、プラットフォーム固有のキャッシュ性設定と／やデバイス固有のキャッシュ性設定は異なる場合があります。
    - コヒーレンスPMA：PMA で非コヒーレントとマークされたメモリ領域のメモリ一貫性モデルは、現在のところプラットフォーム固有および/またはデバイス固有のものです: ロード値公理、原子性公理、進行公理はすべて非コヒーレントメモリで違反する可能性があります。しかし、コヒーレントメモリはハードウェアキャッシュコヒーレンスプロトコルを必要としないことに注意してください。RISC-V 特権 ISA 仕様では、メインメモリのハードウェア非コヒーレント領域は推奨されていませんが、メモリモデルは、ハードウェアコヒーレンス、ソフトウェアコヒーレンス、読み取り専用メモリによる暗黙のコヒーレンス、1つのエージェントのみがアクセスすることによる暗黙のコヒーレンス、またはその他の場合に互換性があります。
    - べき等PMAs (べき等PMAs) べき等PMA は、ロードと／やストアが副作用をもたらす可能性のあるメモリ領域を指定するために使用され、これはマイクロアーキテクチャによって、例えばプリフェッチが合法かどうかを判断するために使用されます。この区別はメモリモデルには影響しません。

### A.4.2 I/Oオーダー

I/Oについては、ロード値公理や原子性公理は一般的には適用されません。なぜなら、読み取りと書き込みの両方にデバイス固有の副作用があり、同じアドレスへの最新のストアによって「書き込まれた」値以外の値を返す可能性があるからです。それでも、I/O メモリへのアクセスには、以下の保存されたプログラム順序の規則が一般的に適用されます：プログラム順序で *a が b* よりも優先され、以下のうちの 1 つ以上の条件が満たされている場合、グローバルメモリ順序ではメモリアクセス *a が*メモリアクセス *b よりも優先されます。*

1. *aは*第14章で定義されているように、保存されたプログラム順序で*b*に先行します。ただし、取得と解放の順序の注釈は、1つのメモリ操作から別のメモリ操作へ、1つのI/O操作から別のI/O操作へのみ適用されますが、メモリ操作からI/Oへ、またはその逆は適用されません。
2. *a*と*bは、*I/O領域内の重複するアドレスへのアクセスです。
3. *a*と*bは*同じ強順I/O領域へのアクセスです。
4. *a* と *b は I*/O 領域へのアクセスであり、*a* または *b の*いずれかによってアクセスされた I/O 領域に関連付けられたチャネルをチャネル 1 とします。
5. *a* と *b は*、同じチャネルに関連付けられた I/O 領域へのアクセスです (チャネル 0 を除く)。

FENCE命令は、その前任者と後任者のセットでメインメモリ演算とI/O演算を区別していることに注意してください。I/O操作とメインメモリ操作の間で順序を強制するには、コードはPI、PO、SI、と／やSOに加えてPR、PW、SR、と／やSWを加えたFENCEを使用しなければなりません。例えば、メインメモリへの書き込みとデバイスレジスタへのI/O書き込みの間で順序を強制するには、FENCE W,O以上が必要です。

sd t0, 0(a0)

fence w, o

sd a0, 0(a1)

図A.17: メモリとI/Oアクセスの順番

実際にフェンスが使用されている場合、実装ではMMIO信号を受信した直後にデバイスがメモリにアクセスしようとする可能性があり、そのデバイスからメモリへの後続のメモリ・アクセスは、そのMMIO操作の前に順序付けられたすべてのアクセスの影響を受けなければならないと仮定しなければならない。言い換えれば、図A.17では、0(a0)がメイン・メモリ内にあり、0(a1)がI/Oメモリ内のデバイス・レジスタのアドレスであるとします。デバイスがMMIO書き込みを受けて0(a0)にアクセスする場合、そのロードはRVWMOメモリ・モデルの規則に従って0(a0)への最初のストアの後に概念的に現れなければなりません。一部の実装では、これを確実にする唯一の方法は、MMIO書き込みが発行される前に最初のストアが実際に完了していることを要求することです。他の実装では、より積極的な方法を見つけることができるかもしれませんが、他の実装では、I/Oとメイン・メモリ・アクセスについては全く異なることをする必要がないかもしれません。それにもかかわらず、RVWMOメモリモデルはこれらのオプションを区別するものではありません。

多くのアーキテクチャでは、特にI/O(通常のメインメモリとは対照的に)に関連して、 ”順序付け"と“完了"のフェンスを別々に定義しています。順番フェンスはメモリ操作が順番通りに行われることを保証しますが、補完フェンスは前任者のアクセスがすべて完了してから後任者が見えるようにすることを保証します。RISC-Vでは、順序フェンスと補完フェンスを明示的に区別していません。代わりに、この区別は単に FENCE ビットの使用法の違いから推測されています。

RISC-V Unixプラットフォーム仕様に準拠した実装では、I/OデバイスとDMA操作は、コヒーレントに、かつ強順のI/Oチャネルを介してメモリにアクセスすることが要求されます。したがって、外部デバイスによって同時にアクセスされる通常のメインメモリ領域へのアクセスも、標準の同期メカニズムを使用することができます。Unixプラットフォーム仕様に準拠していない実装、および/またはデバイスがメモリに首尾一貫してアクセスしない実装では、首尾一貫性を強制するためのメカニズム（現在のところプラットフォーム固有またはデバイス固有）を使用する必要があります。

アドレス空間内のI/O領域は、その領域のPMAではキャッシュされない領域と考えるべきです。このような領域は、どのエージェントによってもキャッシュされていない場合、PMAによってコヒーレントとみなすことができます。

本節の順序保証は、RISC-Vコアとデバイス間のプラットフォーム固有の境界を超えて適用されない場合があります。特に、外部バス（PCIeなど）を介して送信されたI/Oアクセスは、最終目的地に到達する前に順序が変更される可能性があります。このような状況では、それらの外部デバイスやバスのプラットフォーム固有の規則に従って、順序付けを行う必要があります。

## A.5 コードの移植とマッピングのガイドライン

|  |  |
| --- | --- |
| x86/TSO操作 | RVWMOマッピング |
| ロード | l{b|h|w|d}; fence r,rw |
| ストア | fence rw,w; s{b|h|w|d} |
| アトミックRMW | amo<op>.{w|d}.aqrl OR  loop: lr.{w|d}.aq; <op>; sc.{w|d}.aqrl; bnez loop |
| フェンス | fence rw,rw |

表A.2: TSO演算からRISC-V演算へのマッピング

表A.2にTSOメモリ操作からRISC-Vメモリ命令へのマッピングを示します。通常の x86 のロードとストアは、すべて本質的に 獲得-RCpc と 解放-RCpc の操作です。TSOは、デフォルトですべてのロード・ロード、ロード・ストア、ストア・ストアの順序を強制します。したがって、RVWMOの下では、TSOのすべてのロードはロードにマップされ、FENCE R,RWの後にロードが続き、TSOのすべてのストアはストアにマップされ、FENCE RW,Wの後にストアが続く必要があります。TSOアトミックリード・モディファイ・ライトとLOCK接頭辞を使用したx86命令は完全に順序付けされており、*aq*と*rl*の両方がセットされたAMOを介して実装するか、または*aqが*セットされたLR、問題の演算処理、*aq*と*rlの*両方がセットされたSC、成功条件をチェックする条件分岐を介して実装することができます。後者の場合LR上の*rl*注釈は（明らかでない理由で）冗長であることが判明し、省略できます。

表A.2の代替案も可能である。TSOストアは、*rl*を設定してAMOSWAPにマッピングすることができます。しかし、RVWMO PPO規則3では、AMOから後続の負荷への値の転送が禁止されているため、ストアにAMOSWAPを使用すると性能に悪影響を及ぼす可能性があります。TSOロードは、*aqが*設定されたLRを使用してマッピングすることができます。このようなLR命令はすべてペアリングされないが、それ自体はロードにLRを使用することを排除するものではありません。しかし、このマッピングが本来意図した以上に予約機構に圧力をかけると、性能に悪影響を及ぼす可能性があります。

|  |  |
| --- | --- |
| 電源操作 | RVWMOマッピング |
| ロード | l{b|h|w|d} |
| ロード-リザーブ | lr.{w|d} |
| ストア | s{b|h|w|d} |
| ストア条件付き | sc.{w|d} |
| lwsync | fence.tso |
| sync | fence rw,rw |
| isync | fence.i; fence r,r |

表A.3: パワー動作からRISC-V動作へのマッピング

表A.3は、Powerのメモリ操作とRISC-Vのメモリ命令のマッピングを示しています。

Power ISYNCはRISC-VではFENCE.IとFENCE R,Rに対応します。RVWMOでは存在しない「制御＋制御のフェンス」の依存関係を定義するためにISYNCが使用されるため、後者のフェンスが必要になります。

|  |  |
| --- | --- |
| ARM動作 | RVWMOマッピング |
| ロード | l{b|h|w|d} |
| ロード-獲得 | fence rw, rw; l{b|h|w|d}; fence r,rw |
| ロード-排他 | lr.{w|d} |
| ロード-獲得-排他 | lr.{w|d}.aqrl |
| ストア | s{b|h|w|d} |
| ストア-解放 | fence rw,w; s{b|h|w|d} |
| ストア-排他 | sc.{w|d} |
| ストア-解放-排他- | sc.{w|d}.rl |
| dmb | fence rw,rw |
| ｄｍｂ.ｌｄ | fence r,rw |
| ｄｍｂ．st | fence w,w |
| isb | fence.i; fence r,r |

表A.4: ARM動作からRISC-V動作へのマッピング

表A.4にARMのメモリ操作からRISC-Vのメモリ命令へのマッピングを示します。RISC-Vには現在のところ*aq*や*rlのアノテーションを*持つプレーンなロード/ストアオペコードがないため、ARMのロード-獲得とストア-解放操作は代わりにフェンスを使用してマッピングする必要があります。さらに、ストア-解放-ロード-獲得の順序を強制するためには、ストア-解放とロード-獲得の間にFENCE RW,RWがなければなりません；表A.4では、各獲得操作の前に常にFENCEを配置することでこれを強制しています。ARMのロード-排他命令とストア-排他命令も同様にRISC-VのLRとSCにマップすることができるが、*aq*をセットしたLRの前にFENCE RW,RWを配置する代わりに、単に*rl*をセットするだけです。ARMのISBはRISC-V上でFENCE.Iの後にFENCE R,Rが続き、ISYNCがPowerにマッピングするのと同じようにRISC-V上でFENCE.Iがマッピングされます。

表A.5にLinuxメモリ順序付けマクロのRISC-Vメモリ命令へのマッピングを示します。Linux のフェンス dma rmb() と dma wmb() はそれぞれ FENCE R,R と FENCE W,W にマッピングしていますが、RISC-V Unix プラットフォームはコヒーレント DMA を必要とするので、非コヒーレント DMA のプラットフォームでは、それぞれ FENCE RI,RI と FENCE WO,WO にマッピングされます。非コヒーレントＤＭＡを有するプラットフォームでは、キャッシュラインをフラッシュおよび／または無効にすることができるメカニズムも必要になるかもしれません。そのようなメカニズムは、デバイス固有のものであり、将来のISAの拡張で標準化されることになる。

リリース操作のための Linux のマッピングは必要以上に強力に見えるかもしれませんが、これらのマッピングは、Linux がより直感的なマッピングよりも強力な順序付けを必要とするいくつかのケースをカバーするために必要とされています。特に、このテキストが書かれている時点では、Linux は、ある重要なセクションのアクセスと、同じハートで、同じ同期オブジェクトで保護された後続の重要なセクションのアクセスの間で、ロード-ロード、ロード-ストア、ストア-ストアの順序付けを要求するかどうかを活発に議論しています。FENCE RW,W/FENCE R,RW マッピングと *aq*/*rl マッピングの*すべての組み合わせがそのような順序付けを提供するとは限りません。この問題を回避する方法はいくつかあります。

1. 常にFENCE RW,W/FENCE R,RWを使用し、決して*aq*/*rlを*使用しない。これで十分ですが、*aq*/*rl*修飾子の目的を達成できないので望ましくありません。

|  |  |
| --- | --- |
| Linux操作 | RVWMOマッピング |
| smp\_mb() | fence rw,rw |
| smp\_rmb() | fence r,r |
| smp\_wmb() | fence w,w |
| dma\_rmb() | fence r,r |
| dma\_wmb() | fence w,w |
| mb() | fence iorw,iorw |
| rmb() | fence ri,ri |
| wmb() | fence wo,wo |
| smp\_load\_acquire() | l{b|h|w|d}; fence r,rw |
| smp\_store\_release() | fence.tso; s{b|h|w|d} |
| Linux 校正 | RVWMO AMOマッピング |
| atomic <op> relaxed | amo<op>.{w|d} |
| atomic <op> acquire | amo<op>.{w|d}.aq |
| atomic <op> release | mo<op>.{w|d}.rl |
| atomic <op> | amo<op>.{w|d}.aqrl |
| Linux 校正 | RVWMO LR/SCマッピング |
| atomic <op> relaxed | loop: lr.{w|d}; <op>; sc.{w|d}; bnez loop |
| atomic <op> acquire | loop: lr.{w|d}.aq; <op>; sc.{w|d}; bnez loop |
| atomic <op> release | loop: lr.{w|d}; <op>; sc.{w|d}.aqrl∗; bnez loop OR  fence.tso; loop: lr.{w|d}; <op>; sc.{w|d}∗; bnez loop |
| atomic <op> | loop: lr.{w|d}.aq; <op>; sc.{w|d}.aqrl; bnez loop |

表A.5: LinuxメモリプリミティブからRISC-Vプリミティブへのマッピング。

他のコンストラクト(スピンロックなど)もそれに従うべきです。非コヒーレント DMA を持つプラットフォームやデバイスは、追加の同期化(キャッシュフラッシュや無効化メカニズムなど)を必要とするかもしれません。

1. 常に *aq*/*rl を*使用し、FENCE RW,W/FENCE R,RW は絶対に使用しないでください。これは、*aq*と*rlの*修飾子を持つロードとストアのオペコードがないため、現在は動作しません。
2. どちらかのタイプの獲得マッピングの存在下で十分な順序付けを強制するように、リリース操作のマッピングを強化します。これは現在推奨されている解決策であり、表A.5に示されているものです。

|  |  |
| --- | --- |
| Linuxのコード:  (a) int r0 = \*x;  (bc) spin\_unlock(y, 0);  ...  ...  (d) spin\_lock(y);  (e) int r1 = \*z; | RVWMOマッピング。  (a) lw a0, 0(s0)  (b) fence.tso // vs. fence rw,w  (c) sd x0,0(s1)  ...  ループ:  (d) amoswap.d.aq a1,t1,0(s1) bnez a1,loop  (e) lw a2,0(s2) |

図A.18: Linuxにおけるクリティカルセクション間の順序付け

例えば、現在 Linux コミュニティで議論されているクリティカルセクションの順序付け規則では、図 A.18 の(a)は(e)の前に順序付けられなければなりません。もしそれが本当に要求されるのであれば、(b) は FENCE RW,W としてマッピングするには不十分です。とはいえ、これらのマッピングは Linux カーネル・メモリ・モデルの進化に合わせて変更される可能性があります。

|  |  |
| --- | --- |
| C/C++ コンストラクト | RVWMOマッピング |
| Non-atomic load | l{b|h|w|d} |
| atomic load(memory order relaxed) | l{b|h|w|d} |
| atomic load(memory order acquire) | l{b|h|w|d}; fence r,rw |
| atomic load(memory order seq cst) | fence rw,rw; l{b|h|w|d}; fence r,rw |
| Non-atomic store | s{b|h|w|d} |
| atomic store(memory order relaxed) | s{b|h|w|d} |
| atomic store(memory order release) | fence rw,w; s{b|h|w|d} |
| atomic store(memory order seq cst) | fence rw,w; s{b|h|w|d} |
| atomic thread fence(memory order acquire) | fence r,rw |
| atomic thread fence(memory order release) | fence rw,w |
| atomic thread fence(memory order acq rel) | fence.tso |
| atomic thread fence(memory order seq cst) | fence rw,rw |
| C/C++ コンストラクト | RVWMO AMO マッピング |
| atomic <op>(memory order relaxed) | amo<op>.{w|d} |
| atomic <op>(memory order acquire) | amo<op>.{w|d}.aq |
| atomic <op>(memory order release) | amo<op>.{w|d}.rl |
| atomic <op>(memory order acq rel) | amo<op>.{w|d}.aqrl |
| atomic <op>(memory order seq cst) | amo<op>.{w|d}.aqrl |
| C/C++ コンストラクト | RVWMO LR/SC マッピング |
| atomic <op>(memory order relaxed) | loop: lr.{w|d}; <op>; sc.{w|d};  bnez loop |
| atomic <op>(memory order acquire) | loop: lr.{w|d}.aq; <op>; sc.{w|d};  bnez loop |
| atomic <op>(memory order release) | loop: lr.{w|d}; <op>; sc.{w|d}.rl;  bnez loop |
| atomic <op>(memory order acq rel) | loop: lr.{w|d}.aq; <op>; sc.{w|d}.rl;  bnez loop |
| atomic <op>(memory order seq cst) | loop: lr.{w|d}.aqrl; <op>;  sc.{w|d}.rl; bnez loop |

表A.6: C/C++プリミティブからRISC-Vプリミティブへのマッピング。

表A.6にC11/C++11のアトミック演算のRISC-Vメモリ命令へのマッピングを示します。*aq*および*rl*修飾子を持つロードおよびストアオペコードが導入された場合、表A.7のマッピングで十分です。ただし、この2つのマッピングは、atmic\_<op>(memory order seq cst) が *aq*と*rlの*両方が設定されたLRを用いてマッピングされた場合にのみ正しく相互運用されることに注意します。

AMOはLRとSCのペアでエミュレートできますが、LRを起点とするPPO命令はSCを起点とし、SCで終了するPPO命令はLRで終了するように注意しなければなりません。例えば、ロード・オペレーションにはデータ依存性の概念がないため、LRはAMOが持つデータ依存性を尊重するようにしなければなりません。

|  |  |
| --- | --- |
| C/C++ コンストラクト | RVWMOマッピング |
| Non-atomic load | l{b|h|w|d} |
| atomic load(memory order relaxed) | l{b|h|w|d} |
| atomic load(memory order acquire) | l{b|h|w|d}.aq |
| atomic load(memory order seq cst) | l{b|h|w|d}.aq |
| Non-atomic store | s{b|h|w|d} |
| atomic store(memory order relaxed) | s{b|h|w|d} |
| atomic store(memory order release) | s{b|h|w|d}.rl |
| atomic store(memory order seq cst) | s{b|h|w|d}.rl |
| atomic thread fence(memory order acquire) | fence r,rw |
| atomic thread fence(memory order release) | fence rw,w |
| atomic thread fence(memory order acq rel) | fence.tso |
| atomic thread fence(memory order seq cst) | fence rw,rw |
| C/C++ コンストラクト | RVWMO AMO マッピング |
| atomic <op>(memory order relaxed) | amo<op>.{w|d} |
| atomic <op>(memory order acquire) | amo<op>.{w|d}.aq |
| atomic <op>(memory order release) | amo<op>.{w|d}.rl |
| atomic <op>(memory order acq rel) | amo<op>.{w|d}.aqrl |
| atomic <op>(memory order seq cst) | amo<op>.{w|d}.aqrl |
| C/C++ コンストラクト | RVWMO LR/SC Mapping |
| atomic <op>(memory order relaxed) | lr.{w|d}; <op>; sc.{w|d} |
| atomic <op>(memory order acquire) | lr.{w|d}.aq; <op>; sc.{w|d} |
| atomic <op>(memory order release) | lr.{w|d}; <op>; sc.{w|d}.rl |
| atomic <op>(memory order acq rel) | lr.{w|d}.aq; <op>; sc.{w|d}.rl |
| atomic <op>(memory order seq cst) | lr.{w|d}.aq∗; <op>; sc.{w|d}.rl |

\*表 A.6 に基づいてマッピングされたコードと相互運用するために、lr.{w|d}.aqrl でなければなりません。

表A.7: ネイティブのロード-獲得およびストア-解放 オペコードが導入された場合の、C/C++プリミティブからRISC-Vプリミティブへの仮想的なマッピング。

同様に、同じハート内の他の場所にある FENCE R,Rの効果を、そのフェンスを尊重しないSCにも適用する必要がある。エミュレータは、AMOをlr.aq; <op>; sc.aqrlに単純にマッピングすることで、この効果を得ることができます。これは、完全に順序付けられたアトミックに使用されているマッピングと同じです。

## A.6 実施要領

RVWMO および RVTSO メモリモデルは、マイクロアーキテクトがより高い性能を実現するために高度な投機技術や他の形態の最適化を採用することを妨げるものではありません。また、これらのモデルは、特定のキャッシュ階層を使用する必要はなく、キャッシュコヒーレンスプロトコルを使用する必要もありません。その代わりに、これらのモデルはソフトウェアに公開できる動作を指定するだけです。マイクロアーキテクチャは、メモリモデルの規則を満たす実行のみを認める設計であれば、どのようなパイプライン設計、コヒーレントまたは非コヒーレントなキャッシュ階層、オンチップ相互接続などを使用することも自由です。

とはいえ、メモリモデルの実際の実装を理解してもらうために、このセクションでは、アーキテクトやプログラマがどのようにメモリモデルの規則を解釈すべきかについて、いくつかのガイドラインを示しています。

RVWMOとRVTSOはどちらもマルチ-コピー アトミック（または“アザー-マルチ-コピー-アトミック''）です。であり、最初に発行したハート以外のハートから見えるストア値は、システム内の他のすべてのハートからも概念的に見える必要があります。言い換えれば、ハーツは、すべてのハーツからグローバルに見えるようになる前に、自分の前のストアから転送することができますが、ハーツ間の早期転送は許可されていません。マルチコピーのアトミック性は、さまざまな方法で強制されます。キャッシュとストアバッファの物理的な設計によって本質的に保持される場合もあれば、シングルライター/マルチリーダーのキャッシュコヒーレンスプロトコルによって強制される場合もありますし、その他のメカニズムによって保持される場合もあります。

マルチコピーアトミック性はマイクロアーキテクチャにいくつかの制限を課しますが、メモリモデルが極端に複雑にならないようにするための重要な特性の一つです。例えば、あるハートは隣接するハートのプライベートストアバッファからの値を合法的に転送することはできません(もちろん、新しい違法な動作がアーキテクチャ的に見えなくなるような方法で行われない限り)。また、キャッシュコヒーレンスプロトコルは、コヒーレンスプロトコルが他のキャッシュからの古いコピーをすべて無効にするまで、あるハートから別のハートに値を転送してはいけません。もちろん、マイクロアーキテクチャは、準拠していない動作がプログラマに公開されない限り、投機や他の最適化によって隠れてこれらの規則に違反する可能性があります（そして、高パフォーマンスの実装は、おそらく違反するでしょう）。

RVWMOにおけるPPO規則の解釈の大まかな指針として、ソフトウェアの観点から以下のようなことを期待しています。

* プログラマーは、PPO規則1と4-8を定期的に積極的に使用します。
* エキスパートプログラマはPPO規則9-11を使用して、重要なデータ構造の重要なパスを高速化します。
* 専門のプログラマーであっても、PPO規則2-3と12-13を直接使用することはほとんどありません。これらは、セクションB.3で説明した共通のマイクロアーキテクチャ最適化(規則2)と運用上のフォーマルモデリングアプローチ(規則3と12-13)を容易にするために含まれています。これらはまた、同様の規則を持つ他のアーキテクチャからコードを移植するプロセスを容易にします。

また、ハードウェア的には以下のようなことを期待しています。

* PPOの規則1と3-6は、アーキテクトにはほとんど驚きを与えないように、よく理解された規則を反映しています。
* PPO規則2は、自然で一般的なハードウェアの最適化を反映していますが、非常に微妙なため、慎重にダブルチェックする価値があります。
* PPO規則7はアーキテクトにはすぐにはわからないかもしれませんが、標準的なメモリモデルの要件です。
* ロード値公理、原子性公理、PPO規則8-13は、極端な最適化が含まれていない限り、ほとんどのハードウェア実装が自然に実施するであろう規則を反映しています。もちろん、実装者はこれらの規則をダブルチェックするようにしなければなりません。ハードウェアはまた、構文依存性が「最適化されていない」ことを保証しなければなりません。

アーキテクチャは、メモリモデルの規則を自由に選択して保守的に実装することができます。例えば、ハードウェアの実装では、以下のいずれかまたはすべてを選択することができます。：

* 実際に設定されているビットに関係なく、すべてのフェンスをFENCE RW,RW（I/Oが関与している場合はFENCE IORW,IORW）であるかのように解釈します。
* PWとSRを使ったすべてのフェンスを、あたかもFENCE RW,RW（またはI/Oが関係する場合はFENCE IORW,IORW）のように実装します。なぜなら、PWとSRは4つの可能なメインメモリ順序付けコンポーネントの中で最も高価だからです。
* セクションA.5で説明したように、aqとrlをエミュレートします
* “fri-rfi" や ”RSW" のようなパターンがあっても、すべての同じアドレスのロードロード順を強制します。
* ストアバッファ内のストアから後続のAMOやLRに同じアドレスの値を転送することを禁止します。
* ストアバッファ内のAMOまたはSCから、同じアドレスへの後続のロードへの値の転送を禁止します。
* すべてのメモリアクセスにTSOを実装し、PWとSRの順序付けを含まないメインメモリフェンスを無視します（例えば、Ztsoの実装がそうであるように）。
* アノテーションに関係なく、すべてのアトミックをRCscまたは完全に順序付けして実装します。

RVTSOを実装したアーキテクチャは、次のようなことを安全に行うことができます。：

* PWとSRの両方を持たない全てのフェンスは無視します(フェンスがI/Oも命令しない限り)
* RVTSOを想定した場合、他のPPO規則と重複するため、規則4～7を除くすべてのPPO規則を無視します。

その他一般的な注意事項。

* サイレント・ストア（すなわち、メモリ・ロケーションに既に存在する同じ値を書き込むストア）は、メモリ・モデルの観点からは、他のストアと同様に動作します。同様に、実際にはメモリ内の値を変更しない AMO (例えば、*rs2* の値が現在のメモリ内の値よりも小さい AMOMAX) は、意味的にはストア操作と見なされます。サイレントストアを実装しようとするマイクロアーキテクチャは、特にサイレントストアと互換性がない傾向にある RSW (セクション A.3.5) のようなケースでは、メモリモデルが依然として従うことを確実にするために注意を払わなければなりません。
* 書き込みは、結果として生じる動作がメモリモデルのセマンティクスに違反しない限り、結合（すなわち、同じアドレスへの2つの連続した書き込みを結合することができます）または包含（すなわち、同じアドレスへの2つのバックツーバックの書き込みのうち、より早い方の書き込みを排除することができます）することができます。

書き捨ての問題は、以下の例から理解できます。

書かれているように、ロード(d)が値1を読み込んだ場合、グローバルメモリの順序では(a)が(f)の前になければなりません。：

* 規則2により、グローバルメモリの順序では(a)が(c)に先行します。

ハート 0

ハート1

li t1, 3

li t3, 2

li t2, 1

(a) sw t1,0(s0)

(d) lw a0,0(s1)

(b) fence w, w

(e) sw a0,0(s0)

(c) sw t2,0(s1)

(f) sw t3,0(s0)

a:Wx=3

c:Wy=1

e:Wx=1

d:Ry=1

f:Wx=2

fence

ppo

co

rfＦ

data ppo

co

ppoい

図A.19: 包摂リトマス試験を書く、許可された実行。

* (c)が(d)に先行するのは、ロードバリューの公理のためである。
* 規則7により、グローバルメモリの順序では(d)が(e)よりも前になります。
* f規則1のため、グローバルメモリの順序では(e)が(f)に先行する。

つまり、アドレスがs0にあるメモリ位置の最終値は2（ストア(f)が書き込んだ値）でなければならず、3（ストア(a)が書き込んだ値）にすることはできません。

非常に攻撃的なマイクロアーキテクチャでは、(f)が優先されるため、誤って(e)を破棄することになるかもしれません。これにより、マイクロアーキテクチャは、(d)と(f)の間の依存関係を解消してしまうかもしれません（したがって、(a)と(f)の間の依存関係も解消してしまいます）。これは、メモリモデルの規則に違反するため、禁止されています。例えば、(d)と(e)の間にデータの依存関係がない場合は、書き込みの包含は別のケースでは合法となります。

### A.6.1 将来の拡張可能性

以下の将来の拡張機能のいずれか、またはすべてがRVWMOメモリモデルと互換性があることを期待しています。

* ‘V' ベクトルISA拡張
* “T” ISA拡張のトランザクションメモリのサブセット
* ‘J' JIT拡張
* *aq* と *rl が*設定されたロードおよびストアオペコード用のネイティブエンコーディング
* 特定のaddressに限定されたフェンス
* キャッシュ・ライトバック/フラッシュ/無効/その他の命令

ハート 0 ハート 1

|  |  |
| --- | --- |
| li t1, 1 | li t1, 1 |
| (a) lw a0,0(s0) | (d) lw a1,0(s1) |
| (b) fence rw,rw | (e) amoswap.w.rl a2,t1,0(s2) |
| (c) sw t1,0(s1) | (f) ld a3,0(s2)  (g) lw a4,4(s2)  xor a5,a4,a4  add s0,s0,a5  (h) sw a2,0(s0) |

結果：a0=1、a1=1、a2=0、a3=1、a4=0

図A.20: 混合サイズの不一致（公理モデルでは許容され，運用モデルでは禁止される）

ハート 0 ハート1

|  |  |
| --- | --- |
| li t1, 1 | li t1, 1 |
| (a) lw a0,0(s0) | (d) ld a1,0(s1) |
| (b) fence rw,rw | (e) lw a2,4(S1) |
| (c) sw t1,0(s1) | xor a3,a2,a2  add s0,s0,a3  (f) sw a2,0(s0) |

結果：a0=0, a1=1, a2=0

図A.21: 混合サイズの不一致（公理モデルでは許容され，運用モデルでは禁止される）

ハート 0 ハート 1

|  |  |
| --- | --- |
| li t1, 1 | li t1, 1 |
| (a) lw a0,0(s0) | (d) sw t1,4(s1) |
| (b) フェンスrw,rw | (e) ld a1,0(s1) |
| (c) sw t1,0(s1) | (f) lw a2,4(s1)  xor a3,a2,a2  add s0,s0,a3  (g) sw a2,0(s0) |

結果：a0=1、a1=0x100000001、a1=1

図A.22: 混合サイズの不一致（公理モデルでは許容され，運用モデルでは禁止される

## A.7 既知の問題

### A.7.1 混合サイズRSW

図A.20-A.22に示されている混合サイズのRSWバリアントのファミリ内では、運用上の仕様と公理上の仕様の間には既知の不一致があります。これに対処するために、以下の新しいPPO規則のようなものを追加することができます。*プログラムの順序でaがbに先行し、aとbの両方が（I/O領域ではなく）通常のメインメモリにアクセスし、aがロードで、bがストアで、aとbの間にロードmがあり、aとmの両方が読み込むバイトxがあり、aとmの間にxに書き込むストアがなく、PPOでmがbに先行する場合、保存されたプログラムの順序（したがって、グローバルメモリの順序）でメモリ操作aがメモリ操作bに先行します*。言い換えれば、群れの構文では、PPOに “(po-loc & rsw);ppo;[W]" を追加することを選択することができます。多くの実装では、すでに自然にこの順序が適用されています。

そのため、この規則は公式なものではありませんが、将来的にRVWMOにこの規則が追加される可能性があるため、互換性を確保するために、実装者はこの規則を実施することを推奨します。

**付録B**

# 形式記憶モデル仕様書、バージョン0.1

この章では、RVWMOの形式的な分析を容易にするために、さまざまなツールとモデル化アプローチを用いた一連の形式化を紹介します。矛盾は意図的なものではありません。モデルは全く同じ法的行動のセットを記述していることが期待されています。

この付録は、解説として扱われるべきです。すべての規範的な資料は、第14章とISA仕様の本文の残りの部分で提供されている。現在知られているすべての不一致は、セクションA.7にリストアップされています。その他の不一致は意図的なものではありません。

## B.1 F合金の標準的な公理的な指定

合金におけるRVWMOメモリモデルの正式な仕様を紹介します([http://alloy.mit.edu)](http://alloy.mit.edu/)。このモデルはオンラインで <https://github.com/daniellustig/riscv-memory-model>から入手可能です[。](https://github.com/daniellustig/riscv-memory-model)

オンライン資料には、いくつかのリトマステストと、セクションA.5のマッピングのいくつかをモデル化してチェックするために合金をどのように使用することができるかのいくつかの例も含まれています。

////////////////////////////////////////////////////////////////////////////////   
// =RVWMO PPO=

// 保存されたプログラム・オーダー fun ppo : Event->Event -  
 // 同一アドレスのオーダー   
 po.loc :> Store

+ ｒｄｗ

+ (AMO + StoreConditional) <： rfi

// 明示的な同期化

+ ppo\_fence

+ Acquire <:. ^po :> MemoryEvent   
+ MemoryEvent <: ^po :> Release

+ RCsc <: ^po :> RCsc

+ pair

// 構文的な依存関係

+ addrdep

+ datadep

+ ctrldep :> Store

// パイプラインの依存関係  
+ (addrdep+datadep).rfi  
+ addrdep .^po :> Store

}

// グローバルメモリオーダーは、保存されたプログラムオーダーを尊重しています。   
fact { ppo in ^gmo }

図B.1: 合金(1/5:PPO)で形式化されたRVWMOメモリモデル

////////////////////////////////////////////////////////////////////////////////   
// =RVWMO公理=

// ロード値公理

fun candidates [r: MemoryEvent ] : set MemoryEvent {

(r.~^ gmo & Store & same\_addr [r]) // 先行するrをgmoに書き込む

+ (r.^~ po & Store & same\_addr [r]) // 先行するrをpoに書き込む

}

fun latest\_among [s: set Event ] : Event { s - s.~^ gmo }

pred LoadValue {

all w: Store | all r: Load |

w->r in rf <=> w = latest\_among [ candidates [r]]

}

// Atomicity Axiom

pred Atomicity {

all r: Store .~ pair | // lrから始まります ,

no x: Store & same\_addr [r] | // 同じアドレスへのストアxは存在しません

x not in same\_hart [r] // x が異なる hart のものであるような,

and x in r.~ rf .^ gmo // xはgmoで（rが読み込んだストア）の後に続きます ,

and r. pair in x.^ gmo //そしてrはgmoでxに続きます

}

// 暗黙の進行公理： 合金は有限実行のみを考慮します

pred RISCV\_mm { LoadValue and Atomicity /\* と進歩 \*/ }

図B.2: 合金で形式化されたRVWMOメモリモデル (2/5: 公理)

////////////////////////////////////////////////////////////////////////////////

// メモリの基本モデル

sig Hart { // ハードウェアスレッド

start : one Event

}

sig Address {}

abstract sig Event {

po: lone Event // プログラムの順序

}

abstract sig MemoryEvent extends Event {

address : one Address ,

acquireRCpc : lone MemoryEvent ,

acquireRCsc : lone MemoryEvent ,

releaseRCpc : lone MemoryEvent ,

releaseRCsc : lone MemoryEvent ,

addrdep : set MemoryEvent ,

ctrldep : set Event ,

datadep : set MemoryEvent ,

gmo : set MemoryEvent , // グローバル・メモリー・オーダー

rf: set MemoryEvent

}

sig LoadNormal extends MemoryEvent {} // l{b|h|w|d}

sig LoadReserve extends MemoryEvent { // lr

pair : lone StoreConditional

}

sig StoreNormal extends MemoryEvent {} // s{b|h|w|d}

// モデル内のすべてのストア・コンディショナルが成功していると仮定した場合

sig StoreConditional extends MemoryEvent {} // sc

sig AMO extends MemoryEvent {} // amo

sig NOP extends Event {}

fun Load : Event { LoadNormal + LoadReserve + AMO }

fun Store : Event { StoreNormal + StoreConditional + AMO }

sig Fence extends Event {

pr: lone Fence , // オペコードビット

pw: lone Fence , // オペコードビット

sr: lone Fence , // オペコードビット

sw: lone Fence // オペコードビット

}

sig FenceTSO extends Fence {}

/\* 合金エンコーディングの詳細： オペコードビットは設定される（例：f.pr in idenとしてエンコードされる）か、

\* 設定されていない（f.pr が idenにない）かのどちらかです。

\* これらのビットは他の用途には使用できません。 \*/|

fact { pr + pw + sr + sw in iden }

// 同様に、アノテーションの順序付け

fact { acquireRCpc + acquireRCsc + releaseRCpc + releaseRCsc in iden }

// FenceTSOをpr/pw/sr/swでエンコードしようとしないで、そのまま使用してください。

fact { no FenceTSO .( pr + pw + sr + sw) }

図B.3: Alloyで形式化されたRVWMOメモリモデル(3/5:メモリのモデル)

////////////////////////////////////////////////////////////////////////////////

// 基本的なモデルの規則

// 順序付け注釈グループ  
fun Acquire : MemoryEvent { MemoryEvent . acquireRCpc + MemoryEvent . acquireRCsc }

fun Release : MemoryEvent { MemoryEvent . releaseRCpc + MemoryEvent . releaseRCsc }

fun RCpc : MemoryEvent { MemoryEvent . acquireRCpc + MemoryEvent . releaseRCpc }

fun RCsc : MemoryEvent { MemoryEvent . acquireRCsc + MemoryEvent . releaseRCsc }

// ストア-取得、ロード-解放というのは、両方でない限りありません。

fact { Load & Release in Acquire }

fact { Store & Acquire in Release }

// FENCE PPO

fun FencePRSR : Fence { Fence .( pr & sr) }

fun FencePRSW : Fence { Fence .( pr & sw) }

fun FencePWSR : Fence { Fence .( pw & sr) }

fun FencePWSW : Fence { Fence .( pw & sw) }

fun ppo\_fence : MemoryEvent -> MemoryEvent {

( Load <: ^po :> FencePRSR ).(^ po :> Load )

+ ( Load <: ^po :> FencePRSW ).(^ po :> Store )

+ ( Store <: ^po :> FencePWSR ).(^ po :> Load )

+ ( Store <: ^po :> FencePWSW ).(^ po :> Store )

+ ( Load <: ^po :> FenceTSO ) .(^ po :> MemoryEvent )

+ ( Store <: ^po :> FenceTSO ) .(^ po :> Store )

}

// 補助定義

fun po\_loc : Event -> Event { ^po & address .~ address }

fun same\_hart [e: Event ] : set Event { e + e.^~ po + e.^ po }

fun same\_addr [e: Event ] : set Event { e. address .~ address }

// 初期ストア

fun NonInit : set Event { Hart . start .\* po }

fun Init : set Event { Event - NonInit }

fact { Init in StoreNormal }

fact { Init ->( MemoryEvent & NonInit ) in ^ gmo }

fact { all e: NonInit | one e .\*~ po .~ start } //各イベントは正確に1つのハートの中にあります。

fact { all a: Address | one Init & a.~ address } // アドレスごとに1つのinitストア

fact { no Init <: po and no po :> Init }

図B.4: 合金で形式化されたRVWMOメモリモデル(4/5:基本モデル規則)

// po

fact { acyclic [po] }

// gmo

fact { total [^ gmo , MemoryEvent ] } // gmoは、すべてのMemoryEventsに対する全順序です。

// rf

fact { rf .~ rf in iden } // 読み込み時には、書き込み時の値が返されます。

fact { rf in Store <: address .~ address :> Load }

fun rfi : MemoryEvent -> MemoryEvent { rf & (\* po + \*~ po) }

// dep

fact { no StoreNormal <: ( addrdep + ctrldep + datadep ) }

fact { addrdep + ctrldep + datadep + pair in ^po }

fact { datadep in datadep :> Store }

fact { ctrldep .\* po in ctrldep }

fact { no pair & (^ po :> ( LoadReserve + StoreConditional )).^ po }

fact { StoreConditional in LoadReserve . pair } //すべてのSCが成功したと仮定

// rdw

fun rdw : Event -> Event {

( Load <: po\_loc :> Load ) // すべての同じアドレスのロード - ロードのペアで開始します。

- (~ rf.rf) // 同じストアから読み込まれたペアを減算します。

- ( po\_loc . rfi) // そして、「fri - rfi」のパターンを引き算します。

}

// 冗長なインスタンスやビジュアライゼーションを排除

fact { no gmo & gmo. gmo } // 視覚的にスッキリした状態を維持します

fact { all a: Address | some a.~ address }

// //////////////////////////////////////////////////////////////////////////////

// = オプション：オペコードのエンコーディング制限 =

// 祝福されたフェンスのリスト

fact { Fence in

Fence .pr.sr

+ Fence .pw.sw

+ Fence .pr.pw.sw

+ Fence .pr.sr.sw

+ FenceTSO

+ Fence .pr.pw.sr.sw

}

pred restrict\_to\_current\_encodings {

no ( LoadNormal + StoreNormal ) & ( Acquire + Release )

}

// //////////////////////////////////////////////////////////////////////////////

// = 合金のショートカット =

pred acyclic [ rel: Event -> Event ] { no iden & ^ rel }

pred total [ rel: Event ->Event , bag: Event ] {

all disj e, e ’: bag | e->e’ in rel + ~rel

acyclic [rel ]

}

図B.5: 合金で形式化されたRVWMOメモリモデル (5/5: Auxiliaries)

## B.2 群れの正規公理仕様

このツール群は、メモリモデルとリトマステストを入力として受け取り、メモリモデルの上でテストの実行をシミュレートします。メモリモデルはドメイン固有の言語であるCATで記述されます。このセクションでは、RVWMOの2つのCATメモリモデルを提供します。最初のモデルである図B.7は、CATモデルとしては可能な限り、RVWMOの定義である第14章の*グローバルメモリ順序*に従っています。2番目のモデルである図B.8は、等価で、より効率的な、部分順序に基づくRVWMOモデルです。

シミュレータの群れは、DIY ツール・スイートの一部です --- ソフトウェアとドキュメントについては [http://diy.inria.fr を](http://diy.inria.fr/)参照してください。モデルなどは <http://diy.inria.fr/cats7/riscv/>からオンラインで入手できます[。](http://diy.inria.fr/cats7/riscv/)

(\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\* ユーティリティー\*)

(\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\* すべてのフェンスの関係 \*)

let fence .r.r = [R]; fencerel ( Fence .r.r);[ R]

let fence .r.w = [R]; fencerel ( Fence .r.w);[ W]

let fence .r.rw = [R]; fencerel ( Fence .r.rw );[ M]

let fence .w.r = [W]; fencerel ( Fence .w.r);[ R]

let fence .w.w = [W]; fencerel ( Fence .w.w);[ W]

let fence .w.rw = [W]; fencerel ( Fence .w.rw );[ M]

let fence .rw.r = [M]; fencerel ( Fence .rw.r);[ R]

let fence .rw.w = [M]; fencerel ( Fence .rw.w);[ W]

let fence .rw.rw = [M]; fencerel ( Fence .rw.rw );[ M]

let fence .tso =

let f = fencerel ( Fence .tso ) in

([W];f;[W]) | ([R];f;[M])

let fence =

fence .r.r | fence .r.w | fence .r.rw |

fence .w.r | fence .w.w | fence .w.rw |

fence .rw.r | fence .rw.w | fence .rw.rw |

fence .tso

(\* 同一のアドレスで、同じアドレスへのWアクセスはできません。 \*)

let po -loc -no -w = po -loc ／ (po -loc ?;[ W];po -loc )

(\* 読んで同じ書く \*)

let rsw = rf ^ -1; rf

(\* 取得、またはより強力 \*)

let AQ = Acq | AcqRel

(\* 解放、またはより強力 \*)

and RL = RelAcqRel

(\* 全 RCsc \*)

let RCsc = Acq |Rel | AcqRel

(\* AmoのイベントはRとWの両方があり、関係rmwはペアのｌｒ/ｓｃに関係します。\*)

let AMO = R & W

let StCond = range (rmw )

(\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\* ppo 規則\*)

(\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\*重複するアドレスの順序\*)

let r1 = [M];po -loc ;[W]

and r2 = ([R];po -loc -no -w;[R]) ／ rsw

and r3 = [AMO | StCond ]; rfi ;[R]

(\* 明示的な同期化 \*)

and r4 = fence

and r5 = [AQ ]; po ;[M]

and r6 = [M]; po ;[ RL]

and r7 = [ RCsc ]; po ;[ RCsc ]

and r8 = rmw

(\* 構文依存性 \*)

and r9 = [M]; addr ;[M]

and r10 = [M]; data ;[W]

and r11 = [M]; ctrl ;[W]

(\*パイプラインの依存関係 \*)

and r12 = [R];( addr | data );[ W]; rfi ;[R]

and r13 = [R]; addr ;[M]; po ;[W]

let ppo = r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9 | r10 | r11 | r12 | r13

図 B.6: 保存されたプログラム順序の群れ定義である riscv-defs.cat (1/3)

合計

(\* 群れが独自のrf関係を定義していることに注目してください。 \*)

(\* 定義 ppo \*)

include "riscv - defs .cat "

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\* グローバルなメモリーオーダーの生成 \*)

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

let gmo0 = (\* 前駆体：すなわち、gmo0を含む総てのオーダーとしてgmoを構築する。 \*)

loc & (W／FW) \* FW | # 同じ場所への書き込み後の最終書き込み

ppo | # ppo 互換性

rfe # 群れの外部RF（最適化）を含む

(\* gmo0のすべての線形拡張の上を歩く \*)

with gmo from linearizations (M／IW , gmo0 )

(\* rfGMOの計算に便利な初期書き込みを前もって追加する。 \*)

let gmo = gmo | loc & IW \* (M／IW)

(\*\*\*\*\*\*\*\*\*\*)

(\* 公理\*)

(\*\*\*\*\*\*\*\*\*\*)

(\* ロード値公理（別名：rfGMO）に従ってrfを計算する \*)

let WR = loc & ([W];( gmo |po );[ R])

let rfGMO = WR ／ (loc &([ W]; gmo ); WR)

(\* 群れのrfとrfGMOの同等性の確認 \*)

empty (rf／ rfGMO )|( rfGMO ／rf) as RfCons

(\* 原子性の公理 \*)

let infloc = (gmo & loc )^ -1

let inflocext = infloc & ext

let winside = ( infloc ;rmw ; inflocext ) & ( infloc ;rf;rmw ; inflocext ) & [W]

empty winside as Atomic

図B.7: RVWMOメモリモデルの群れバージョンであるriscv.cat (2/3)

部分的

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\* 定義\*)

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\* 定義 ppo \*)

include "riscv - defs .cat "

(\* コヒーレンス関係の計算 \*)

include "cos -opt .cat "

(\*\*\*\*\*\*\*\*\*\*)

(\* 公理\*)

(\*\*\*\*\*\*\*\*\*\*)

(\* ロケーションごとのSc \*)

acyclic co|rf|fr|po -loc as Coherence

(\* 主要モデルの公理 \*)

acyclic co|rfe |fr|ppo as Model

(\* 原子性の公理 \*)

empty rmw & (fre;coe) as Atomic

図B.8：RVWMOメモリモデルの代替群の提示であるriscv.cat (3/3)

## B.3 オペレーショナルメモリモデル

これは、RVWMOメモリモデルを運用スタイルで表現したものです。これは、公理表現と全く同じ拡張動作を認めることを目的としています。任意のプログラムに対して、公理表現がそれを許す場合にのみ実行を認めます。

公理的表現は、完全な候補実行に関する述語として定義されます。対照的に、この演算表現は抽象的なマイクロアーキテクチャー的な特徴を持っています：ステートマシンとして表現され、ハードウェアマシンの状態を抽象的に表現した状態を持ち、明示的な順序外実行と投機的実行を持ちます（ただし、レジスタのリネーム、ストアバッファ、キャッシュ階層、キャッシュプロトコルなど、より実装固有のマイクロアーキテクチャー的な詳細は抽象化しています）。そのため、有用な直感を提供することができます。また、インクリメンタルに実行を構築することができ、より大きな例の動作をインタラクティブかつランダムに探索することが可能になりますが、公理モデルは公理をチェックできる完全な候補実行を必要とします。

動作説明では、異なるパワーオブツーバイトサイズのメモリアクセスが重なり合う可能性のあるミックスサイズ実行をカバーしています。不整合アクセスは、シングルバイトアクセスに分割されます。

運用モデルは、RISC-V ISAセマンティクスの断片(RV64IとA)とともに、 rmem探索ツール[(https://github.com/rems-project/rmem)](https://github.com/rems-project/rmem)に統合されています。 rmemは、リトマステスト(A.2参照)と小さなELFバイナリを網羅的に、疑似ランダムに、対話的に探索することができます。rmemでは、ISAセマンティクスはSailで明示的に表現されており(Sail言語については [https://github.com/ rems-project/sail、](https://github.com/rems-project/sail)RISC-V ISAモデルについては [https://github.com/rems-project/sail-riscv を参照](https://github.com/rems-project/sail-riscv))、同時実行セマンティクスはLemで表現されています(Lem言語については [https: //github.com/rems-project/lemを](https://github.com/rems-project/lem)参照)。

rmemには、コマンドラインインターフェースとウェブインターフェースがあります。ウェブインタフェースはクライアント側で完全に動作し、リトマステストのライブラリと共にオンラインで提供されています: [http://www.cl.cam.ac.uk/ \~pes20/rmem.](http://www.cl.cam.ac.uk/~pes20/rmem)コマンドラインインタフェースは、特に網羅的なモードでは、ウェブインタフェースよりも高速です。

以下に、モデルの状態と遷移を非公式に紹介します。形式的なモデルの説明は次のサブセクションから始まります。

用語解説：公理的な表現とは対照的に、ここではすべてのメモリ操作はロードまたはストアのいずれかです。したがって、AMOはロードとストアという2つの異なるメモリ操作を生み出します。命令と一緒に使う場合、“ロード''と”ストア''という用語はそのようなメモリ操作を引き起こす命令を指します。そのため、どちらもAMO命令を含んでいます。“取得”とは、acquire-RCpcまたはacquire-RCscアノテーションを持つ命令（またはそのメモリ操作）のことです。“解放"とは、release-RCpcまたはrelease-RCscアノテーションを持つ命令（またはそのメモリ操作）のことです。

**モデルの状態** モデル状態は、共有メモリとハート状態のタプルで構成されています。

|  |  |  |
| --- | --- | --- |
| ハート 0 | **...** | ハート *n* |

↑ ↓ ↑ ↓

共有メモリ

共有メモリの状態は、これまでに伝搬したすべてのメモリストア操作を伝搬した順に記録します(これはより効率的にすることができますが、プレゼンテーションを簡単にするためにこの方法にしておきます)。

各ハート状態は、主に命令インスタンスのツリーで構成されており、いくつかの命令インスタンスは*完了したものと完了してい*ないものがあります。終了していない命令インスタンスは*再起動の*対象となることがあります。例えば、順番外の負荷や投機的な負荷に依存していて、それが不健全であることが判明した場合などです。

条件分岐命令や間接ジャンプ命令は、命令ツリー内に複数の後継者を持つことがあります。そのような命令が終了した場合、取られていない代替パスはすべて破棄されます。

命令ツリー内の各命令インスタンスは、命令内セマンティクス（この命令のISA擬似コード）の実行状態を含む状態を持っています。このモデルでは、Sail の命令内セマンティクスを形式化したものを使用しています。命令の実行状態は、擬似コード制御状態、擬似コード呼び出しスタック、ローカル変数の値を表現したものと考えることができます。命令インスタンスの状態には、インスタンスのメモリとレジスタのフットプリント、レジスタの読み書き、メモリ操作、終了の有無などの情報も含まれます。

**モデル遷移** モデルは任意のモデル状態に対して、許容される遷移のセットを定義し、それぞれが新しい抽象マシン状態への1つの原子的なステップとなります。1 つの命令の実行には通常多くの遷移が含まれますが、オペレーショナルモデルの実行では、他の命令から発生する遷移と混在している場合があります。各遷移は単一の命令インスタンスから発生し、そのインスタンスの状態を変更し、そのインスタンスの残りのハート状態と共有メモリ状態に依存したり変更したりしますが、他のハート状態には依存せず、それらを変更することはありません。以下に遷移を紹介し、それぞれについて前提条件と遷移後のモデル状態の構築を伴ってB.3.5節で定義します。

すべての命令のためのトランジション：

* フェッチ命令：この遷移は、以前にフェッチされた命令インスタンス（または最初のフェッチアドレス）のプログラム順序の後継として、新しい命令インスタンスのフェッチとデコードを表します。

このモデルは命令メモリが固定されていることを前提としており、自己修正するコードの動作は記述していません。特に、Fetch命令の遷移はメモリロード操作を生成せず、共有メモリは遷移に関与しません。その代わりに、モデルは、メモリ位置が与えられたときにオペコードを提供する外部神託に依存します。

* レジスタ ライト：レジスタ値の書き込みです。
* レジスタ リード：これは、そのレジスタに書き込む最新のプログラム命令前置命令インスタンスからのレジスタ値の読み出しです。
* 擬似コード内部ステップ：これは、疑似コードの内部計算をカバーします: 算術、関数呼び出しなど。
* 命令終了：この時点で命令の疑似コードは完了し、命令は再起動できず、メモリアクセスは破棄できず、すべてのメモリ効果が発生しています。

条件分岐命令や間接ジャンプ命令では、*pc* レジスタに書き込まれたアドレス以外のアドレスから取得されたプログラム順序の後継命令は、その下にある命令インスタンスのサブツリーと共に破棄されます。

ロード命令に固有のトランジション：

* メモリロード操作を開始：この時点でロード命令のメモリフットプリントが暫定的に把握され(以前の命令が再起動された場合は変更される可能性があります)、個々のメモリロード操作が開始されます。
* 伝搬されていないメモリストアからの転送でメモリロード動作を満足させる：プログラムオーダ前のメモリストア動作からの転送によるメモリロード動作の一部または全部を満たします。
* メモリからのメモリロード動作を満足させる：これは、メモリからの単一のメモリロード動作の顕著なスライスを完全に満たします。
* 完全なロード操作：この時点で、命令のすべてのメモリロード操作が完全に満たされ、命令の擬似コードは実行を続けることができます。ロード命令は、Finish命令遷移まで再起動される可能性があります。しかし、ある条件の下では、モデルはロード命令が終了する前であっても再起動できないものとして扱うかもしれません（例：ストア操作の伝搬を参照）。

ストア命令に固有のトランジション：

* メモリ・ストアのオペレーション・フットプリントを開始する：この時点で、ストアのメモリフットプリントは暫定的に知られています。
* メモリストア操作の値をインスタンス化する：この時点でメモリストア操作の値が設定されており、そこから転送することでプログラム順後続のメモリロード操作を満足させることができます。
* 完遂ストア命令：この時点で、ストア処理は保証され（命令はもう再起動も破棄もできない）、メモリへの伝搬を開始することができます。
* ストア操作を伝搬：1つのメモリストア操作をメモリに伝搬します。
* 完遂ストア操作：この時点で、命令のすべてのメモリストア操作がメモリに伝搬され、命令の疑似コードは実行を続けることができます。

sc命令に特有の遷移。

* 初期の sc の失敗：これは、自然発生的に失敗したか、プログラムオーダー前のlrとペアになっていないために、scが失敗する原因となります。
* ペアの sc：この伝搬は、scがlrとペアになっていることを示し、成功する可能性があることを示しています。
* scのストア操作を完遂して伝搬します：これは遷移のアトミックな実行です。

完遂ストア命令と伝搬ストア動作は、lrが読み込んだストアが上書きされていない場合にのみ有効になります。

* 遅い sc 失敗：これは sc が失敗する原因となります。自然発生的に失敗したか、または lr が読み込んだストアが上書きされたためです。

AMO命令に特有の遷移：

* AMO の操作を満足させ、コミットし、伝搬させる：これは、ロード操作を満足させ、必要な演算を行い、ストア操作を伝搬させるために必要なすべての遷移をアトミックに実行します。

フェンスの命令に特有の遷移：

* 完遂フェンス

〇というラベルの付いた遷移は、他の動作を除外することなく、前提条件が満たされるとすぐに、常に熱心に実行できます。 ●はできません。フェッチ命令は●でマークされていますが、何度も何度も実行されない限り、熱心に実行できます。

非AMOロード命令のインスタンスは、フェッチされた後、通常、この順序で以下の遷移を経験します：

1. レジスタ読み取り
2. メモリロード操作の開始
3. 非伝搬ストアからの転送でメモリロード動作を満足させる、またはメモリからのメモリロード動作を満足させる（インスタンスの全てのロード動作を満足させるために必要な数だけ）。
4. 完全なロード操作
5. レジスタ書き込み
6. 終了命令

上記のトランジションの前、間、および後に、任意の数の疑似コード内部ステップ遷移が出現する可能性があります。さらに、次のプログラム位置の命令をフェッチするためのフェッチ命令遷移は、それが取られるまで利用可能です。

以上で、運用モデルの非公式な説明を終了します。以下のセクションでは、正式な運用モデルについて説明します。

### B.3.1 命令内擬似コード実行

各命令インスタンスの命令内意味規則は、状態遷移として表現され、本質的に命令の疑似コードを実行します。擬似コードの実行状態が与えられると、それは次の状態を計算します。ほとんどの状態は、疑似コードによって要求され、メモリモデルが実行しなければならない保留中のメモリまたはレジスタ操作を識別します。状態は以下の通りです（これはタグ付きの結合で、タグは小文字で表示されています）。

|  |  |  |
| --- | --- | --- |
| LOAD\_MEM(*種類、アドレス、サイズ、ロード継続*) | - | メモリロード操作 |
| EARLY\_SC\_FAIL(再*継続*) | - | scが早期に失敗することを許可 |
| STORE\_EA(*種類、アドレス、サイズ、次の状態*) | - | メモリストアの実効アドレス |
| STORE\_MEMV(*メモリ値、ストア継続*) | - | メモリストア値 |
| FENCE(*種類、次の状態*) | - | フェンス |
| READ\_REG(*登録名、読み取り継続*) | - | レジスタの読み出し |
| WRITE\_REG(*登録名、登録値、次の状態*) | - | レジスタの書き込み |
| INTERNAL(*次の状態*) | - | 疑似コード内部ステップ |
| DONE | - | 疑似コードの終了 |

ここで：

* *mem値*と*reg値*はバイトのリストです。
* *アドレス*はXLENビットの整数です。
* ロード/ストアの場合、*種類*はlr/sc、取得-RCpc/解放-RCpc、取得-RCsc/解放-RCsc、取得-解放-RCscであるかを識別します。
* フェンスの場合は、*種類*によって通常のフェンスかTSOかを識別し、（通常のフェンスの場合）前任者と後任者の順序ビットを識別します。
* *reg name*は，レジスタとそのスライス（開始ビットと終了ビットのインデックス）を識別します。
* 継続は、周辺のメモリモデルで提供される可能性のある各値に対して、命令インスタンスがどのように継続するかを記述するものです。(*ロード継続*と*リード継続*は、メモリからロードされた値と前回のレジスタライトから読み出された値を取り、*ストア継続*は、scが失敗した場合は*false*を取り、それ以外の場合は*true*を取り、*res継続*は、scが失敗した場合は*false*を取り、それ以外の場合は*true*を取ります）。

*例えば、ロード命令* lw x1,0(x2) *が与えられた場合、実行は通常以下のようになります。最初の実行状態は、与えられたオペコードの疑似コードから計算されます。これは、*READ\_REG*(*x2*, リード継続)と予想できます。最近書き込まれたレジスタ* x2 の*値 (レジスタ値が利用可能になるまで命令のセマンティクスは必要に応じてブロックされます)、例えば* 0x4000 *をリード継続に供給すると、*LOAD\_MEM*(*plain load*,* 0x4000*,* 4*, ロード継続) が返されます。メモリロケーション*0x4000*からロードされた4バイトの値を、0x*42*などのメモリロケーションからロード継続に供給すると、*WRITE\_REG*(*x1*, 0*x42*,* Done*)が返ってきます。上記の状態の前後には、多くの*内部*(次の状態)が出現する可能性があります。*

メモリへの書き込みは、STORE\_EAとSTORE\_MEMVの2つのステップに分割されていることに注意してください: 最初のステップでは、ストアのメモリフットプリントが暫定的に知らされ、2番目のステップでは、格納される値が追加されます。擬似コードではこれらがペアになっていることを確認していますが(STORE\_EAの後にSTORE\_MEMVが続く)、これらの間には他のステップがあるかもしれません。

STORE\_EAは、*格納されるべき値が決定される前に発生し得ることが観察可能です。例えば、リトマステストLB+fence.r.r.rw+data-poが動作モデルで許可されるためには（RVWMOで許可されているように）、ハート1の最初のストアは、その値が決定される前に*STORE\_EAの*ステップを踏まなければならず、それによって2番目のストアは、それが重複しないメモリフットプリントにあることを知ることができ、コヒーレンスに違反することなく2番目のストアを順番外でコミットすることができます。*

各命令の擬似コードは、ロードとストアを正確に1つずつ実行するAMOを除いて、最大で1つのストアまたはロードを実行します。これらのメモリアクセスは、その後、ハートセマンティクスによってアーキテクチャ的にアトミックな単位に分割されます（後述のメモリロード操作の開始とメモリストア操作のフットプリントの開始を参照してください）。

非公式には、レジスタリードの各ビットは、そのビットを書き込むことができる最新の（プログラム順の）命令インスタンスによるレジスタライトから（または、そのようなライトがない場合はhartの初期レジスタ状態から）満たされる必要があります。したがって、各命令インスタンスのレジスタ書き込みフットプリントを知ることは不可欠であり、これは命令インスタンスが作成されたときに計算します（後述のFetch命令の動作を参照してください）。擬似コードでは、各命令が各レジスタビットに対して最大1回のレジスタ書き込みを行い、また、書き込んだばかりのレジスタ値を読み出そうとしないようにしています。

モデル内のデータフロー依存性（アドレスとデータ）は、各レジスタ読み出しが適切なレジスタ書き込みが実行されるのを待たなければならないという事実から生じます（上述のように）。

### B.3.2 命令インスタンス状態

各命令インスタンス*iは、以下*からなる状態を持っています。

* *プログラム・ロック：*命令がフェッチされたメモリ・アドレス。
* *命令の種類：*ロード、ストア、AMO、フェンス、ブランチ/ジャンプ、または“単純な”命令であるかどうかを識別します（これには、擬似コードの実行状態について説明したものと同様の種類も含まれます）。
* *src regs：* 命令の疑似コードから静的に決定されたソースレジスタ名（システムレジスタを含みます）のセット。
* *dst regs：* 命令の疑似コードから統計的に決定されたデスティネーションのREG名（システムレジスターを含みます）。
* 疑*似コード状態：* (あるいは略して “状態” と呼ばれることもあります) の一つです (これはタグ付きの組合です。スモールキャップのタグ)。

Plain(isa 状態) - 擬似コードに移行する準備ができています。

Pending mem loads(ロード継続) - メモリのロード操作を要求しています。

Pending mem stores(ストア継続) - メモリのストア操作を要求しています。

* *reg reads,：*インスタンスが実行したレジスタ・リードで、それぞれにリード元のレジスタ・ライト・スライスが含まれます。
* *reg writes：*インスタンスが実行したレジスタライト。
* mem\_loads：メモリ・ロード・オペレーションのセットで、それぞれについて、まだ満足していないスライス（まだ満足していないバイト・インデックス）と、満足したスライスについて、それを満足させたストア・スライス（それぞれがメモリ・ストア・オペレーションとそのバイト・インデックスのサブセットから構成される）が含まれます。
* mem stores: メモリストア操作の集合体であり，それぞれの操作に対して，伝播されたかどうか（共有メモリに渡されたかどうか）を示すフラグが設定されています。
* インスタンスがコミットされたか、終了したかなどを記録する情報。

*memストア*、メモリストア操作のセット、各メモリストアに対して伝播（共有メモリに渡される）されたかどうかを示すフラグ、インスタンスがコミットされたかどうか、終了したかどうかなどを記録する情報

各メモリロード操作は、メモリフットプリント（アドレスとサイズ）を含みます。各メモリストア操作は、メモリフットプリントを含み、利用可能な場合には、値を含みます。

空ではない*メモリロードを*持つロード命令インスタンスで、すべてのロード操作が満たされている（つまり、満たされていないロードスライスがない）場合は、*完全に満たされ*ていると言われます。

非公式には、命令インスタンスは、そのソースレジスタに供給されるロード（およびsc）命令が終了した場合、*完全に決定されたデータ*を持つと言われます。同様に、命令インスタンスは、そのメモリ操作アドレスレジスタに供給するロード（およびsc）命令が終了した場合、*完全に決定されたメモリフットプリント*を有すると言われます。形式的には、まず、*完全に決定されたレジスタライト*の概念を定義します。命令インスタンス*i*の*レジスタライト*からのレジスタライト*w*は、以下の条件のいずれかが成立する場合、*完全に決定されている*といいます。

1. *i は*終了しました。 または
2. *w*によって書き込まれた値は、*iが行った*メモリ操作（すなわち、メモリからロードされた値やscの結果）の影響を受けず、*wに*影響を与える*iが*行ったすべてのレジスタ読み出しについて、*iが*読み出したレジスタの書き込みが完全に決定されます（または、*i*は初期レジスタ状態から読み出します）。

ここで、命令インスタンス*i*は、*reg\_reads*から*r*を読み出すごとに、*r*が読み出すレジスタライトが完全に決定されている場合、*完全に決定されたデータ*を持っていると言われます。命令インスタンス*i*は、*i*のメモリ操作アドレスにフィードするreg\_readsからのレジスタリード*r*ごとに、*r*がリードするレジスタライトが完全に決定されている場合、*完全に決定されたメモリフットプリント*を持っていると言われています。

rmem*ツールは、レジスタ書き込みごとに、書き込みを実行した時点でこの命令が読み込んだ他の命令からのレジスタ書き込みのセットを記録します。このツールが対象とする命令の擬似コードを慎重に配置することで、書き込みが依存するレジスタ書き込みのセットを正確に記録することができました。*

### B.3.3 ハート状態

シングルハートのモデル状態は、次のように構成されています。

* hart id：hartのユニークな識別子。.
* initial register state; 各レジスタの初期レジスタ値。
* initial fetch address：初期の命令フェッチアドレス。
* instruction tree: プログラム順にフェッチされた(廃棄されなかった)命令インスタンスのツリー。

### B.3.4 共有メモリの状態

共有メモリのモデル状態は、共有メモリに伝搬した順にメモリストア操作のリストで構成されています。

ストア操作が共有メモリに伝搬されると、単純にリストの最後に追加されます。メモリからロード操作が満たされると、ロード操作の各バイトに対して、最新の対応するストアスライスが返されます。

*ほとんどの目的のためには、共有メモリを配列、すなわちメモリロケーションからメモリストア操作のスライスへのマップとして考える方が簡単です。各メモリロケーションは、そのロケーションに対する最新のメモリストアオペレーションの1バイトスライスにマッピングされます。しかし、この抽象化は、*sc*命令を適切に扱うのに十分に詳細ではありません。ＲＶＷＭＯ原子性公理は、*ｓｃ*と同じハートからのストア操作が、*ｓｃ*のストア操作と、ペアになった*ｌｒが*読み込んだストア操作との間に介在することを可能にする。このようなストア操作の介入を許可し、他の操作を禁止するためには、より多くの情報を記録するために配列の抽象化を拡張しなければなりません。ここでは、非常に単純なのでリストを使用しますが、より効率的でスケーラブルな実装では、おそらくより良いものを使用する必要があります。*

### B.3.5 遷移

以下の各段落では、1 つの種類のシステム遷移について説明します。説明は、現在のシステムの状態に対する条件から始まります。条件が満たされた場合にのみ、現在の状態で遷移を行うことができます。この条件の後には、新しいシステム状態を生成するために、遷移が実行されたときにその状態に適用されるアクションが続きます。

**フェッチ命令** 命令インスタンス*i*のプログラム順序の後続の可能性があるものは、もしあればアドレス*loc*から取得することができます。

1. つまり、ハートの命令ツリーでiの直接の後継者がlocから来ていないということです。 そして
2. *i の擬似*コードが既に *pc* にアドレスを書き込んでいる場合、*loc は*そのアドレスでなければなりません、そうでなければlocは：

* 条件付き分岐の場合は、分岐先アドレスまたは分岐先アドレス。
* 直接ジャンプ・リンク命令(jal)の場合は，ターゲットアドレス。
* 間接ジャンプ命令（jalr）の場合は，任意のアドレス。
* その他の命令は、i.program loc + 4。

動作: プログラム・メモリの*loc*にある命令に対して、新たに初期化された命令インスタンスi'を構築し、*命令の種類*、*src regs、dst regs*などの疑似コードから得られる静的情報を含む、命令の疑似コードから計算された状態PLAIN（*isa 状態*）を持ち、i'をiの後継としてhartの*命令ツリー*に追加します。

*iをフェッチした直後に、次のフェッチが可能なアドレス（loc）が得られ、モデルは擬似コードがpcに書き込まれるのを待つ必要がありません。これにより、アウトオブオーダー実行や、条件分岐やジャンプを越えた推測が可能になります。ほとんどの命令では、これらのアドレスは命令の疑似コードから簡単に得ることができます。唯一の例外は間接ジャンプ命令(jalr)で、アドレスはレジスタに保持されている値に依存します。原理的には、数学モデルはここでの任意のアドレスへの推測を可能にするはずです。*

rmem *ツールの網羅的検索は、間接ジャンプごとに次のフェッチアドレスの可能性のあるセットを増やしながら、網羅的検索を複数回実行することでこれを処理します。最初の検索では空のセットを使用しますので、間接ジャンプ命令の後には、命令の擬似コードが*pc*に書き込まれるまでは、次の命令のフェッチにその値を使用します。網羅的検索の次の反復を開始する前に、各間接ジャンプ（コード位置でグループ化されている）について、前の検索反復におけるすべての実行でpcに書き込まれた値のセットを収集し、それを命令の可能な次のフェッチ・アドレスとして使用します。この処理は、新しいフェッチアドレスが検出されなかった場合に終了します。*

**メモリロード操作の開始** PLAIN(Load mem(種類, アドレス, サイズ, *load 継続*))の状態の命令インスタンス*iは*、常に対応するメモリロード操作を開始することができます。

動作：

1. 適切なメモリロード操作 *mlos を*構築します。

* *アドレスがサイズ*に合わせられている場合、*mlosはアドレス*からの*サイズ*バイトのシングルメモリロード操作です。
* そうでなければ、*mlosは、アドレス...address* + *size* - 1から、それぞれ1バイトずつのサイズメモリロード操作のセットです。

1. *i* の *mem ロードを mlos* に設定します。 そして
2. *iの*状態をPENDING\_MEM\_LOADS(*load 継続*)に更新します。

*第14.1節では、不整合メモリアクセスは任意の粒度で分解できると言われています。ここでは、この粒度が他のすべてを包含するので、それらを１バイトアクセスに分解します。*

**未伝搬のストアからの転送でメモリロード操作を満たす。**

PENDING\_MEM\_LOADS(*load 継続*)の状態にある非AMOロード命令インスタンス*i*と、*i.mem loads*の中にある未充足のスライスを持つメモリロードオペレーション*mlo*の場合、*i*よりもプログラムオーダーが前のストア命令インスタンスによる未伝搬のメモリストアオペレーションからのフォワードによって、メモリロードオペレーションの一部または全部を満たすことができます。

1. .srと.pwのセットを使ったプログラムオーダー前のフェンスの命令がすべて終了しました。
2. .srと.prが設定され、.pwが設定されていない、プログラムオーダー以前のフェンス命令*f*に対して、*fが*終了していない場合、*f*以前のプログラムオーダーであるすべてのロード命令が完全に満たされます。
3. プログラム順序以前の*fence*.tso命令*fが*終了していない場合、*f*以前のプログラム順序であるすべてのロード命令が完全に満たされます。
4. *iが*ロード-取得-RCscの場合は、すべてのプログラムオーダー-過去のストア-解放-RCscを終了します。
5. *iが*ロード-取得-解放の場合は、すべてのプログラム-オーダ-前の命令が終了します。
6. すべての未完成のプログラム-オーダー-前のロード-取得の命令は完全に満たされています。
7. すべてのプログラム-オーダー-前のストア-取得-解放の命令は終了しました。

*msossは、*プログラム順序*i*以前の非SCストア命令インスタンスからのすべての伝搬されていないメモリストアオペレーションスライスの集合であり、すでに格納される値が計算されており、*mlo*の不満足なスライスと重複しており、介入するストアオペレーションまたは介入する負荷によって読み出されるストアオペレーションによって優先されないものであるとします。最後の条件は、命令*i‘* からの各メモリ・ストア・オペレーション・スライス*msos* in *msossを*要求します。

* iとi'の間に、メモリ・ストア・オペレーションがオーバーラップしているストア命令のプログラム・オーダーがないこと。
* iとi'の間に、異なるハートからのメモリ・ストア・オペレーション・スライスが重なって満たされたロード命令のプログラム・オーダーがないこと。

動作：

1. *mlo が msoss* によって満たされたことを示すために *mem をロードします*。
2. この結果、コヒーレンスを侵害した投機的な命令を再起動する、つまり *i*のプログラムオーダー・サクセサーであるすべての終了していない命令*i'*と、*msoss'*から満たされた*i'*のすべてのメモリロードオペレーション*mlo'*について、*msoss'*にメモリストアオペレーションスライス*msos'*が存在し、*msos*sにある別のメモリストアオペレーションからのオーバーラップするメモリストアオペレーションスライスが存在し、*msos'*が*i*のプログラム-オーダー-サクセサーである命令からのものではない場合、*i'*とその再起動-依存物を再起動します。

ここで、命令*jの再起動依存関係は：*

* jのレジスタライトにデータ-フローが依存しているjのプログラム-オーダー-サクセサー。
* jのメモリ・ストア・操作から読み出すメモリ・ロード・操作を持つjのプログラム-オーダー-サクセサー（フォワードにより）。
* jがロード-取得の場合、jのすべてのプログラム-オーダー-サクセサー。
* jがロードである場合、jのプログラム-オーダー-サクセサーである、.srと.prが設定され、.pwが設定されていないすべてのフェンスfについて、fのプログラム-オーダー-サクセサーであるすべてのロード命令。
* jがロードである場合、jのプログラム-オーダー-サクセサーであるすべてのfence.tso, fについて、fのプログラム-オーダー-サクセサーであるすべてのロード命令。
* (再帰的に)上記のすべての命令インスタンスのすべての再起動-依存物。

*メモリストア操作をメモリロードに転送すると、ロードの一部のスライスだけが満たされ、他のスライスは満たされない場合があります。*

*上記の移行を行ったときには利用できなかったプログラム-オーダ-前のストア操作が利用可能になったときに、msossが暫定的に不健全になる（コヒーレンスに違反する）可能性があります。そのストアはロードが終了するのを防ぎ（終了命令を参照）、そのストア操作が伝搬されたときにロードが再起動されるようになります（ストア操作の伝搬を参照）。*

*上記の移行条件の結果として、ストア-解放-RCscのメモリストア操作は、ロード-取得-RCsc命令に転送することができません。msossには、終了したストアからのメモリストア操作は含まれておらず（これらは伝播されたメモリストア操作でなければならないため）、上記の条件では、ロードが取得-RCscであるときに、すべてのプログラム-オーダー-前 ストア-解放-RCscが終了している必要があります。*

**メモリからのメモリロード操作を満足させる。** 非AMOロード命令または“AMOの操作を満たし、コミットし、伝播”遷移の文脈におけるAMO命令の命令インスタンス*i*について、*i.memロード*の中で未充足のスライスがあるメモリロード操作*mlo*は、未伝播のストアからの転送によるメモリロード操作の充足の条件がすべて満たされていれば、メモリから充足することができます。

動作：*msoss*を*mlo*の不満足なスライスをカバーするメモリからのメモリストア操作のスライスとし、未伝搬のストアからの転送によるメモリロード操作の満足のアクションを適用します。

*ただし，プロパゲーションされていないストアからの転送によるメモリロード動作をサティスフィで行うと，メモリロード動作の一部のスライスが満たされないままになってしまう場合があるので，その場合は，再度トランジションを行うか，メモリからサティスフィを行うことで満たさなければならなりません．一方、メモリからのメモリロード操作を実行すると、メモリロード操作のすべての満たされていないスライスが常に満たされます。*

**ロード操作の完了**状態 PENDING\_MEM\_LOADS(ロード 継続)のロード命令インスタンス*iは、*すべてのメモリロード操作*i.mem\_loadsが*完全に満たされている(すなわち、不満足なスライスがない)場合に完了することができます(finishedと混同されないように)。アクション：*i*の状態をPLAIN(ロード *継続(mem 値*))に更新します。ここでは、*mem 値はi.memロードを*満たした全てのメモリストア操作のスライスから組み立てられます。

**早期SC失敗** PLAIN(EARLY\_SC\_FAIL(*res 継続*))の状態のsc命令インスタンス*i*は、常に失敗させることができます。アクション: *i* の状態を PLAIN(*res 継続(失敗))* に更新します。

**ペアリングされた** sc 命令インスタンス *i の*状態が PLAIN(早期 sc 失敗(*res 継続*)) の場合、*i が* lr とペアリングされていれば、(成功する可能性のある) 実行を継続することができます。アクション: *i* の状態を PLAIN(*res 継続(成功))* に更新します。

**メモリストア操作のフットプリントを開始** PLAIN(ストア EA(種類, アドレス, サイズ, 次状態))の状態にある命令インスタンス*iは*、保留中のメモリストア操作のフットプリントを常にアナウンスすることができます。  
 アクション：

1. 適切なメモリストア操作 *msos* (ストア値なし) を構築します。

* *アドレスがサイズにアラインされている場合、msosは、アドレスへのサイズバイトの単一のメモリストア操作となります。*
* *そうでなければ、msosは、address ... address + size - 1への、それぞれ1バイトサイズの、サイズメモリストア演算のセットです。*。

1. *i.memストアをmsos*に設定します。そして
2. *i の*状態を PLAIN(*次の状態*)に更新します。

*上の遷移を取った後、メモリ・ストア操作はまだその値を持っていないことに注意してください。このトランジションを下のトランジションから分割することの重要性は、他のプログラム順後続のストア命令がこの命令のメモリフットプリントを観察できるようにすることです。*

*そして、それらが重ならない場合は、できるだけ早く（つまり、データレジスタの値が利用可能になる前に）順不同を伝搬させます。*

\_

**メモリストア操作値のインスタンス化** PLAIN(ストアMEMV(メモリ\_値, ストア\_継続))状態の命令インスタンス*iは*、常にメモリストア操作*i.mem storesの*値をインスタンス化することができます。  
アクション：。

1. メモリストア操作*i.memストアの*間で*mem値を*分割します；そして
2. *iの*状態をPENDING＿MEM\_STORES(ストア継続)に更新します。

**コミットストア命令 コミット**されていないストア命令または sc 命令のインスタンス *i* がコミットされていない場合、“scのストア操作をコミットして伝搬させる''というトランジションのコンテキストで、PENDING\_MEM\_STORES(*ストア継続*)の状態にある場合、コミットすることができます(伝搬とは混同しないように)。

1. *iは*データを完全に決定しています。
2. プログラム順前の条件分岐命令と間接ジャンプ命令をすべて終了します。
3. すべてのプログラムオーダー前のフェンスの命令と.swのセットが終了しました。
4. すべてのプログラムオーダー-前のfence.tsoの命令が終了しています。
5. すべてのプログラムオーダー前のロード-獲得 命令が終了しました。
6. すべてのプログラムオーダー前のストア獲得・解放の命令は終了しました。
7. *iが*スト-解放の場合は、すべてのプログラムオーダー前の命令が終了します。
8. すべてのプログラムオーダー前のメモリアクセス命令は、完全に決定されたメモリフット-プリントを持っています。
9. 失敗した sc を除くすべてのプログラムオーダー前のストア命令が開始されたため、空ではない *mem ストアを*持っています。
10. すべてのプログラムオーダー前のロード命令が開始されたため、空ではない*メモリロードが*あります。

アクション: *iが*コミットされたことを記録します。

*条件8が満たされると、条件9と10も満たされるか、あるいはいくつかの熱心な遷移を取った後に満たされることに注意してください。したがって、これらを要求してもモデルは強化されません。これらを要求することで、前のメモリアクセス命令が、その命令が次に取るであろう遷移である伝搬ストア動作の条件チェックのために、そのメモリ動作が見えるようにするのに十分な遷移を取ったことを保証し、その条件をより単純にします。*

**ストア操作の伝搬** 状態PENDING\_MEM\_STORES(ストア 継続)にあるコミットされた命令インスタンスiと、i.mem storesにある伝播されていないメモリ・ストア操作msoについて、以下の場合、msoを伝播することができます。

1. *mso*と重なるプログラム順前ストア命令のすべてのメモリストア操作は、すでに伝播しています。
2. *mso*と重なるプログラムオーダ前のロード命令のメモリロード操作がすべて満たされており、(ロード命令は)*再起動できません*(以下の定義を参照)。そして、
3. *mso を*転送することで満たされたすべてのメモリロード操作が完全に満たされます。

ここで、非終了命令インスタンス*jが再起動不可能な*場合。

1. *s* のストア命令 *s* とプロパゲートされていないメモリストア操作 *mso が*存在しない場合、「ストア操作のプロパゲート」トランジションを *mso* に適用すると *j が*再起動されます。
2. 未完了のロード命令 *l* と *l* のメモリロード操作 *mlo が存在しない場合、“*プロパゲーションされていないストアからの転送でメモリロード操作を満足させる''/”メモリからのメモリロード操作を満足させる''遷移の動作を *mlo* に適用すると(たとえ *mlo が*既に満足されていたとしても) *j が*再起動されてしまいます。

アクション。

1. 共有メモリの状態を *mso* で更新します。
2. *mso が*伝播されたことを示すために *i.mem ストアを*更新します。そして
3. この結果、コヒーレンスを侵害した投機的な命令を再起動する、つまり msoss'から満たされたi'のすべての非終了命令i'プログラムオーダー後iとすべてのメモリロード操作mlo'について、msoss'にmsoとオーバーラップし、msoからではないメモリストア操作スライスmsos'が存在し、msos'がiのプログラムオーダー後の命令からではない場合、i'とその再起動依存者を再起動します（未伝達ストアからの転送によるメモリロード操作の満足度を見ます）。

sc **のコミット**と伝播ストア操作 ハート hのPENDING\_MEM\_STORES continuation)の状態で、あるストアスライスmsossによって満たされたペアのlr i'を持つ、コミットされていないsc命令インスタンスiは、以下の場合、同時にコミットされ、伝播されることがあります。

1. i’は終了しました。
2. *i’に*転送された全てのメモリストア操作が伝搬されます。
3. コミットストア命令の条件が満たされていることを示します。
4. ストア動作を伝播させる条件が満たされています(sc命令は1つのメモリストア動作しかできないことに注意してください); そして
5. *msoss*からストア スライス*msosを*取得した場合、 msos*が*メモリに伝搬されてからどの時点でも、*h*ではないハートからのストアによって、共有メモリ内で*msosが*上書きされていないことを示します。

アクション：

1. コミットストア命令のアクションを適用します。
2. 伝搬ストア操作のアクションを適用します。

**遅い** sc 失敗PENDING\_MEM\_STORES（ストア・継続）の状態にあるsc命令インスタンスiは、メモリ・ストア・オペレーションを伝播していないので、常に失敗させることができます。アクション：

1. *i.memストアを*クリアします。そして
2. *i の*状態を PLAIN(ストア *継続(失敗))* に更新します。

*効率化のために、*rmem *ツールでは、*sc *トランジションのコミットと伝搬ストア操作ができない場合にのみ、この遷移を許可しています。これは許可される最終状態のセットには影響しませんが、対話的に探索する場合、sc*が*失敗した場合には、この遷移を待つ代わりに早期* sc 失敗*遷移を使用する必要があります。*

**ストアオペレーションの完了**  状態PENDING\_MEM\_STORES（ストア・コンティニュエーション）のストア命令インスタンスiは、i.memストアのすべてのメモリ・ストア操作が伝搬されているので、常に完了することができます（終了と混同してはいけません）。アクション：iの状態をPLAIN(ストア 継続(成功))に更新します。。

**AMO のサティスフィ、コミット、伝搬動作** PENDING\_MEM\_LOADS（ロード・コンティニュエーション）の状態にあるAMO命令インスタンスiは、以下の一連の遷移を間に挟まずに実行できる場合、そのメモリ・アクセスを行うことができます。

1. メモリからのメモリロード動作を満足させる
2. 完全な負荷操作
3. 疑似コード内部ステップ（0回か、それ以上）
4. メモリストアの操作値のインスタンス化
5. コミットストア命令
6. ストア操作の伝搬
7. 完全なストア操作

また、Finish命令の条件は、*iが*PLAIN(DONE)の状態であることを必要としないことを除いて、これらの遷移の後も保持されます。動作：上記の一連の遷移（Finish命令は含まない）を、遷移を挟まずに次々と実行します。

*プログラムオーダー前のストアは、AMOのロードに転送できないことに注意してください。これは単純に、上の遷移のシーケンスに転送遷移が含まれていないからです。しかし、仮にそれが含まれていたとしても、このシーケンスでは、ストア操作のトランジション「伝搬」を実行しようとすると失敗します。なぜなら、このトランジションでは、オーバーラップするメモリ・フットプリントに対するすべてのプログラム順序前のストア操作が伝搬される必要があり、転送ではストア操作が伝搬されないことが必要だからです。*

*また、AMO のストアをプログラムオーダ後続ロードに転送することはできません。上記の遷移を行う前は、AMOのストア動作は値を持たないため転送できず、上記の遷移を行った後はストア動作が伝播されるため転送できません。*

**コミットフェンス** 状態PLAIN(Fence(*種類*, 次\_状態))にあるフェンス命令インスタンス*iは*、もしあればコミットできます。以下

1. 通常のフェンスに.prがセットされていれば、プログラムオーダー前のロード命令はすべて終了します。
2. 通常のフェンスで、.pwが設定されている場合は、すべてのプログラムオーダー前のストアの命令が終了します。
3. もし*iが*fence.tsoであれば、すべてのプログラムオーダー前のロードとストアの命令は終了します。

アクション：

1. *i が*コミットされたことを記録します。そして、
2. *i の*状態を PLAIN(*次の状態*)に更新します。

**レジスタ読み取り** PLAIN(Read reg(*reg 名*, *read cont*))の状態の命令インスタンス*iは*、読み出す必要のあるすべての命令インスタンスがすでに期待された*reg名の*レジスタ書き込みを実行している場合、*reg名の*レジスタ読み出しを行うことができます。

*読み込みソースには、reg名の*各ビットについて、そのビットへの書き込みが可能な最も新しい（プログラム順の）命令インスタンスによるそのビットへの書き込みがある場合には、そのビットへの書き込みを含むようにします。そのような命令がない場合、ソースは*初期レジスタ状態*からの初期レジスタ値である。*reg値をリードソース*から組み立てた値とします。アクションとします。

1. *読み取りソース*と*reg値を*持つ*i.regの読み込み*に*reg名を*追加します。
2. *i の*状態を PLAIN(*read cont(reg 値))* に更新します。

**Register write** PLAIN(Write reg(reg name, reg value, next state))状態の命令インスタンスiは、常にreg名のレジスタライトを行うことができます。。アクション：

1. *i.reg の書き込み*に、*deps* と *reg の値を指定*して reg *名を*追加します。そして
2. *i の*状態を PLAIN(*次の状態*)に更新します。

ここで、depsは、i.regリードからのすべてのリードソースのセットと、iがすでに完全に満たされているロード命令インスタンスである場合に真となるフラグのペアです。

**擬似コード内部ステップ** 状態PLAIN(INTERNAL(次 *状態*))の命令インスタンス*i*は、常に疑似コード内部ステップを行うことができます。アクション: *i* の状態を PLAIN(次 *状態)* に更新します。

**完了命令** PLAIN(DONE)の状態にある未完了命令インスタンス*iは*、以下の場合に完了することができます。

1. *iが*ロード命令であれば
   1. すべてのプログラムオーダー前のロード取得命令が終了します。
   2. .srが設定されたプログラム順に前のFENCE命令がすべて終了します。
   3. プログラム順序が先の *fence*.tso 命令 *f* が終了していない場合，*f より*前のプログラム順序のロード命令はすべて終了します．
   4. *i*のメモリロード操作によって読み取られる値がコヒーレンス違反を起こさないことが保証されています。すなわち、任意のプログラムオーダー先行の命令インスタンス*i'*について、*i*と*i'*の間のストア命令プログラムオーダー先行から伝播されたメモリストア操作と、*i'*を含むiと*i'*の間のストア命令プログラムオーダー先行から*i*に転送された固定メモリストア操作のフットプリントを合わせて*/cfp*とすると、*/cfp*は*i*のメモリフットプリントにおける*/cfp*の補数となります。*/cfp*が空でない場合：。
      1. *i’*メモリフットプリントが完全に決定されています。
      2. *i’/cfp*と重複するプロパゲーションされていないメモリストア操作がありません。そして
      3. *i'*が*/cfp*とオーバーラップするメモリフットプリントを持つロードであれば、*/cfp*とオーバーラップする*i'*のすべてのメモリロード操作が満たされ、*i'*は再起動できません（命令が再起動できないであるかどうかを判断する方法については、ストア操作の伝播トランジションを参照）。

ここで、ストア命令がデータを完全に決定した場合、メモリストア動作は固定と呼ばれます。

1. *iは*完全に決定されたデータを有します；そして
2. *iがfenceで*ない場合は、プログラム順前の条件分岐と間接ジャンプ命令はすべて終了します。

アクション：。

1. *i*が条件付き分岐または間接ジャンプ命令である場合，実行されなかったパスを破棄します。すなわち，*命令ツリー*で実行された分岐/ジャンプによって到達できないすべての命令インスタンスを削除します。そして、
2. 命令が終了したことを記録します。つまり、finishedをtrueに設定します。

### B.3.6 制限事項

* このモデルはユーザレベルのRV64IとRV64Aをカバーしています。特に、位置ずれしたアトミック拡張機能“Zam”またはストア順序拡張機能“Ztso”はサポートされていません。

このモデルをRV32I/AやG、Q、C拡張に適応させるのは簡単なはずですが、私たちは試したことがありません。これには、ほとんどの場合、命令のためにSailコードを書く必要がありますが、同時実行モデルに変更があるとしても、最小限の変更で済みます。

* このモデルは通常のメモリアクセスのみをカバーしています（I/Oアクセスは扱いません）。
* このモデルはTLB関連の効果をカバーしていません。
* このモデルでは、命令メモリが固定されていることを前提としています。特に、Fetch命令の遷移はメモリロード操作を発生させず、共有メモリは遷移に関与しません。その代わりに、モデルは、メモリ位置が与えられたときにオペコードを提供する外部オラクルに依存します。
* このモデルでは、例外、トラップ、割り込みはカバーしていません。

# 参考文献

1. RISC-V ELF psABI仕様。[https://github.com/riscv/riscv-elf-psabi-doc/。](https://github.com/riscv/riscv-elf-psabi-doc/)
2. 32ビットマイクロプロセッサのためのIEEE規格。IEEE Std. 1754-1994、1994年。
3. G. M. アムダール、G. A. ブラウ、およびジュニア. F. P. ブルックス。IBM システム/360のアーキテクチャ。*IBM ジャーナルof R.*
4. ヴェルナー・ブッフホルツ、編者。*コンピュータシステムの計画。プロジェクト・ストレッチ*。マグロウヒルブック会社、1962年。
5. クーロシュ・ガラコールー、ダニエル・レノスキ、ジェームズ・ロードン、フィリップ・ギボンズ、アヌープ・グプタ、ジョン・ヘネシー。スケーラブルな共有メモリマルチプロセッサにおけるメモリの一貫性とイベントの順序付け。*第17回コンピュータ・アーキテクチャに関する国際シンポジウムの議事録にて。*ページ 15-26, 1990.
6. ティモシー・H・ハイルとジェームズ・E・スミス。選択的デュアルパス実行。技術報告書、ウィスコンシン大学マディソン校、1996年11月。
7. ANSI/IEEE Std 754-2008, 浮動小数点演算のための IEEE 標準、 2008.
8. マノリス・G・H・カテベニス,ロバート・W・シャーバーン・ジュニア,デビッド・A・パターソン,カルロ・H・・スパンキン.RISC II マイクロアーキテクチャ.*VLSI 83 会議議事録*, 1983年8月.
9. ヘソン・キム、オヌール・ムトゥル、ジャレッド・スターク、エール・N・パット。願望分岐：条件付き分岐と予測を組み合わせた、適応的な予測実行。。

*第38回IEEE/ACM国際マイクロアーキテクチャシンポジウム 議事録*, マイクロ 38, 43-54ページ, 2005.

1. A.クラウザー、T.オースティン、D.グルンヴァルト、B.カルダー。非述語命令セットアーキテクチャのための動的ハンモックの予言。*1998年の並列建築とコンパイル技術に関する国際会議の議事録では*、PACT '98、ワシントンDC、米国、1998.
2. デビッド・D・リー,シン・I・コング,マーク・D・ヒル,ジョージ・S・テイラー,デビッド・A・ホッジス,ランディ・H・カッツ,デビッド・A・パターソン。 マルチプロセッサ ワークステーション用の VLSI チップ セット - パート I: コプロセッサー・インターフェースとシンボリック処理のサポートを備えた RISC マイクロプロセッサー。*IEEE JSSC*, 24(6):1688--1698, 1989年12月.
3. オープンコア。オープンRISC 1000 アーキテクチャマニュアル、アーキテクチャバージョン 1.0、2012 年 12 月。
4. ハイディ・パン、ベンジャミン・ヒンドマン、クルステ・アサノヴィック。Lithe: 並列ライブラリの効率的な構成を可能にします。*並列性のホットトピックスに関する第1回USENIXワークショップ（HotPar '09）の議事録*、カリフォルニア州バークレー、2009年3月。
5. ハイジ・パン、ベンジャミン・ヒンドマン、クルステ・アサノヴィク。Litheで並列ソフトウェアを効率的に作成します。*プログラミング言語設計と実装に関する第31回会議で、*カナダのトロント、2010年6月。
6. デビッド・A・パターソンとカルロ・H・スパンコール。 RISC I: 縮小命令セットVLSIコンピューター。*ISCA* では、1981 年 443-458 ページ。
7. ラヴィ・ラージワールとジェームズ・R・グッドマン。投機的ロックの排除: マルチスレッドの実行の同時実行を可能にします。*マイクロアーキテクチャーに関する第34回ACM/IEEE国際シンポジウムの議事録では*、MICRO 34、294-305ページ。IEEEコンピュータ協会,2001年
8. バララム・シンハロイ、R.カラ、W.J.スターク、H.Q.ル、R.カルニョーニ、J.A.ヴァン・ノルストランド、B.J.ロンケッティ、J.ストゥーチェリ、J.リーンストラ、G.L.ガスリー、D.Q.グエン、B.ブラナー、C.F.マリノ、E.。IBM POWER7 マルチコア・サーバー・プロセッサー。*IBM 研究開発ジャーナル*, 55(3):1--1, 2011.
9. ジェームズ・E・ソーントン制御データ6600における並列運転．*1964年10月27-29日，秋季合同計算機会議，パートII：超高速計算機システム*，AFIPS '64（秋季，パートII），33-40ページ，1965年の*議事録*で．
10. マーク・トレンブレイ、ジェフリー・チャン、シャイレンダー・ショードリー、アンドリュー・W・コニリアーロ、シン・シェン・ツ。MAJC アーキテクチャ: 並列処理とスケーラビリティの合成。*IEEEマイクロ*,20(6):12-25,2000.
11. J.ツェンとK.アサノビッチ。エネルギー効率の高いレジスタアクセス。*集積回路とシステム設計シンポジウムの第13回,*377–384,マナウス,ブラジル,2000年9月。
12. デビッド・アンガー、リッキー・ブラウ、ピーター・フォーリー、デイン・サンプル、デビッド・パターソン。SOARのアーキテクチャ:RISCのスモールトーク。*ISCA*では,188-197ページ,アナーバー,MI,1984年。
13. アンドリュー・ウォーターマン*RISC-V圧縮によるエネルギー効率の向上とコードサイズの削減。*修士論文、カリフォルニア大学バークレー校、2011年。
14. アンドリュー・ウォーターマン*RISC-V命令セットアーキテクチャの設計*。博士論文、カリフォルニア大学バークレー校、2016年。
15. アンドリュー・ウォーターマン,ユンサップ・リー,デビッド・A・パターソン,クルステ・アサノビッチRISC-V 命令セットマニュアル、ボリューム I: 基本ユーザー・レベル ISA。テクニカルレポート UCB/EECS-2011-62, EECS部門, カリフォルニア大学バークレー校, 2011年5月.
16. アンドリュー・ウォーターマン,ユンサップ・リー,デビッド・A・パターソン,クルステ・アサノビッチRISC-V 命令セットマニュアル、ボリューム I: 基本ユーザー レベル ISA バージョン 2.0。技術報告書UCB/EECS-2014-54、カリフォルニア大学バークレー校EECS部門、2014年5月。

あとがき

この文章は RISC-V の非特権仕様を 柴田貴康　@shibatchii がRISC-Vアーキテクチャ勉強のため訳したものです。

原文は

https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdfです。

原文のライセンス表示

ですが、

This document is released under a Creative Commons Attribution 4.0 International License.

This document is a derivative of “The RISC-V Instruction Set Manual, Volume I: User-Level ISA

Version 2.1” released under the following license: ⃝c 2010–2017 Andrew Waterman, Yunsup Lee,

David Patterson, Krste Asanovi´c. Creative Commons Attribution 4.0 International License.

となっており、まずは「RISC-V非特権仕様 バージョン20191213」 日本語訳 @shibatchii

ということで進めます。

この文章は https://github.com/shibatchii/RISC-V

に置いてあります。

翻訳方法としては、原文のLaTexにちょっと手を入れてLigature(合字)をしないようにしてPDF生成、それをMS office wordで読み込みdoc形式へ変換し保存します。次にDeepLで全文自動翻訳をかけ、できてきた翻訳を原文を見つつ訳の修正や体裁整えを行っています。

フォントは今までメイリオを使ってましたが、見やすさを考えてUDフォントにしてあります。BIZ UDP明朝の方が原文に近い感じがしますが、読みにくいと感じたので BIZ UDPゴシックにしてあります。文字の大きさは基本9ポイントです。

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。

Twitter: @shibatchii

Mail:shibatchii@gmail.com

2021/03/07 @shibatchii　翻訳開始

2022/01/01 @shibatchii　翻訳完了

1. [↑](#footnote-ref-1)