はじめに

この文章は RISC-V の命令マニュアルを @shibatchii がRISC-Vアーキテクチャ勉強のためメモしながら訳しているものです。

原文は <https://riscv.org/specifications/> にある riscv-spec-v2.2.pdf です。

原文のライセンス表示

The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version

2.2", Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, May 2017.

Creative Commons Attribution 4.0 International License

この日本語訳のライセンスも原文のライセンスを引き継いで

RISC-V命令セットマニュアル 第一巻：ユーザーレベルISA文書 2.2版 日本語訳 @shibatchii

Creative Commons Attribution 4.0 International License

です。

<https://github.com/shibatchii/RISC-V>

に置いてあります。

英語は得意でないので誤訳等あるかもしれません。ご指摘歓迎です。

Google翻訳、Bing翻訳、Webilo翻訳、Exclite翻訳 を併用しながら翻訳し、勉強しています。

まずは意味が分からないところもあるかもしれませんが、ざっくり訳して2周位回ればまともになるかなと。

体裁とかは後で整えようと思います。

文章は以下の様に色分けしてます。

黒文字：翻訳した文書。

赤文字：@shibatchiiコメント。わからないところとか、こう解釈したとか。

青文字：RISC-Vにあまり関係なし。訳した日付とか、集中力が切れた時に書くヨタ話とか。

2018/03/26 @shibatchii

RISC-V命令セットマニュアル

第I巻：ユーザーレベルのISA

ドキュメントバージョン2.2

編集者：Andrew Waterman 1 、Krste Asanovic 1,2

1 SiFive Inc.、

カリフォルニア大学バークレー校のEECS部門 2 CS課

[andrew@sifive.com](mailto:andrew@sifive.com)、krste@berkeley.edu

2017年5月7日

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

このドキュメントはクリエイティブコモンズ帰属4.0国際ライセンスの下で公開されています。

このドキュメントは、「RISC-V命令セットマニュアル第1巻：ユーザーレベルISA バージョン2.1 」を次のライセンスの下でリリースした派生物です：(c) 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、クレステ・アサノビック クリエイティブコモンズ帰属4.0国際ライセンス。

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

配布条件はこれですね。<https://creativecommons.org/licenses/by/4.0/deed.ja>

制限が少ない大変良いですね。

-2018/03/26

序文

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

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

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

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

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

このバージョンのドキュメントの主な変更点は次のとおりです。

・このドキュメントの以前のバージョンは、元の作者が作成したクリエイティブコモンズ帰属4.0国際ライセンスの下でリリースされました。このドキュメントの今後のバージョンは、同じライセンスでリリースされる予定です。

・すべての拡張子を標準的な順序で最初に置くように章を再編成しました。

・説明と解説の改善。

・LUI / JALRとAUIPC / JALRペアのより高度なマクロオペレーションの融合をサポートするためのJALRに関する暗黙のヒント提案を修正しました。

↑なんじゃろ。JALRのところででてくるかな。

-2018/03/27

・ロード予約/ストア条件付きシーケンスに対する制約の明確化。

・コントロールとステータスレジスタ（CSR）のマッピングの新しいテーブル。

・fcsrの上位ビットの目的と動作を明確にしました。

・FNMADD.fmtおよびFNMSUB.fmt命令の説明が修正されました。これは、ゼロ結果の符号が正しくないことを示唆していました。

・命令FMV.S.XとFMV.X.SはFMV.W.XとFMV.X.Wに変わらなかったそれらの意味論とより一致するようにそれぞれ名前が変更されました。古い名前は引き続きツールでサポートされます。

↑意味が合うように命令を変えたってことか。

・より狭い（＜FLEN）浮動小数点値の指定された作用は、NaN-ボクシング・モデルを使っているより広いfレジスターをおさえました。

↑広いfレジスタに拡張(移行)されたってことかな。

・FMA（1、0、qNaN）の例外作用を定めました。

・P拡張が整数レジスタを使用した固定小数点演算の整数パックSIMD提案に再加工される可能性があることを示すノートを追加しました。

・Vベクトル命令セット拡張のドラフト提案。

・Nユーザレベルのトラップ拡張の早期草案。

・拡張された疑似命令リスト。

・RISC-V ELF psABI仕様に取って代わった呼び出し規約の章の削除 [1]。

・Cの拡張機能は凍結され、バージョン番号は2.0に変更されました。

-2018/03/28

文書 版2.1の序文

これは、RISC-Vユーザーレベルアーキテクチャを記載している文書の版2.1です。

凍結されたユーザーレベルISAベースと拡張IMAFDQ版2.0がこの文書の前の版から変わらなかった点に注意してください [36]、しかし、いくつかの仕様の穴は修正されました、そして、文章は改善されました。

ソフトウェアの規則にいくつかの変更が加えられました。

・解説セクションへの多数の追加と改良。

・各々の章のための別々のバージョン番号

・非常に長い命令フォーマットで rd 指定子を移動させないようにするために、64ビットを超える長い命令符号化への変更。

・CSR命令は、後で浮動小数点セクション（および付随する特権アーキテクチャマニュアル）に導入されるのとは対照的に、カウンタレジスタが導入される基本整数フォーマットで記述されます。

・SCALL命令とSBREAK命令は、それぞれECALLとEBREAKに名前が変更されました。それらのエンコーディングと機能は変更されていません。

・浮動小数点NaN処理の明確化、および新しい標準NaN値

・オーバーフローする浮動小数点から整数への変換によって返される値の明確化。

・LR / SCの明確化は、シーケンス内の圧縮命令の使用を含む、成功と必要な失敗を許しました。

・整数レジスタ数を削減し、MAC拡張をサポートする新しいRV32EベースのISA提案。

・改訂された呼び出し規約。

・ソフトフロート呼び出し規約のためのリラックスしたスタックアライメント、 およびRV32E 呼び出し規約の説明。

-2018/03/29

・C の圧縮された拡張のための修正案、バージョン1.9。

バージョン2.0の序文

これは、ユーザー isa の仕様の2番目のリリースであり、我々は、ベースユーザーの isa plus の一般的な拡張機能 (すなわち、IMAFD) の仕様は、将来の開発のために固定されたままにするつもりです。

この ISA 仕様のバージョン1.0 [35] 以降では、以下の変更が行われています。

・ISAはいくつかの標準拡張を持つ整数ベースに分割されています。

・命令フォーマットは、即時エンコードをより効率的にするために再編成されています。

・ベースISAは、リトルエンディアンのメモリシステムを持ち、ビッグエンディアンまたはバイエンディアンが非標準のバリアントであると定義されています。

・アトミック命令拡張で、Load-Reserved / Store-Conditional（LR / SC）命令が追加されました。

・AMOおよびLR / SCは、リリース一貫性モデルをサポートできます。

・フェンス命令は、より細かいメモリとI / O順序を提供します。

・fetch-and-XOR（AMOXOR）のAMOが追加され、部屋を作るためにAMOSWAPのエンコーディングが変更されました。

↑部屋を作る って場所を空けるっていう感じか。

・AUIPC命令は、PCに20ビットの上位の即値を追加し、現在のPC値のみを読み取るRDNPC命令を置き換えます。これにより、位置に依存しないコードが大幅に節約されます。

・JAL命令は、明示的なデスティネーションレジスタを持つUタイプフォーマットに移動し、J命令は、RAL = x0のJALに置き換えられ、なくなりました。

これにより、暗黙のデスティネーションレジスタを持つ唯一の命令が削除され、ベースISAからJ-Type命令フォーマットが削除されます。これに伴い、JALの到達範囲が縮小されますが、ベースISAの複雑さは大幅に軽減されます。

・JALR命令の静的ヒントが削除されました。このヒントは、標準の呼び出し規約に準拠したコードのrdおよびrs1レジスタ指定子では冗長です。

・ハードウェアを単純化し、補助情報を関数ポインタに格納できるように、JALR命令は計算されたターゲットアドレスの最下位ビットをクリアするようになりました。

・MFTX.S命令とMFTX.D命令は、それぞれFMV.X.SとFMV.X.Dに名前が変更されました。同様に、MXTF.SおよびMXTF.D命令は、それぞれFMV.S.XおよびFMV.D.Xに名前が変更されました。

・MFFSRおよびMTFSR命令は、それぞれFRCSRおよびFSCSRに名前が変更されました。

fcsrの丸めモードおよび例外フラグのサブフィールドに個別にアクセスするために、FRRM、FSRM、FRFLAGS、およびFSFLAGS命令が追加されました。

・FMV.X.SおよびFMV.X.D命令は、rs2ではなくrs1からオペランドがソースになります。この変更により、データパス設計が簡素化されます。

・FCLASS.SおよびFCLASS.D浮動小数点分類命令が追加されました。

・より単純なNaN生成および伝播方式が採用されています。

・RV32Iの場合、システムパフォーマンスカウンターは64ビット幅に拡張されており、上位および下位32ビットへの個別の読み取りアクセスが可能です。

・標準的なNOPおよびMVエンコーディングが定義されています。

-2018/03/30

・標準的な命令長エンコーディングは、48ビット、64ビット、および >64ビット命令で定義されています。

・128ビットアドレス空間バリアントRV128の説明が追加されました。

・32ビットの基本命令フォーマットの主なオペコードは、ユーザ定義のカスタム拡張のために割り当てられています。

・rdのデータをストアすることを示唆している誤植は、rs2を参照するように修正されました。

-2018/04/01

内容

序文

1 前書き 1

1.1 RISC-V ISAの概要。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 3

1.2 命令長符号化。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 5

1.3 例外、トラップ、および割り込み。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 7

2 RV32Iベース整数命令セット、バージョン2.0 9

2.1 基本整数サブセットのプログラマーのモデル。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 9

2.2 基本命令フォーマット。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 11

2.3即値エンコーディングの変形。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 11

↑valiantをどう訳すか

2.4 整数計算命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 13

2.5 コントロール転送命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 15

2.6 ロードとストア命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 18

2.7 メモリモデル。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 20

2.8 制御と状態レジスタ命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 21

2.9 環境呼び出しとブレークポイント。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 24

3 RV32Eベース整数命令セット、バージョン1.9 27

3.1 RV32Eプログラマのモデル。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 27

3.2 RV32E命令セット。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 27

3.3 RV32E拡張。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 28

4 RV64Iベース整数命令セット、バージョン2.0 29

4.1 レジスタの状態。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 29

4.2整数計算命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 29

4.3ロードおよびストア命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 31

4.4システム命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 32

5 RV128I基本整数命令セット、バージョン1.7 33

6 "M"整数乗算および除算標準拡張、バ​​ージョン2.0 35

6.1 乗算演算。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 35

6.2 除算演算。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 36

7 "A"原子命令の標準拡張、バ​​ージョン2.0 39

7.1原子命令の順序付けの指定。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 39

7.2ロード予約/ストア条件付き命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 40

7.3原子メモリ操作。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 43

↑Atomicってどういう意味だろう？ なんて訳すのがいいのか。

8 単精度浮動小数点"F"標準拡張 バージョン2.0 45

8.1 Fレジスタ状態。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 45

8.2 浮動小数点制御およびステータスレジスタ。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 47

8.3 NaNの生成と伝播。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 48

8.4 非正常な算術。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 49

8.5 単精度ロード命令とストア命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 49

8.6 単精度浮動小数点計算命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 49

8.7 単精度浮動小数点変換および移動命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 51

8.8 単精度浮動小数点比較命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 52

8.9 単精度浮動小数点分類命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 53

9 "D"倍精度浮動小数点標準拡張 バージョン2.0 55

9.1 Dレジスタの状態。 。 。。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 55

-2018/04/02

9.2 NaNのより狭い値の箱詰め。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 55

9.3倍精度ロード命令とストア命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 56

9.4倍精度浮動小数点計算命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 57

9.5倍精度浮動小数点変換および移動命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 57

9.6倍精度浮動小数点比較命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 59

9.7倍精度浮動小数点分類命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 59

10 "Q"四倍精度浮動小数点の標準拡張 バージョン2.0 61

10.1 四倍精度ロード命令とストア命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 61

10.2 四倍精度計算命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 62

10.3 四倍精度変換および移動命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 62

10.4 四倍精度浮動小数点比較命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 63

10.5 四倍精度浮動小数点分類命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 63

11 "L" 10進浮動小数点の標準拡張、バージョン0.0 65

11.1 10進浮動小数点レジスタ。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 65

12 "C" 圧縮された命令の標準拡張、バージョン2.0 67

12.1 概要。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 67

12.2 圧縮された命令フォーマット。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 69

12.3 ロードとストア命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 71

12.4 制御転送命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 74

12.5 整数計算命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 76

12.6 LR / SCシーケンスにおけるC命令の使用。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 80

12.7 RVC命令セットリスト。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 81

13 "B" ビット操作の標準拡張、バージョン0.0 85

14 "J" 動的に翻訳された言語の標準拡張、バージョン0.0 87

15 "T"トランザクションメモリの標準拡張、バージョン0.0 89

16 "P" 圧縮された(パックされた)SIMD命令の標準拡張、バージョン0.1 91

17 "V"ベクトル演算標準拡張、バージョン0.2 93

17.1 ベクターユニットの状態。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 93

17.2 要素のデータ型と幅。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 93

17.3 ベクトル構成レジスタ（vcmaxw、vctype、vcp）。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 95

17.4 ベクトルの長さ。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 97

17.5迅速な設定手順。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 97

18 "N” ユーザーレベル割り込みの標準的な拡張、バージョン1.1 101

18.1 追加のCSR。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 101

18.2ユーザステータスレジスタ(ustatus）。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 102

18.3 その他のCSR。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 102

18.4 N拡張命令。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 102

18.5 前後関係交換オーバーヘッドの削減。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 102

19 RV32 / 64G命令セット一覧 103

20 RISC-V アセンブリプログラマーズハンドブック 109

21 RISC-V拡張 113

21.1拡張用語。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 113

21.2 RISC-V拡張設計哲学(の考え方)。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 116

21.3 固定幅の32ビット命令フォーマット内の拡張。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 116

21.4整列した(アライン)64ビット命令拡張の追加。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 118

21.5 VLIWエンコーディング支援(提供?)。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 118

22 ISAサブセット命名規則 121

22.1大文字小文字の区別。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 121

22.2基本整数ISA。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 121

22.3命令拡張名。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。121

22.4バージョン番号。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 122

22.5 非標準拡張名。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 122

22.6 管理レベルの命令サブセット。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 122

22.7 管理レベルの拡張。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 122

22.8 サブセット命名規則。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 123

23歴史と謝辞 125

23.1 ISAマニュアルのリビジョン1.0履歴。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 125

23.2 ISAマニュアルのリビジョン 2.0履歴。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 126

23.3 改訂履歴2.1。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 128

23.4 改訂履歴2.2。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 128

23.5 資金調達。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 129

-2018/04/03

1ページ空き。次ベージへ。

第1章

前書き

RISC-V（「リスク5」と発音）は、コンピュータアーキテクチャの研究と教育をサポートするためにもともと設計された新しい命令セットアーキテクチャ（ISA）であり、

私たちが今望んでいるのは、業界標準の自由で解放されたアーキテクチャです。

RISC-Vを定義する私たちの目標は次のとおりです。：

・学界や産業界が自由に利用できる完全に解放されたISAです。

・実際のISAシミュレーションまたはバイナリ変換だけでなく、直接ネイティブハードウェア実装に適しています。

・ISAは特定のマイクロアーキテクチャスタイル（例えばマイクロコード化、インオーダー、デカップル、アウトオブオーダー）や実装技術（フルカスタム、ASIC、FPGAなど）を「オーバーアーキテクチャー」することなく、効率的にこれらのいずれかの実装を可能にします。

・ISAは、汎用のソフトウェア開発をサポートするために、カスタマイズされたアクセラレータまたは教育目的のためのベースとして使用可能な小さな基本整数ISAとオプションの標準拡張子に分かれています。

・改訂された2008 IEEE-754浮動小数点標準のサポート [14]。

・広範なユーザーレベルのISA拡張機能と特殊なバリアントをサポートするISA。

↑specialized variants. ってどう訳す？ 特別な変形、変異体、変数、展開

・アプリケーション、オペレーティングシステムカーネル、およびハードウェア実装の32ビットおよび64ビットアドレス空間の両バリエーション。

・異種マルチプロセッサを含む、高度に並列なマルチコアまたは多くのコアの実装を支援するISA。

↑multicoreとmanycoreの違いは？ どっちも沢山のコアのように思えるが

・オプションの可変長命令は、使用可能な命令エンコーディング空間を拡張と、オプションの高密度命令エンコーディングを提供し、パフォーマンス、静的コードサイズ、およびエネルギー効率を向上させます。

・ハイパーバイザー開発を容易にする完全仮想化ISA

・新しいスーパーバイザーレベルとハイパーバイザーレベルのISAデザインを使用して実験を簡単にするISA。

↑experiments 実験、試み なんだけどなんかしっくりこない

-----------------------------------------------------------------------------------------------------------------------------------

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

-----------------------------------------------------------------------------------------------------------------------------------

RISC-Vという名前は、UC Berkeley（RISC-I [23]、RISC-II [15]、SOAR [32]、SPUR [18]の最初の4つ）から5番目の主要なRISC ISA設計を代表するものです。

さまざまなデータ並列アクセラレータを含む一連のアーキテクチャ研究の支援がISA設計の明白な目標で、 "バリエーション"と "ベクトル"を表すローマ数字 "V"の使用についてもしゃれ(だじゃれ)を言います。

↑1つの文になってるけど、2つの文として考えた方がいいような。～明確な目標です。と ～言います。

-----------------------------------------------------------------------------------------------------------------------------------

我々は、実際のハードウェア実装の研究アイデアこの仕様書の初版以来11種類のシリコン製作を完了している）に特に関心のある研究と教育のニーズを支援するためにRISC-Vを開発しました。 (RISC-VプロセッサのRTLデザインは、Berkeleyの複数の学部および大学院クラスで使用されています)。

我々の現在の研究では、従来のトランジスタスケーリングの終了によって課された電力制約によって推進される、特殊(専門)で異種なアクセラレータへの移行に特に関心があります。

私たちは、私たちの研究努力を築くための非常に柔軟で拡張性のあるベースISAを求めました。

我々が繰り返し尋ねられた質問は、なぜ新しいISAを開発するのかということです。 既存の商用ISAを使用する最大の明白な利点は、研究と教育に活用できる開発ツールと移植されたアプリケーションの両方の、大規模で広くサポートされているソフトウェアエコシステムです。

↑既存のISAはツールとアプリケーションがたくさんあるよ。ってことかな

その他の利点としては、大量の文章と例題(チュートリアル)があります。

しかし、商用の命令セットを研究と教育に使用した経験では、実際にはこれらの利点が小さく、欠点を上回らないことです。

-2018/04/05

・商用ISAは専有です。

オープンなIEEE標準[2]であるSPARC V8を除き、商用ISAのほとんどの所有者は、知的財産を慎重に保護し、自由に利用可能な競争的実装を歓迎しません。

これは、ソフトウェアシミュレータのみを使用した学術研究や教育では大きな問題は少ないですが、実際のRTL実装を共有しようとするグループにとっては大きな懸念事項でした。

また、商用ISA実装のいくつかのソースを信頼したくないが、独自のクリーンルーム実装を作成することは禁止されているエンティティにとって、大きな懸案事項です。

すべてのRISC-V実装に第三者の特許侵害がないことを保証することはできませんが、RISC-V実装者を訴えるしようとしないことを保証することができます。

・商用ISAは特定の市場分野でのみ普及しています。

執筆時点での最も明白な例は、ARMアーキテクチャがサーバ空間で十分にサポートされていないことと、インテルx86アーキテクチャ（または他のほとんどすべてのアーキテクチャ）がモバイル領域で十分にサポートされていないことです。インテルとARMは互いの市場区分(領域)に参入しようとしています。

もう1つの例は、拡張可能なコアを提供するが、埋め込み領域に焦点を当てているARCとTensilicaです。

この市場分割は、特定の商用 ISA を実際にはソフトウェアエコシステムが特定のドメインにのみ存在し、他のユーザーのために構築しなければならないという利点を希釈します。

↑特定の所だけのソフトウェアエコシステムなので、ほかのユーザーのためにならないよ ってことかな

・商業的なISAが出て行きます。

以前の研究基盤は、もはや普及していない（SPARC、MIPS）、あるいはもはや生産段階でない（Alpha）、商用のISAまわりに構築されました。

これらは、活発なソフトウェア生態系の利益を失い、ISAおよび支援ツールに関する長引く知的財産の問題は、関心のある第三者がISAの支援を継続する能力を妨げます。

オープンISAは人気を失うかもしれないが、利害関係者はいずれもエコシステムの使用と開発を続けることができます。

・一般的な商用ISAは複雑です。

主要な商用ISAs（x86およびARM）は、ハードウェアで一般的なソフトウェアスタックおよびオペレーティングシステムをサポートするレベルを実装するのに非常に複雑です。

さらに悪いことに、ほとんどすべての複雑さは、真に効率を改善する機能ではなく、間違った、あるいは少なくとも旧式のISAデザインの決定によるものです。

・商用ISAだけでは、アプリケーションを起動するには十分ではありません。

市販のISAを実装する努力を費やしたとしても、そのISA用に既存のアプリケーションを実行するだけでは不十分です。

ほとんどのアプリケーションでは、ユーザーレベルのISAだけでなく、実行するための完全なABI（アプリケーション バイナリ インターフェイス）が必要です。

ほとんどのABIはライブラリに依存しており、オペレーティングシステムの支援に依存しています。

既存のオペレーティングシステムを実行するには、OSが期待する管理レベルのISAとデバイスインタフェースを実装する必要があります。

これらは通常、ユーザレベルのISAよりもはるかに明確ではなく、実装がかなり複雑です。

・一般的な商用ISAは拡張性のために設計されていませんでした。

支配的な商用ISAは、特に拡張性のために設計されておらず、その結果、命令セットが大きくなったため、命令エンコーディングの複雑さが増しています。

Tensilica（Cadence社買収）やARC（Synopsys社買収）などの企業では、拡張性を重視してISAとツールチェーンを構築していますが、汎用コンピューティングシステムではなく組み込みアプリケーションに注力しています。

・変更された商用ISAは新しいISAです。

私たちの主な目標の1つは、主要なISA拡張を含むアーキテクチャ研究をサポートすることです。

小さな拡張機能でもコンパイラを修正し、拡張機能を使用するためにソースコードからアプリケーションを再構築する必要があるため標準のISAを使用する利点は少なくなります。

新しいアーキテクチャ状態を導入するより大きな拡張では、オペレーティングシステムを変更する必要があります。

最終的に、変更された商用ISAは新しいISAになりますが、ベースISAのすべての遺産障害を引継ぎます。

私たちの立場は、ISAはおそらくコンピューティングシステムの中で最も重要なインターフェースであり、そのような重要なインターフェースが独自(専有)のものでなければならない理由はありません。

支配的な商用ISAは、30年以上前に既によく知られている命令セットの概念に基づいています。

ソフトウェア開発者はオープンな標準ハードウェア目標を目標とすることができ、商用プロセッサ設計者は実装品質を競うべきです。

我々はハードウェアの実装に適したオープンなISA設計を最初に検討しているところはありません。

↑我々は最初から遠いです。って言ってるんだけどいまいちわからん。

我々はまた、OpenRISCアーキテクチャ[22]に最も近いものとして、他の既存のオープンISA設計を検討しました。

私たちはいくつかの技術的な理由からOpenRISC ISAを採用することに反対しました。

・OpenRISCはより高性能な実装が複雑な条件コードと分岐遅延スロットを持っています。

・OpenRISCは、固定の32ビットエンコーディングと16ビット即値を使用しており、これにより、より高密度の命令エンコーディングが排除され、後でISAを拡張するためのスペースが制限されます。

・OpenRISCは2008年のIEEE 754浮動小数点標準の改訂をサポートしていません。

・OpenRISC 64ビットデザインは、私たちが始めたときに完成していませんでした。

白紙の状態から始めることによって、すべての目標を達成したISAを設計することができましたが、当初計画していたよりもはるかに労力がかかりました。

ドキュメント、コンパイラツールチェーン、オペレーティングシステムポート、参照(基準)ISAシミュレータ、FPGA実装、効率的なASIC実装、アーキテクチャ試験項目群、教材など、RISC-V ISAインフラストラクチャを構築するために多大な労力を投じました。

このマニュアルの最後の版以降、学界と産業界でRISC-V ISAかなりの取り込みが行われており、我々は標準を保護し促進するために非営利のRISC-V財団を創設しました。

RISC-V 財団ウェブサイト（http：// riscv.org）には、RISC-Vを使用した財団メンバーシップと様々なオープンソースプロジェクトに関する最新情報が掲載されています。

RISC-Vマニュアルは2つの巻で構成されています。

この巻は、オプションのISA拡張を含むユーザーレベルのISA設計について説明します。

2番目の巻は特権アーキテクチャを提供します。

-----------------------------------------------------------------------------------------------------------------------------------

このユーザーレベルのマニュアルでは、特定のマイクロアーキテクチャー機能や特権アーキテクチャーの詳細に依存することを排除することを目指しています。

これは、明快さと、代替実装のための最大の柔軟性を可能にするためです。

-2018/04/05

1.1 RISC-V ISAの概要

RISC-V ISAは、任意の実装に存在しなければならない基本整数ISAと、基本ISAへのオプションの拡張として定義されます。

基本整数ISAは、分岐遅延スロットがなく、オプションの可変長命令エンコーディングをサポートしている点を除いて、初期のRISCプロセッサのものと非常によく似ています。

ベースはコンパイラ、アセンブラ、リンカ、およびオペレーティングシステム（追加の管理者レベルの操作を含む）のための合理的なターゲットを提供するのに十分な最小限の命令セットに注意深く制限されているので、便利なISAおよびソフトウェアツールチェーン "スケルトン" よりカスタマイズされたプロセッサISAを構築することができます。

各基本整数命令セットは、整数レジスタの幅および対応するユーザアドレス空間のサイズによって特徴付けられます。

2つの主要な基本整数の変形、RV32IとRV64Iがあり、それぞれ第2章と第4章で説明します。これらは32ビットまたは64ビットのユーザーレベルのアドレス空間を提供します。

ハードウェアの実装およびオペレーティングシステムは、ユーザープログラムに対してRV32IとRV64Iの一方または両方のみを提供する場合があります。

第3章では、小型マイクロコントローラをサポートするために追加されたRV32Iベース命令セットのRV32Eサブセットの変形について説明します。

第5章では、将来の128ビットユーザアドレス空間をサポートする基本整数命令セットのRV128I変形について説明します。

基本整数命令セットは、符号付き整数値に対して2の補数表現を使用します。

-----------------------------------------------------------------------------------------------------------------------------------

大規模システムでは64ビットのアドレス空間が必要ですが、32ビットのアドレス空間は多くのエンベデッド・デバイスやクライアント・デバイスにとって今後も数十年の間十分なままであり、メモリ・トラフィックとエネルギー消費を削減することが望まれます。

さらに、教育目的では32ビットのアドレス空間で十分です。

最終的には128ビットのアドレス空間が必要になりますので、これをRISC-V ISAフレームワークに収めることができます。

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

-----------------------------------------------------------------------------------------------------------------------------------

基本整数ISAのサブセットは教育目的には役立つかもしれませんが、ベースが逸脱しているため、実際のハードウェアの実装をサブセット化するインセンティブはほとんどなく、メモリの不整合のサポートを省略し、すべてのSYSTEM命令を単一のトラップとして扱います 。

↑ここは何をいっているのかいまいちわからん。要約(意訳)すると、基本整数ISAは教育には役立つかも。でもこの基本はメモリのミスアライン(整列)を省略してるし、すべてのシステム命令を1トラップのように扱ってるので、実際のハードウェア実装のサブセットの動機(報酬)は少ないよ。 ってとこか

RISC-Vは、豊富なカスタマイズと特殊化をサポートするように設計されています。

基本整数ISAは、1つまたは複数のオプションの命令セット拡張で拡張できますが、基本整数命令は再定義できません。

RISC-V命令セット拡張機能を標準および非標準拡張に分割します。

標準拡張は一般的に有用であり、他の標準拡張と競合しないようにする必要があります。

非標準拡張は高度に専門化されているか、他の標準または非標準拡張と競合する可能性があります。

命令セット拡張は、基本整数命令セットの幅に応じて若干異なる機能を提供することがあります。

第21章では、RISC-V ISAを拡張するさまざまな方法について説明します。

また、RISC-Vベース命令と命令セット拡張の命名規則も開発しました（第22章で詳しく説明しています）。

より一般的なソフトウェア開発をサポートするために、整数の乗算/除算、アトミック演算、単精度浮動小数点演算と倍精度浮動小数点演算を提供する標準拡張のセットが定義されています。

基本整数ISAは "I"（整数レジスタの幅に応じてRV32またはRV64が前に付く）という名前で、整数計算命令、整数ロード、整数ストア、および制御フロー命令を含み、すべてのRISC-V実装で必須です。

標準の整数の乗算および除算の拡張は「M」と命名され、整数レジスタに保持された値を乗算およ​​び除算するための命令を追加します。

「A」で示される標準的な原子命令拡張は、プロセッサ間同期のためにメモリを原子的に読み取り、修正し、書き込む命令を追加する。

"F"で示される標準単精度浮動小数点拡張は、浮動小数点レジスタ、単精度計算命令、および単精度ロードおよびストアを追加します。

"D"で示される標準倍精度浮動小数点拡張は、浮動小数点レジスタを拡張し、倍精度の計算命令、ロード、およびストアを追加します。

整数ベースに加えて、これら4つの標準拡張（「IMAFD」）には、略語「G」が与えられ、汎用スカラ命令セットが提供される。

現在、RV32GとRV64Gはコンパイラツールチェーンのデフォルトターゲットです。

後の章では、これらおよびその他の計画された標準的なRISC-V拡張について説明します。

↑アトミックとアトミカリってどう訳すべきか。ARM AMBA AXI規格に アトミック転送ってあったような。あれと同じ？

ベース整数ISAおよび標準の拡張を超えて、新しい命令がすべてのアプリケーションに大きな利点をもたらすことはまれですが、特定のドメインにとっては非常に有益です。

エネルギー効率の懸念により、より専門化が強まっているので、ISA仕様の必要部分を簡素化することが重要であると考えています。

他のアーキテクチャでは通常、ISAを単一のエンティティとして扱いますが、命令が時間の経過と共に追加されるにつれて新しいバージョンに変更されますが、RISC-Vはベースと各標準拡張を時間の経過とともに一定に保ち、 その代わりにさらにオプションの拡張として新しい命令を階層化します。

たとえば、基本整数ISAsは、後続の拡張機能に関係なく、完全に支持されている独立型ISAsとして引き続き使用さ続けます。

-----------------------------------------------------------------------------------------------------------------------------------

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

1.2 命令長エンコーディング

ベースRISC-V ISAは、固定長の32ビット命令を備えています。これらの命令は、32ビット境界で自然に整列する必要があります。

しかし、標準のRISC-Vエンコーディング方式は、可変長命令を持つISA拡張をサポートするように設計されています。各命令は任意の数の16ビット命令区切りになり、16ビット境界で自然に整列されます。

第12章で説明した標準の圧縮ISA拡張命令は、圧縮された16ビット命令を提供することでコードサイズを縮小し、すべての命令（16ビットと32ビット）を16ビット境界で整列させてコード密度を向上させるための整列制約を緩和します。

図1.1に、標準的なRISC-V命令長エンコード規則を示します。

ベースISAのすべての32ビット命令は、最下位の2ビットが11に設定されています。

オプションの圧縮された16ビット命令セット拡張は、00,01、または10に等しい最低2ビットを有しています。

32ビット以上でエンコードされた標準命令セット拡張では、図1.1に示す48ビットおよび64ビットの長さ規則に従って、下位ビットが1に設定され追加されます。

80ビットと176ビットの間の命令長は、最初の5x16 ビットワードに加えて、16ビットワードの数を与えるビット [14:12] の3ビットフィールドを使用して符号化されます。

ビット[14:12]が111にセットされた符号化は、将来のより長い命令符号化のために予約されています。

-----------------------------------------------------------------------------------------------------------------------------------

圧縮されたフォーマットのコードサイズと省エネルギーを考えると、これを補足として追加するのではなく、ISA符号化方式の圧縮フォーマットをサポートしたいと考えましたが、より簡単な実装を可能にするために、必須です。

また、実験とより大きな命令セット拡張をサポートするために、より長い命令をオプションで許可したいと考えました。

私たちの符号化規約では、コアRISC-V ISAの符号化がより厳密に要求されていましたが、これにはいくつかの有益な効果があります。

標準G ISAの実装では、命令キャッシュに最上位30ビットしか保持する必要がありません（6.25％の節約）。

命令キャッシュリフィルでは、不正な命令例外の動作を保持するためにキャッシュに格納する前に、ロービットクリアのいずれかの命令が不正30bit 命令に記録される必要があります。

↑ここは何を言っているのかよくわからない。例外が起こった時には30bitは保持してねということかな。？？？

おそらくもっと重要なのは、基本ISAを32ビット命令語のサブセットに集約することによって、カスタム拡張で使用可能な空間が増えたことです。

第21章で説明したように、標準の圧縮命令拡張機能のサポートを必要としない実装では、≧32ビット命令セット拡張を超える標準のサポートを維持しながら、3つの30ビット命令スペースを32ビットの固定幅フォーマットにマップできます。さらに、実装はまた、長さが>32ビットを超える命令を必要としない場合、それはさらに4つの主要なオペコードを回復することができます。

16-bit (aa≠11)

xxxxxxxxxxxxxxaa

32-bit (bbb≠111)

xxxxxxxxxxxbbb11

xxxxxxxxxxxxxxxx

xxxxxxxxxx011111

…xxxx 48-bit

xxxxxxxxxxxxxxxx

xxxxxxxxx0111111

xxxxxxxxxxxxxxxx

…xxxx 64-bit

xnnnxxxxx1111111

xxxxxxxxxxxxxxxx

…xxxx (80+16\*nnn)-bit,nnn≠111

x111xxxxx1111111

xxxxxxxxxxxxxxxx

…xxxx Reserved for ≥192-bits

Base Address: base+4 base+2 base

図 1.1 RISC-V 命令長符号化

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

-----------------------------------------------------------------------------------------------------------------------------------

すべてのゼロビットを含む任意の長さの命令が正当でないという特徴は、ゼロメモリ領域への誤ったジャンプを素早くトラップするためです。

同様に、プログラムされていない不揮発性メモリデバイス、切断されたメモリバス、または壊れたメモリデバイスで観察される他の一般的パターンを補足するために、すべてのものを含む命令符号化は不正命令として確保します。

ベースRISC-V ISAにはリトルエンディアンのメモリシステムがありますが、非標準のバリアントではビッグエンディアンまたはバイエンディアンのメモリシステムを提供できます。

命令はメモリに格納され、各16ビット区画は、実装の自然なエンディアンに従ってメモリハーフワードに格納されます。

1つの命令を形成する区画は、命令仕様の最も低い番号のビットを保持する最下位の区画を有します、すなわちメモリシステムのエンディアンに関係なく小規模の区画配列に常に格納されます。

図1.2のコードシーケンスは、メモリシステムのエンディアンに関係なく、32ビット命令をメモリに正しく格納します。

-----------------------------------------------------------------------------------------------------------------------------------

リトルエンディアンシステムが現在商業的に支配的しているため、我々はRISC-Vメモリシステムのためにリトルエンディアンバイトオーダリングを選択しました（すべてのx86システム、iOS、Android、Windows for ARM）。

目立たない点は、我々はハードウェア設計者にとっては、リトルエンディアンのメモリシステムがより自然なものであることがわかったということです。

しかし、IPネットワーキングなどの特定のアプリケーション分野は、ビッグエンディアンのデータ構造で動作するため、非標準のビッグエンディアンまたはバイエンディアンの可能性を残しています。

長さエンコーディングビットが必ず最初にハーフワードアドレス順に現れるようにするために、メモリシステムエンディアンとは独立して命令区画がメモリに格納される順序を修正する必要があります。

これにより、最初の16ビット命令区画の最初の数ビットのみを調べることによって、命令フェッチユニットによって可変長命令の長さを迅速に決定することが可能になります。

リトルエンディアンのメモリシステムと命令区画の順序付けを修正した後は、命令形式のLSB位置に長さエンコードビットを配置し、オペコードフィールドの分割を避けることができるようになりました。

// x2レジスタの32ビット命令をx3が指し示す位置に格納します。

sh x2, 0(x3) // 命令の下位ビットを第1区画に格納します。

srli x2, x2, 16 // 上位ビットを下位ビットに移動し、x2を上書きします。

sh x2, 2(x3) // 上位ビットを2番目の区画に格納します。

図1.2： レジスタからメモリへの32ビット命令を格納する推奨コードシーケンス。

ビッグエンディアンとリトルエンディアンの両方のメモリシステムで正しく動作し、可変長命令セット拡張で使用される場合の誤整列アクセスを回避します。

1.3 例外、トラップ、および割り込み

現在のRISC-Vスレッドの命令に関連付けられた実行時に発生する異常な状態を参照するために、例外という用語を使用します。

トラップという用語は、RISC-Vスレッド内で発生する例外的条件によって引き起こされるトラップハンドラへの制御の同期転送を参照するために使用します。

トラップハンドラは通常より特権のある環境で実行されます。

現在のRISC-Vスレッドとは非同期に発生する外部イベントを参照するために、割り込みという用語を使用します。

処理されなければならない割込みが発生すると、割込み例外を受け取るためにいくつかの命令が選択され、続いてトラップが発生します。

次章の命令説明では、実行中に例外が発生する条件について説明しています。

これらがトラップに変換されるかどうかは実行環境に依存しますが、例外が通知されたときにはほとんどの環境で正確なトラップが実行されることが予想されます (ただし、浮動小数点例外を除き、標準浮動小数点数の拡張は、トラップを引き起こしません)。

-----------------------------------------------------------------------------------------------------------------------------------

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

-2018/04/14

1ページ空き。次ベージへ。

第2章

RV32Iベース整数命令セット、

バージョン2.0

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

解説の多くはRV64I変形にも当てはまります。

-----------------------------------------------------------------------------------------------------------------------------------

RV32Iは、コンパイラターゲットを形成し、最新のオペレーティングシステム環境をサポートするのに十分なものになるように設計されています。

ISAは最小限の実装で必要なハードウェアを削減するように設計されています。

RV32Iには47個のユニークな命令が含まれていますが、シンプルな実装では、8個のSCALL / SBREAK / CSRR \*命令を常に1つのSYSTEMハードウェア命令でカバーすることができますが、FENCEおよびFENCE.I命令をNOPとして、 ハードウェア命令数を合計38に減らします。

↑いまいちよくわからんが、RV32Iは47命令あるけど、おまとめすると38命令に減るよ。っていってるんだろうか

RV32Iはほとんどの他のISA拡張をエミュレートすることができます（アトミック性を追加するハードウェアサポートが必要なA拡張機能を除く）。

2.1基本整数サブセットのプログラマモデル

図2.1に、基本整数サブセットのユーザーが表示できる状態を示します。

整数値を保持する31個の汎用レジスタx1〜x31があります。

レジスタx0は定数0にハードワイヤードされています。

ハードワイヤードのサブルーチンリターンアドレスリンクレジスタはありませんが、標準ソフトウェア呼び出し規約ではレジスタx1を使用して呼び出し時にリターンアドレスを保持します。

RV32では、xレジスタは32ビット幅で、RV64では64ビット幅です。

このドキュメントでは、XLENという用語を使用して、xレジスタの現在の幅（32または64のいずれか）を参照します。

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

-2018/04/17

-----------------------------------------------------------------------------------------------------------------------------------

使用可能なアーキテクチャレジスタの数は、コードサイズ、パフォーマンス、およびエネルギー消費に大きな影響を与える可能性があります。

コンパイルされたコードを実行する整数ISAには16個のレジスタで十分ですが、3アドレス形式を使用して16ビットの命令で16個のレジスタで完全なISAをエンコードすることは不可能です。

2アドレス形式も可能ですが、命令数が増えてそれと効率が低下します。

基本的なハードウェアの実装を簡素化するために中間命令サイズ（Xtensaの24ビット命令など）を避け、32ビット命令サイズが採用されれば32の整数レジスタを維持するのは簡単でした。

整数レジスタの数が多いほど、ループ展開、ソフトウェアパイプライン、およびキャッシュタイリングを幅広く使用できる高性能コードのパフォーマンスも向上します。

↑キャッシュタイリングってなんだろ。キャッシュ領域を埋めるっていう意味かな？

これらの理由から、我々は基本ISA用のために32の整数レジスタを従来のサイズで選択しました。

動的レジスタの使用量は、頻繁にアクセスされるいくつかのレジスタによって支配される傾向があり、regfileの実装は頻繁にアクセスされるレジスタのアクセスエネルギーを削減するために最適化することができます[31]。

オプションで圧縮された16ビット命令フォーマットは主に8つのレジスタにしかアクセスせず、したがって高密度命令エンコーディングを提供することができ、追加の命令セット拡張は必要に応じて非常に大きなレジスタ空間（平坦かもしくは階層的に）をサポートすることができます。

資源に制約のある組み込みアプリケーションの場合、RV32Eサブセットを定義しました。このサブセットには16個のレジスタしかありません（第3章）。

XLEN-1 0

|  |
| --- |
| X0 / Zero |
| 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ユーザーレベルの基本整数レジスタの状態

-2018/04/18

2.2基本命令フォーマット

ベースISAには、図2.2に示すように、4つのコア命令フォーマット（R / I / S / U）があります。

すべて固定長32ビットであり、メモリ内の4バイト境界で整列しなければなりません。

もし対象アドレスが4バイト整列でない場合、命令不一致例外が、取得された分岐または無条件ジャンプで生成されます。

実行されない条件分岐に対して、命令フェッチ・不整列例外が生成されません。

↑この訳はなんか変。最初のNoがどこにかかってくるか。

-----------------------------------------------------------------------------------------------------------------------------------

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

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct7 | rs2 | rs1 | funct3 | rd | opcode | R-型 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Imm[11:0] | rs1 | funct3 | rd | opcode | I-型 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Imm[11:5] | rs2 | rs1 | funct3 | Imm[4:0] | opcode | S-型 |

|  |  |  |  |
| --- | --- | --- | --- |
| Imm[31:12] | rd | opcode | U-型 |

図2.2：RISC-Vの基本命令フォーマット

各即値サブフィールドは、通常行われる命令の即値フィールド内のビット位置ではなく、生成される即値のビット位置（imm [x]）でラベル付けされます。

-2018/04/19

RISC-V ISAはソース（rs1とrs2）とデスティネーション（rd）レジスタをすべてのフォーマットの同じ位置に保ち、デコードを簡単にします。

CSR命令（セクション2.8）で使用されている5ビット即値を除き、即値は常に符号拡張されており、一般に命令の最も左側の利用可能なビットに向かって詰め込まれ、ハードウェアの複雑さを軽減するために割り当てられています。

特に、すべての直の符号ビットは、符号拡張回路を高速化するため常に命令のビット31にあります。

-----------------------------------------------------------------------------------------------------------------------------------

レジスタ指定子のデコードは、通常、実装のクリティカルパス上にあります。したがって、すべてのレジスタ指定子をすべてのフォーマットの同じ位置に保持するように命令フォーマットが選択されました（RISC-IVと共有されるプロパティ 別名SPUR [18]）。

実際には、ほとんどの即値は小さいか、すべてのXLENビットが必要です(必要とします)。

我々は、通常の命令で利用可能なオペコード空間を増加させるために、非対称即時分割（通常の命令では12ビットと20ビットの特別なロード上の即値命令）を選択した。

↑(規則正しい命令12bitに加え20bitの特別なロード上位即値命令) なのかな

即値は、MIPS ISAのようにいくつかの即値に対してゼロ拡張を使用する利点を観察せず、ISAをできるだけ単純なものに保ちたいという理由で、符号拡張されています。

↑値はゼロ拡張はあまり良くなかったし、単純にしたかったので符号拡張にしたよ。

2.3 即時符号化変形

!即値のエンコード種類 の方がいいかな

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

SフォーマットとBフォーマットとの唯一の違いは、12ビットの即値フィールドを使用して、Bフォーマットの2の倍数で分岐オフセットを符号化するために使用されることです。

命令符号化された命令内のすべてのビットを、従来のようにハードウェアで1つ左にシフトする代わりに、中間ビット（imm [10：1]）および符号ビットは固定位置に留まり、Sフォーマットの最下位ビット（inst [ 7]）は、Bフォーマットの上位ビットを符号化します。

同様に、UフォーマットとJフォーマットとの唯一の違いは、20ビット即値が12ビット左にシフトしてU即値を形成し、1ビットがJ即値を形成することです。

UおよびJフォーマット即値における命令ビットの位置は、他のフォーマットと互いに重なり(共通部分)を最大にするように選択されます。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct7 | rs2 | rs1 | funct3 | rd | opcode | R-型 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | opcode | I-型 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-型 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-型 |

|  |  |  |  |
| --- | --- | --- | --- |
| Imm[31:12] | rd | opcode | U-型 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-型 |

図 2.3: 即値変形を示す RISC V の基本命令形式。

↑variantsは種類とかに訳した方がいいかな。即値の種類。しかしJ-typeとかなんちゅう並びだ。

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

-2018/04/20

31 30 20 19 12 11 10 5 4 1 0

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| -- inst[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[19:12] | -- 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命令で生成される即値の型

フィールドは、値を構成するために使用される命令ビットでラベル付けされています。

符号拡張は常にinst [31]を使用します。

-----------------------------------------------------------------------------------------------------------------------------------

符号拡張は、即値（特にRV64I）で最も重要な操作の1つであり、RISC-Vでは、すべての即値の符号ビットが常に命令のビット31に保持され、符号拡張が命令デコードと並行して進められます。

より複雑な実装では、分岐とジャンプの計算に別々の加算器が存在する可能性があるため、即値ビットの位置を命令のタイプ間で一定に保つことで恩恵を受けることはありませんが、我々は最も簡単な実装のハードウェアコストを削減する必要がありました。

動的ハードウェアマルチプレクサを使用する代わりにBおよびJ即値の命令エンコーディングでビットを回転(ローテート)することにより、即値に2を乗算することにより、命令信号ファンアウトおよび即時マルチプレクサコストを約2倍に削減します。

スクランブルされた即時エンコードでは、静的または事前コンパイル時に無視できる時間が追加されます。

命令を動的に生成するためには、若干の付加的なオーバーヘッドがありますが、最も一般的な短い前方分岐は簡単な即時エンコーディングを持っています。

- 2018/04/22

2.4 整数計算命令

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

↑destinationは”rd”、算術例外なし

-----------------------------------------------------------------------------------------------------------------------------------

RISC-V分岐を使用して多くのオーバーフローチェックを安価に実装できるようにするため、基本命令セットの整数算術演算のオーバーフローチェックに特別な命令セットサポートは含まれていませんでした。

符号なし加算のオーバーフローチェックでは、加算後に1つの追加の分岐命令しか必要としません。add t0、t1、t2; bltu t0、t1,オーバフロー。

符号付き加算では、1つのオペランドの符号が分かっている場合、加算後に1つの分岐のみが必要になります。addi t0、t1、+ imm; blt t0、t1、オーバフロー。

これは、即値オペランドを用いた加算の一般的なケースをカバーしています。

一般的な符号付き加算については、加算後に3つの追加命令が必要であり、他方のオペランドが負である場合にのみ、その和がオペランドの1つよりも小さくなければならないという考えを利用します。

add t0, t1, t2

slti t3, t2, 0

slt t4, t0, t1

bne t3, t4, overflow

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

整数レジスタ – 即時命令

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Imm[11:0] | rs1 | funct3 | rd | opcode |  |

12 5 3 5 7

I-即値[11:0] src ADDI/SLTI[U] dest OP-IMM

I-即値[11:0] src ANDI/ORI/XORI dest OP-IMM

ADDIは、符号拡張された12ビットの即値をレジスタrs1に加算します。

算術オーバーフローは無視され、結果は単に結果の低いXLENビットになります。

ADDR rd、rs1,0は、MV rd、rs1アセンブラ疑似命令の実装に使用されます。

レジスタrs1が符号拡張された即値よりも小さい場合、両方が符号付き数値として扱われる場合、SLTI（即時より小さい設定）はレジスタrdに値1を置きます。そうでない場合はrdに0が書き込まれます。

SLTIUは類似していまずが、その値を符号なし数として比較します（すなわち、即値は最初に符号拡張されてXLENビットに変換され、次に符号なし数として扱われます）。

注：SLTIU rd、rs1,1は、rs1がゼロの場合はrdを1に設定し、そうでない場合はrdを0に設定します（アセンブラ疑似命令SEQZ rd、rs）。

ANDI、ORI、XORIは、レジスタrs1と符号拡張12ビットの即値をビット単位でAND、OR、XORし、その結果をrdに格納する論理演算です。

注：XORI rd、rs1、-1は、レジスタrs1のビット単位の論理反転を実行します（アセンブラ疑似命令NOT rd、rs）。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| imm[11:5] | imm[4:0] | rs1 | funct3 | rd | opcode |  |

7 5 5 3 5 7

0000000 shamt[4:0] src SLLI dest OP-IMM

0000000 shamt[4:0] src SRLI dest OP-IMM

0100000 shamt[4:0] src SRAI dest OP-IMM

定数によるシフトは、I型形式の特殊化としてエンコードされます。

シフトされるオペランドはrs1であり、シフト量はI-即値フィールドの下位5ビットにエンコードされます。

右シフト型は、I即値の上位ビットで符号化されます。

SLLIは論理左シフトです（ゼロは下位ビットにシフトされます）。 SRLIは論理右シフトです（0は上位ビットにシフトされます）。 SRAIは算術右シフトです（元の符号ビットは空いている上位ビットにコピーされます）。

31 12 11 7 6 0

|  |  |  |  |
| --- | --- | --- | --- |
| imm[31:12] | rd | opcode |  |

20 5 7

U-即値[31:12] dest LUI

U-即値[31:12] dest AUIPC

LUI（上位即値ロード）は、32ビット定数を作成するために使用され、U型形式を使用します。

LUIはU-即値を転送先レジスタrdの上位20ビットに配置し、下位12ビットをゼロで埋めます。

AUIPC（PCに上位即値を追加）は、PC相対アドレスを作成するために使用され、U型形式を使用します。

AUIPCは、20ビットのU-即値から32ビットのオフセットを作成し、最下位の12ビットを0で埋め込み、このオフセットをpcに加え、結果をレジスタrdに配置します。

-----------------------------------------------------------------------------------------------------------------------------------

AUIPC命令は、制御フロー転送とデータアクセスの両方に対してPCからの任意のオフセットにアクセスするための2命令シーケンスをサポートしています。

AUIPCとJALRの12ビットイミディエイトを組み合わせることで、任意の32ビットPC相対アドレスに制御を移すことができ、AUIPCに加えて通常のロード命令またはストア命令の12ビット即値オフセットは32ビットPC相対データアドレス にアクセスできます。

現在のPCは、U-即値を0に設定することで取得できます。

PCを取得するためにJAL +4命令を使用することもできますが、より単純なマイクロアーキテクチャではパイプラインが切断されるか、より複雑なマイクロアーキテクチャではBTB構造が汚染される可能性があります。

整数レジスタ – レジスタ演算

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

funct7フィールドとfunct3フィールドは、操作の種類を選択します。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct7 | rs2 | rs1 | funct3 | rd | opcode |  |

7 5 5 3 5 7

0000000 src2 src1 ADD/SLT/SLTU dest OP

0000000 src2 src1 AND/OR/XOR dest OP

0000000 src2 src1 SLL/SRL dest OP

0100000 src2 src1 SUB/SRA dest OP

ADDとSUBはそれぞれ加算と減算を行います。

オーバーフローは無視され、結果の低XLENビットが出力先に書き込まれます。

SLTとSLTUは、符号付きと符号なしの比較をそれぞれ実行し、rs1 <rs2の場合はrdに1を、それ以外の場合は0を書き込みます。

SLTU rd、x0、rs2は、rs2がゼロでない場合にはrdを1に設定し、そうでない場合はrdをゼロに設定します（アセンブラ疑似演算SNEZ rd、rs）。

AND、OR、およびXORはビット単位の論理演算を実行します。

NOP命令

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | opcode |  |

12 5 3 5 7

0 0 ADDI 0 OP-IMM

NOP命令は、PCを進めることを除いて、ユーザーが見ることのできる状態を変更しません。

NOPはADDI x0、x0、0としてエンコードされます。

-----------------------------------------------------------------------------------------------------------------------------------

NOPsを使用すると、コードセグメントをマイクロアーキテクチャ上重要なアドレス境界に合わせたり、インラインコードを変更するための領域を確保したりすることができます。

NOPをエンコードするには多くの方法がありますが、わかりやすい逆アセンブリ出力と同様に、マイクロアーキテクチャの最適化を可能にする標準NOPエンコーディングを定義しています。

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

-- 2018/05/03

2.5コントロール転送命令

RV32Iには、無条件ジャンプと条件分岐の2種類の制御転送命令があります。

RV32Iの制御転送命令には、アーキテクチャ上見える遅延スロットはありません。

無条件ジャンプ

ジャンプおよびリンク（JAL）命令はJ型形式を使用し、J即時は2バイトの倍数で符号付きオフセットを符号化します。

オフセットは符号拡張され、ジャンプターゲットアドレスを形成するためにPCに追加されます。

したがって、ジャンプは ±1 MiB範囲を対象にすることができます。

JALは、ジャンプ（pc + 4）に続く命令のアドレスをレジスタrdに格納します。

標準ソフトウェア呼び出し規約では、x1をリターンアドレスレジスタとして使用し、x5を代替リンクレジスタとして使用します。

-----------------------------------------------------------------------------------------------------------------------------------

代替リンクレジスタは、通常のリターンアドレスレジスタを保持しながら、ミリコードルーチンを呼び出し（例えば、圧縮コードでレジスタを保存し復元する場合など）をサポートします。

レジスタ x5 は、標準の呼び出し規約で一時的にマップされるように代替リンクレジスタとして選ばれ、通常のリンクレジスタとは異なる1ビットのエンコーディングを持っています。

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

31 30 21 20 19 12 11 7 6 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| imm[20] | Imm[10:1] | imm[11] | Imm[19:12] | rd | opcode |  |

1 10 1 8 5 7

offset[20:1] dest JAL

間接ジャンプ命令JALR（ジャンプおよびリンクレジスタ）は、I型エンコーディングを使用します。

ターゲットアドレスは、レジスタrs1に12ビット符号付きI即値を加算し、結果の最下位ビットをゼロに設定することによって得られます。

ジャンプ後の命令のアドレス（pc + 4）がレジスタrdに書き込まれます。

結果が必要ない場合は、レジスタx0を宛先として使用できます。

1 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | opcode |  |

12 5 3 5 7

offset[11:0] base 0 dest JALR

-----------------------------------------------------------------------------------------------------------------------------------

無条件ジャンプ命令はすべて、PC相対アドレッシングを使用して独立した位置コードをサポートします。

JALR命令は、2命令シーケンスが32ビット絶対アドレス範囲のどこにでもジャンプできるように定義されています。

LUI命令は、まずターゲットアドレスの上位20ビットでrs1をロードし、次にJALRは下位ビットを加算することができます。

同様に、AUIPCそしてJALRは、32ビットのPC相対アドレス範囲のどこにでもジャンプすることができます。

JALR命令は、条件付き分岐命令とは異なり、12ビットの即値を2バイトの倍数として扱わないことに注意してください。

これは、ハードウェアで1つのより即値フォーマットを避けます。

↑これはハードウェアでもう１つの即値フォーマットを避けます。ハードウェアで区別できるようになってるってことか？

実際には、JALRのほとんどの用途は即時にゼロになるか、またはLUIまたはAUIPCとペアになるため、範囲のわずかな縮小は重要ではありません。

JALR命令は、計算されたターゲットアドレスの最下位ビットを無視します。

これは、ハードウェアをわずかに簡略化し、関数ポインタの下位ビットを使用して補助情報を格納することを可能にします。

この場合、エラーチェックのわずかな損失が発生する可能性がありますが、実際には誤った命令アドレスにジャンプすると、通常はすぐに例外が発生します

ベースrs1 = x0で使用すると、JALRを使用して、アドレス空間のどこからでも最低2KiBまたは最高2KiBのアドレス領域への単一命令サブルーチン呼び出しを実装できます。これは、小さなランタイムライブラリへの高速呼び出しを実装するために使用できます。

-- 2018/05/03

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

-----------------------------------------------------------------------------------------------------------------------------------

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

リターンアドレス予測スタックは、高性能命令フェッチユニットの共通の機能ですが、プロシージャコールおよびリターンに使用される命令を正確に検出する必要があります。

RISC-Vの場合、命令の使用に関するヒントは、使用されるレジスタ番号を介して暗黙的に符号化されます。

JAL命令は、rd = x1 / x5の場合にのみリターンアドレスをリターンアドレススタック（RAS）にプッシュする必要があります。

JALR命令は、表2.1に示すようにRASをプッシュ/ポップする必要があります。

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

表2.1：命令で使用されるレジスタ指定子にエンコードされたリターンアドレススタック予測ヒント。

上記では、レジスタがx1またはx5の場合にリンクが真になります。

-----------------------------------------------------------------------------------------------------------------------------------

他のいくつかのISAはリターンアドレススタック操作を導くために間接ジャンプ命令に明示的なヒントビットを追加しました。

これらのヒントに使用されるエンコード領域を減らすために、レジスタ番号と呼び出し規約に結びついている暗黙のヒントを使用します。

2つの異なるリンクレジスタ（x1とx5）がrs1とrdとして与えられると、RASはコルーチンをサポートするために両方プッシュされ、ポップされます。

rs1とrdが同じリンクレジスタ（x1またはx5のいずれか）である場合、RASはプッシュされ、シーケンスのマクロオペレーションの融合が可能になります。

lui ra、imm20; jalr ra、ra、imm12 および auipc ra、imm20; jalr ra、ra、imm12

条件付き分岐

すべての分岐命令は、B型命令形式を使用します。

12ビットのB-即値は符号付きオフセットを2の倍数で符号化し、現在のpcに加算してターゲットアドレスを与えます。

条件分岐範囲は±4 KiBです。

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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | immm[4:1] | imm[11] | opcode |  |

1 6 5 5 3 4 1 7

offset[12,10:5] src2 src1 BEQ/BNE offset[11,4:1] BRANCH

offset[12,10:5] src2 src1 BLT[U] offset[11,4:1] BRANCH

offset[12,10:5] src2 src1 BGE[U] offset[11,4:1] BRANCH

分岐命令は2つのレジスタを比較します。

BEQとBNEは、レジスタrs1とrs2がそれぞれ等しいか等しくなければ、分岐を取ります。

BLTとBLTUは、符号付きと符号なしの比較をそれぞれ使用して、rs1がrs2より小さい場合に分岐を取ります。

BGEとBGEUは、符号付きと符号なしの比較をそれぞれ使用して、rs1がrs2以上の場合は分岐を取ります。

なお、BGT、BGTU、BLE、およびBLEUは、それぞれオペランドをBLT、BLTU、BGE、およびBGEUに逆にすることで合成できます。

↑takeは取ります だけど、 分岐します と訳してもよいかもしれない。

-----------------------------------------------------------------------------------------------------------------------------------

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

ソフトウェアは、シーケンシャルコードパスが最も一般的なパスであり、頻度の低いコードパスが行の外に配置されるように、最適化する必要があります。

ソフトウェアは、少なくとも最初にそれらが遭遇したとき、後方ブランチが取られると予測し、前方ブランチが取られないと予測することも仮定しなければならない。

動的予測は、予測可能な分岐動作をすばやく学習する必要があります。

他のいくつかのアーキテクチャとは異なり、RISC-Vジャンプ（rd = x0のJAL）命令は、常に真の条件の条件付き分岐命令の代わりに常に無条件分岐に対して使用する必要があります。

RISC-VジャンプはPC相対でもあり、分岐よりもずっと広いオフセット範囲をサポートし、条件分岐予測テーブルを圧迫することはありません。

-----------------------------------------------------------------------------------------------------------------------------------

条件分岐は、2つのレジスタ間の算術比較演算を含むように設計されており（PA-RISCとXtensa ISAでも同様です）むしろ、条件コード（x86、ARM、SPARC、PowerPC）、または1つのレジスタをゼロ（Alpha、MIPS）と比較するだけ、または等価のみの2つのレジスタ（MIPS）を使用します。

この設計は、比較と分岐命令を組み合わせたものが通常のパイプラインに適合し、追加の条件コード状態または一時レジスタの使用を回避し、静的コード・サイズおよび動的命令フェッチ・トラフィックを削減するという観察によって動機付けられました。

もう1つのポイントは、ゼロとの比較では、（特に高度なプロセスでは静的ロジックに移行した後に）些細ではない回路の遅延必要なため、算術規模の比較と同じくらい高価です。

↑ゼロとの比較の方が簡単なような気がするが、ここではコストかかるよ～みたいになっている。ハテ？

融合された比較および分岐命令の別の利点は、分岐がフロントエンド命令ストリームの早期に観察され、より早期に予測できることが可能です。

同じ条件コードに基づいて複数の分岐を取ることができる場合には、条件コードを含む設計には多分利点がありますが、このケースは比較的まれであると考えられます。

我々は、命令符号化に静的分岐ヒントが含まれているか考慮したが含まれていませんでした。

これにより、動的予測への負担を軽減することができますが、最適な結果を得るためには、より多くの命令エンコード領域とソフトウェアプロファイリングが必要になり、実動実行がプロファイリング実行と一致しないとパフォーマンスが低下する可能性があります。↑分岐予測ミスるとパフォーマンス低下するよ

我々は、条件付き移動または予測命令が含まれないか考慮しました、予期しない短い前方分岐を効果的に置き換えることができました。

条件付き移動は2つの方が簡単ですが、例外（メモリアクセスと浮動小数点演算）を引き起こす条件付きコードでは使用が困難です。

条件付き実行制御は、システムに追加のフラグ状態、フラグの設定とクリアのための追加命令、およびすべての命令の追加の符号化オーバーヘッドを追加します。

条件付き移動命令と述語命令の両方が、述語が偽であれば、デスティネーションアーキテクチャレジスタの元の値を名前変更されたデスティネーション物理レジスタにコピーする必要があるため、暗黙的な第3のソースオペランドを追加して、順序外のマイクロアーキテクチャに複雑さを追加します。↑predicateは述語だけどどう訳すとそれらしくなるか？

また、分岐の代わりに述語を使用する静的なコンパイル時の決定は、特に予期しない分岐がまれであることを考えると、コンパイラのトレーニングセットに含まれない入力でパフォーマンスが低下し、分岐予測技術が向上するにつれて希少になります。

我々は、予測不能な短い順方向分岐を内部予測されたコードに動的に変換して、分岐予測ミス予測時にパイプラインをフラッシュするコストを回避するため商用プロセッサに実装されている[27]様々なマイクロアーキテクチャ技術が存在することを認識している[13,17,16]。

最も簡単な手法は、フェッチパイプライン全体ではなくブランチシャドウ内の命令をフラッシュするだけで、またはワイド命令フェッチまたはアイドル命令フェッチスロットを使用して両側から命令をフェッチすることによって、誤予測された短い前方分岐から回復するペナルティを軽減するだけです。

アウトオブオーダーコアのより複雑な技法は、分岐述部によって内部述部値が書き込まれた状態で、分岐シャドウの命令に内部述部を追加し、分岐および後続命令を他のコードに対して推論的かつ順不同で実行できるようにします[27]。

2.6ロードとストア命令

RV32Iは、ロードストア命令のみがメモリにアクセスし、算術命令がCPUレジスタ上でのみ動作するロードストアアーキテクチャです。

RV32Iは、バイトアドレスとリトルエンディアンの32ビットのユーザーアドレス空間を提供します。

実行環境は、アドレス空間のどの部分がアクセスに正当であるかを定義する。

x0の宛先を持つロードは、ロード値が破棄されても、例外を発生させ、他の副作用を引き起こす必要があります。

-- 2018/05/05

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | opcode |  |

12 5 3 5 7

offset[11:0] base 0 dest JOAD

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Imm[11:5] | rs2 | rs1 | funct3 | Imm[4:0] | opcode |  |

7 5 5 3 5 7

offset[11:5] src base width offset[4:0] STORE

ロード命令とストア命令は、レジスタとメモリの間に値を転送します。

ロードはI型形式でエンコードされ、ストアはSタイプです。

有効なバイトアドレスは、レジスタrs1を符号拡張された12ビットオフセットに加算することによって得られる。

ロードは、メモリからレジスタrdに値をコピーします。

ストアはレジスタrs2の値をメモリにコピーします。

LW命令は、メモリからrdに32ビット値をロードします。

LHはメモリから16ビットの値をロードし、次にrdに格納する前に32ビットに符号拡張します。

LHUはメモリから16ビットの値をロードしますが、rdに格納する前にゼロを32ビットに拡張します。

LBおよびLBUは、8ビット値についても同様に定義されます。

SW、SH、およびSB命令は、レジスタrs2の下位ビットから32ビット、16ビット、および8ビットの値をメモリに格納します。

最高のパフォーマンスを得るには、すべてのロードとストアの実効アドレスを各データ型（つまり、32ビットアクセスの場合は4バイト境界、16ビットアクセスの場合は2バイト境界）に合わせて自然に配置する必要があります。

ベースISAは、整列していないアクセスをサポートしますが、実装によっては非常に遅く実行される可能性があります。

さらに、自然に整列されたロードとストアはアトミックに実行されることが保証されますが、整列がずれたロードとストアはアトミック性を保証するために追加の同期を必要とされます。

↑atomically、atomicity アトミック、アトミック性ってどういうことなんだろうか

-----------------------------------------------------------------------------------------------------------------------------------

従来のコードを移植するときに、ずれたアクセスが必要になることがあり、多くのアプリケーションでパフォーマンスを向上させるには、任意の形式のパック SIMD 拡張を使用することが不可欠です。

通常のロードおよびストア命令による不整列アクセスをサポートするための我々の理論的根拠は、整列していないハードウェアサポートの追加を簡素化することである。

1つの選択肢は、ベースISAの不整列されたアクセスを禁止し、不整列されたアクセスを処理するための特別な命令、または不整列されたアクセスのための新しいハードウェアアドレッシングモードを提供することです。

特別な命令は、使用するのが難しく、ISAを複雑にし、しばしば新しいプロセッサ状態（例えば、SPARC VIS整列アドレスオフセットレジスタ）を追加するか、既存のプロセッサ状態へのアクセスを複雑にします（MIPS LWL / LWR部分レジスタ書き込み）。

さらに、ループ指向のパックドSIMDコードでは、オペランドの位置がずれている余分なオーバーヘッドが生じるため、ソフトウェアがオペランドの配置に応じて複数の形式のループを提供されるようになり、コード生成が複雑になり、ループ起動オーバーヘッドが増加します。

新しい不整列ハードウェアアドレス指定モードでは、命令符号化においてかなりのスペースを取るか、または非常に単純化されたアドレッシングモード（例えば、レジ​​スタ間接のみ）を必要とします。

我々は、不整列されたアクセスに対してアトミック性を要求するわけではないので、簡単な実装で、機械トラップとソフトウェアハンドラを使用して、一部またはすべての不整列されたアクセスを処理できます。

ハードウェアの不整列サポートが提供されている場合、ソフトウェアは通常のロードおよびストア命令を使用するだけでこれを利用できます。

ハードウェアは、ランタイムアドレスが整列されているかどうかによってアクセスを自動的に最適化することができます。

2.7メモリモデル

-----------------------------------------------------------------------------------------------------------------------------------

このセクションは、現在のプログラミング言語のメモリモデルを効率的にサポートできるように、RISC-Vメモリモデルが現在改訂中であるため、古くなっています。

修正された基本メモリモデルには、少なくとも同じハートからの同じアドレスへのロードを並べ替えることができず、命令間の構文データ依存性が尊重されることを含む、さらなる順序制約が含まれます。

↑ hart ハート ってどこを指しているのか？

ベースRISC-V ISAは、単一のユーザアドレス空間内で複数の同時実行スレッドをサポートします。

各RISC-Vハードウェアスレッドまたはハートは、独自のユーザーレジスタステートとプログラムカウンタを持ち、独立したシーケンシャルな命令ストリームを実行します。

実行環境は、RISC-Vハートの作成および管理方法を定義します。

RISC-Vハーツは、各実行環境の仕様で個別に文書化されている実行環境への呼び出しを介して、または共有メモリシステムを介して直接、他のハートと通信して同期することができます。

RISC-VハーツはI / Oデバイスとやりとりすることも、I / Oに割り当てられたアドレス空間の一部にロードやストアを介して間接的に相互に対話することもできます。

↑ だから、hart harts ってなんなのよ

-----------------------------------------------------------------------------------------------------------------------------------

ハートという用語は、ソフトウェア管理のスレッドコンテキストとは対照的に、ハードウェアスレッドを明確かつ簡潔に記述するために使用します。

↑ hartやっと出てきた。先に言わんかーい （＾＾；）。ハードウェアの実行単位とか並列実行が何個あるかとかのようだな。

ベースRISC-V ISAでは、各RISC-Vハートは、プログラム順に順次実行されるかのように、それ自身のメモリ操作を観察します。

RISC-Vはハート間の緩やかなメモリモデルを持ち、異なるRISC-Vハーツからのメモリ操作の順序を保証するために明示的なFENCE命令を必要とします。

第7章では、追加の同期操作を提供するオプションのアトミックメモリ命令拡張 "A"について説明します。

--2018/05/08

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | PI | PO | PR | PW | SI | SO | SR | SW | rs1 | funct3 | rd | opcode |  |

4 1 1 1 1 1 1 1 1 5 3 5 7

0 predecessor successor 0 FENCE 0 MISC-MEM

↑ 前任者、先行 後任、後続 囲い 多様な-メモリ

フェンス命令は、他のRISC-Vハートや外部デバイスやコプロセッサで見られるように、デバイスI / Oとメモリアクセスを順序付けるために使用されます。

デバイス入力（I）、デバイス出力（O）、メモリ読み出し（R）、およびメモリ書き込み（W）の任意の組み合わせは、それらの任意の組み合わせに関して順序付けられます。

正式には、他のRISC-Vハートまたは外部装置は、フェンスの前に設定された先行のオペレーションの前に、フェンスに続いて後続セットのオペレーションを観察することはできません。

↑非公式に、他の RISC-V ハートまたは外部デバイスは、フェンスの前に設定された前任者の任意の操作がフェンスに続いて、後続のセットで任意の操作を観察することができます。 どこで切るかによって真逆の訳になるような？

実行環境は、どのI/O操作が可能であるか、特にどのロード命令およびストア命令が、メモリ読み取りおよび書き込みではなく、それぞれデバイス入力およびデバイス出力操作として処理および順序付けされるかを定義します。

例えば、メモリマップされたI/Oデバイスは通常、RおよびWビットではなくI/Oビットを使用して順序付けされたキャッシュされていないロードおよびストアによってアクセスされます。

命令セット拡張はまた、フェンス内のIおよびOビットを使用して順序付けられる新しいコプロセッサI/O命令を記述することもできます。

フェンス命令のimm [11：8]、rs1、およびrdの未使用フィールドは、将来の拡張でより細かい微量なフェンス用に予約されています。

前方互換性のために、基本実装はこれらのフィールドを無視し、標準ソフトウェアはこれらのフィールドをゼロにしなければいけません。

--2018/05/09

-----------------------------------------------------------------------------------------------------------------------------------

私たちは、単純なマシン実装から高性能を可能にするリラックスメモリモデルを選択しましたが、しかし、完全にリラックスしたメモリモデルはプログラミング言語メモリモデルをサポートするには弱すぎるため、メモリモデルが強化されています。

リラックスしたメモリモデルは、将来のコプロセッサまたはアクセラレータの拡張機能と最も互換性があります。

I / O命令をメモリR / W命令から分離して、デバイスドライバハート内の不要なシリアル化を回避し、追加のコプロセッサまたはI / Oデバイスを制御する代わりの非メモリパスをサポートします。

単純な実装では、先行フィールドと後続フィールドを無視して、すべての操作で常に保守フェンスを実行することができます。

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | opcode |  |

12 5 3 5 7

０ ０ FENCE.I ０ MISC-MEM

FENCE.I命令は、命令ストリームとデータストリームの同期に使用されます。

RISC-Vは、命令メモリへのストアが、FENCE.I命令が実行されるまで、同じRISC-Vハートの命令フェッチでは表示されることを保証しません。

FENCE.I命令では、RISC-Vハートの後続の命令フェッチで、同じRISC-Vハートにすでに表示されている以前のデータストアがあることを保証するだけです。

FENCE.Iは、他のRISC-Vハーツの命令フェッチがマルチプロセッサシステム内のローカルハートストアを監視することを保証していません。

すべてのRISC-Vハートで命令メモリへのストアを表示させるには、すべてのリモートRISC-VハートがFENCE.Iを実行するように要求する前に、書き込みハートがデータFENCEを実行する必要があります。

FENCE.I命令のimm [1：0]、rs1、およびrdの未使用フィールドは、将来の拡張でより細かいフェンス用に予約されています。

前方互換性のために、基本実装はこれらのフィールドを無視し、標準ソフトウェアはこれらのフィールドをゼロにしなければなりません。

-----------------------------------------------------------------------------------------------------------------------------------

FENCE.I命令は、さまざまな実装をサポートするように設計されています。

簡単な実装では、FENCE.Iが実行されるときにローカル命令キャッシュと命令パイプラインをフラッシュすることができます。

より複雑な実装では、すべてのデータ（命令）キャッシュミスで命令（データ）キャッシュを詮索するか、またはローカルストア命令によって書き込まれているときにプライマリ命令キャッシュからのラインを無効にするために包括的な統一プライベートL2キャッシュを使用することがあります。

命令キャッシュとデータキャッシュがこのようにコヒーレントに保たれている場合は、パイプラインのみをFENCE.Iでフラッシュする必要があります

私たちは(MAJC [30]のように)、「ストア命令語」命令は考慮しましたが、含めませんでした。

JITコンパイラは、単一のFENCE.Iの前に命令の大きなトレースを生成し、Iキャッシュに存在しないことが分かっているメモリ領域に変換された命令を書き込むことによって、命令キャッシュの詮索/無効化オーバーヘッドを償却することができます。

2.8コントロールおよびステータスレジスタの命令

システム命令は、特権アクセスを必要とし、I型命令フォーマットを使用して、符号化されるシステム機能にアクセスするために使用されます。

これらは2つの主要なクラスに分けることができます：

これらは、アトミックにリード・モディファイ・ライト・コントロールおよびステータス・レジスタ（CSR）と、その他すべての潜在的に特権のある命令です。

CSR命令は、次のセクションで説明する2つのユーザーレベルのSYSTEM命令を使用して、このセクションでは説明されます。

-----------------------------------------------------------------------------------------------------------------------------------

SYSTEM命令は、より単純な実装が常に単一のソフトウェアトラップハンドラにトラップできるように定義されています。

より洗練された実装は、ハードウェア内の各システム命令のより多くを実行することができるかもしれません。

CSR命令

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

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| csr | rs1 | funct3 | rd | opcode |  |

12 5 3 5 7

source/dest source CSRRW dest SYSTEM

source/dest source CSRRS dest SYSTEM

source/dest source CSRRC dest SYSTEM

source/dest uimm[4:0] CSRRWI dest SYSTEM

source/dest uimm[4:0] CSRRSI dest SYSTEM

source/dest uimm[4:0] CSRRCI dest SYSTEM

CSRRW（アトミック 読み取り/書き込み CSR）命令は、CSRおよび整数レジスタの値をアトミックに交換します。

CSRRWはCSRの古い値を読み取り、その値をXLENビットにゼロ拡張した後、整数レジスタrdに書き込みます。

rs1の初期値はCSRに書き込まれます。

rd = x0の場合、命令はCSRを読み取らず、CSRの読み込みで発生する可能性のある副作用を引き起こさないものとします。

CSRRS（アトミック リードおよびセット ビット CSR）命令は、CSRの値を読み取り、その値をゼロ拡張してXLENビットにし、それを整数レジスタrdに書き込みます。

整数レジスタrs1の初期値は、CSRに設定するビット位置を指定するビットマスクとして扱われます。

CSRビットが書き込み可能である場合、rs1の上位ビットはCSRに対応するビットを設定します。

CSRの他のビットは影響を受けません（ただし、CSRは書かれたことによる副作用があるかもしれません）。

CSRRC（CSRのアトミックリードとクリアビット）命令はCSRの値を読み取り、値をゼロ拡張してXLENビットにし、それを整数レジスタrdに書き込みます。

整数レジスタrs1の初期値は、CSRでクリアされるビット位置を指定するビットマスクとして扱われます。

CSRビットが書き込み可能である場合、rs1の上位ビットはCSR内の対応するビットをクリアします。

CSRの他のビットは影響を受けません。

CSRRSとCSRRCの両方について、rs1 = x0ならば、命令はCSRにまったく書き込まない。

  したがって、読み取り専用CSRへのアクセスで違法命令の例外を発生させるなど、CSR書き込みで発生する可能性のある副作用を引き起こさないものとします。

rs1がx0以外のゼロの値を保持するレジスタを指定する場合、命令は変更されていない値をCSRに書き戻そうと試み、付随する副作用を引き起こすことに注意してください。

CSRRWI、CSRRSI、およびCSRRCIバリアントは、CSRRW、CSRRS、およびCSRRCにそれぞれ類似していますが、ただし、整数レジスタの値ではなくrs1フィールドにエンコードされた5ビットの符号なし即値(uimm[4:0]) フィールドをゼロ拡張したXLENビット値を使用してCSRを更新する点が異なります。

CSRRSIとCSRRCIの場合、uimm [4：0]フィールドがゼロの場合、これらの命令はCSRに書き込まれず、CSR書き込みで発生する可能性のある副作用を引き起こしません。

CSRRWIの場合、rd = x0の場合、命令はCSRを読み取らず、CSRの読み込みで発生する可能性のある副作用を引き起こしません。

廃止されたカウンタ、instretなどのいくつかのCSRは、命令実行の副作用として変更されることがあります。

↑retired counter って 退役とか引退とかいう意味だと思うが、引退したカウンタって何？

このような場合、CSRアクセス命令がCSRを読み取ると、CSRの命令はその命令の実行前に値を読み取ります。

CSRアクセス命令がCSRを書き込む場合、更新は命令の実行後に行われます。

特に、1つの命令によってinstretに書き込まれた値は、次の命令によって読み取られる値になります。

（すなわち、最初の命令のリタイアに起因するinstretの増分は、新しい値の書き込みの前に行われます。）。

CSR CSRR rd、csr を読み取るためのアセンブラ疑似命令は、CSRRS rd、csr、x0としてエンコードされます。

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

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

CSRS / CSRC csr、rs1; CSRSI / CSRCI csr、uimm。

タイマとカウンタ

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| csr | rs1 | funct3 | rd | opcode |  |

12 5 3 5 7

RDCYCLE[H] 0 CSRRS dest SYSTEM

RDTIME[H] 0 CSRRS dest SYSTEM

RDINSTRET[H] 0 CSRRS dest SYSTEM

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

↑ a number of を 多数のと訳すか、いくつかの と訳すか ？

RDCYCLE擬似命令は、ハートが過去の任意の開始時刻から実行されており、プロセッサコアが実行するクロックサイクル数のカウントを保持し、サイクルCSRの低XLENビットを読み込みます。

！低XLENビットとは64bitの下位32bitの事

RDCYCLEHは、同じサイクルカウンタのビット63-32を読み取るRV32I専用の命令です。

基礎となる64ビットカウンタは実際にはオーバーフローしません。

サイクルカウンタが進む速度は、実装および動作環境によって異なります。

実行環境は、サイクルカウンタがインクリメントする現在のレート（サイクル/秒）を決定する手段を提供する必要があります。

RDTIME擬似命令は、過去の任意の開始時刻から経過したウォールクロックリアルタイムをカウントする time CSRの低XLENビットを読み取ります。

RDTIMEHは、同じリアルタイムカウンタのビット63-32を読み取るRV32I専用の命令です。

基礎となる64ビットカウンタは実際にはオーバーフローしません。

実行環境は、リアルタイムカウンタの周期（秒/ティック）を決定する手段を提供する必要があります。

期間は一定でなければなりません。

単一のユーザアプリケーション内のすべてのハーツのリアルタイムクロックは、リアルタイムクロックの1チック以内に同期される必要があります。

環境は、クロックの精度を決定する手段を提供する必要があります。

RDINSTRET擬似命令は、instret CSRの下位XLENビットを読み取ります。これは、過去の任意の開始点からこのハートによって廃止された命令の数をカウントします。

RDINSTRETHは、同じ命令カウンタのビット63-32を読み取るRV32I専用の命令です。

実際にオーバーフローすることのない根本的な64ビットのカウンタです。

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

again:

rdcycleh x3

rdcycle x2

rdcycleh x4

bne x3, x4, again

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

-----------------------------------------------------------------------------------------------------------------------------------

これらの基本カウンタは、基本的なパフォーマンス分析、適応的かつ動的な最適化、およびアプリケーションがリアルタイムストリームで動作するために不可欠であるため、すべての実装で提供することを義務づけています。

パフォーマンスの問題を診断するために追加のカウンタを用意する必要があり、これらのカウンタは、ユーザーレベルのアプリケーションコードから低オーバーヘッドでアクセスできるようにする必要があります。

我々はカウンターはRV32でも64ビット幅であることを必要とし、それ以外の場合は、値がオーバーフローしたかどうかをソフトウェアが判断することは非常に困難です。

ローエンドの実装では、各カウンタの上位32ビットは、下位32ビットのオーバーフローによってトリガされるトラップハンドラによってインクリメントされるソフトウェアカウンタを使用して実装できます。

上で説明したサンプルコードは、個々の32ビット命令を使用して完全な64ビット幅の値を安全に読み取る方法を示しています。

アプリケーションによっては、同じ瞬間に複数のカウンタを読み取ることができることが重要です。

マルチタスク環境で実行すると、カウンタを読み取ろうとしている間にユーザースレッドがコンテキスト切り替えを受ける可能性があります。

1つの解決策は、ユーザスレッドが他のカウンタを読み取る前後にリアルタイムカウンタを読み取って、シーケンスの途中でコンテキストスイッチが発生したかどうかを判断することです。この場合、読み取りを再試行できます。

ユーザスレッドがカウンタ値をアトミックにスナップショットできるように出力ラッチを追加することを検討しましたが、特に、より豊富なカウンタセットを使用する実装では、ユーザコンテキストのサイズが大きくなります。

-- 2018/05/13

2.9 環境呼び出しとブレークポイント

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct12 | rs1 | funct3 | rd | opcode |  |

12 5 3 5 7

ECALL 0 PRIV 0 SYSTEM

EBREAK 0 PRIV 0 SYSTEM

ECALL命令は、通常はオペレーティングシステムであるサポート実行環境への要求を行うために使用されます。

システムのABIは、環境要求のパラメータがどのように渡されるかを定義しますが、通常、これらは整数レジスタファイルの定義された場所にあります。

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

-----------------------------------------------------------------------------------------------------------------------------------

ECALLとEBREAKは、以前はSCALLとSBREAKという名前でした。

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

-- 2018/05/13

-- 空ページを置いて次ページへ。

第3章

RV32Eベース整数命令セット、バージョン1.9

この章では、組み込みシステム用に設計されたRV32Iの縮小版であるRV32E基本整数命令セットについて説明します。

主な変更点は、整数レジスタの数を16に減らし、RV32Iで必須のカウンタを削除することです。

この章では、RV32EとRV32Iの違いについてのみ説明しますので、第2章の後にお読みください。

-----------------------------------------------------------------------------------------------------------------------------------

RV32Eは、組み込みマイコン用にさらに小型の基本コアを提供するように設計されています。

このドキュメントのバージョン2.0でこの可能性について言及しましたが、最初はこのサブセットの定義に抵抗しました。

しかし、可能な限り最小限の32ビットマイクロコントローラの需要があり、この領域で多様化を先取りするために、RV32I、RV64I、およびRV128Iに加えて、RV32Eを4番目の標準ISAとして定義しました。

Eバリアントは、32ビットのアドレス空間幅に対してのみ標準化されています。

3.1 RV32Eプログラマのモデル

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

-----------------------------------------------------------------------------------------------------------------------------------

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

-----------------------------------------------------------------------------------------------------------------------------------

この変更には、異なる呼び出し規約とABIが必要です。

特に、RV32Eはソフトフロート呼び出し規約でのみ使用されます。

ハードウェア浮動小数点を持つシステムでは、Iベースを使用する必要があります。

3.2 RV32E命令セット

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

-----------------------------------------------------------------------------------------------------------------------------------

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

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

-----------------------------------------------------------------------------------------------------------------------------------

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

3.3 RV32E拡張

RV32EはM、A、とCのユーザーレベルの標準拡張で拡張できます。

-----------------------------------------------------------------------------------------------------------------------------------

RV32Eサブセットでハードウェア浮動小数点をサポートするつもりはありません。

ハードウェア浮動小数点ユニットでは、レジスタ数の削減による節約は無視できるほど小さくなり、ABIsの拡散を減らしたいと考えています。

↑ABIって(application binary interface)だよね

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

-----------------------------------------------------------------------------------------------------------------------------------

私たちは、RV32Eサブセットを備えた完全なUnixスタイルのオペレーティングシステムをサポートするつもりはありません。

OSの可能性のあるコアのコンテキストでは、レジスタ数の削減による節約は無視できなくなり、私たちはOSの断片化を避けたいと考えています。

-- 2018/05/15

第4章

RV64Iベース整数命令セット、

バージョン2.0

この章では、RV64I基本整数命令セットについて説明します。これは、第2章で説明したRV32I変形を基にしています。

この章では、RV32Iとの相違点のみを示していますので、前の章と併せて読んでください。

4.1 レジスタ状態

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

4.2 整数計算命令

RV64Iの32ビット値を操作する追加の命令の変形が用意されています。これはオペコードの接尾辞「W」で示されます。

これらの「\* W」命令は、その入力の上位32ビットを無視し、常に32ビットの符号付き値を生成する。すなわち、ビットXLEN-1〜31が等しくなります。

！XLEN＝64なので、63:31が同じになる。符号拡張されているということ

RV32Iで不正な命令例外が発生します。

-----------------------------------------------------------------------------------------------------------------------------------

コンパイラと呼び出し規約では、すべての32ビット値が64ビットレジスタの符号拡張形式で保持されるという不変量を維持しています。

32ビットの符号なし整数でも、ビット31をビット63から32に拡張する。

その結果、符号付き32ビット整数から符号付き64ビット整数への変換と同様に、符号なし32ビット整数と符号付き32ビット整数の間の変換は、演算なしです。

既存の64ビット幅のSLTUおよび符号なし分岐比較は、この不変量の下で符号なし32ビット整数に対しても正しく動作します。

同様に、32ビット符号拡張整数の既存の64ビット幅論理演算は、符号拡張プロパティを保持します。

32ビット値の妥当な性能を保証するために、追加およびシフトにはいくつかの新しい命令（ADD [I] W / SUBW / SxxW）が必要です。

整数レジスタ - 即時命令

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | opcode |  |

12 5 3 5 7

I-即値[11:0] src ADDIW dest OP-IMM-32

ADDIWは、符号拡張された12ビットの即値をレジスタrs1に加算し、rdの32ビット結果の適切な符号拡張を生成するRV64Iのみの命令です。

オーバーフローは無視され、結果は64ビットに符号拡張された結果の下位32ビットになります。

注、ADDIW rd,rs1,0は、レジスタrs1の下位32ビットの符号拡張をレジスタrdに書き込みます（アセンブラ疑似演算SEXT.W）。

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| imm[11:6] | imm[5] | imm[4:0] | rs1 | funct3 | rd | opcode |  |

6 1 5 5 3 5 7

000000 shamt[5] shamt[4:0] src SLLI dest OP-IMM

000000 shamt[5] shamt[4:0] src SRLI dest OP-IMM

010000 shamt[5] shamt[4:0] src SRAI dest OP-IMM

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

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

010000 0 shamt[4:0] src SRAIW dest OP-IMM-32

定数によるシフトは、RV32Iと同じ命令オペコードを使用してI型形式の特殊化としてエンコードされます。

シフトされるオペランドはrs1であり、シフト量はRV64IのI即値フィールドの下位6ビットにエンコードされます。

右のシフトタイプはビット30で符号化されます。

SLLIは論理的な左シフトです（ゼロは下位ビットにシフトされます）。 SRLIは論理右シフトです（0は上位ビットにシフトされます）。 SRAIは算術右シフトです（元の符号ビットは空いている上位ビットにコピーされます）。

RV32Iの場合、imm [5] 6 = 0の場合、SLLI、SRLI、およびSRAIは不正な命令例外を生成します。

SLLIW、SRLIW、およびSRAIWは、同様に定義されていますが、32ビット値で動作し、符号付き32ビット結果を生成するRV64I専用の命令です。

SLLIW、SRLIW、およびSRAIWはimm [5] ≠ 0の場合、不正な命令例外を生成します。

31 12 11 7 6 0

|  |  |  |  |
| --- | --- | --- | --- |
| Imm[31:12] | rd | opcode |  |

20 5 7

U-即値[31:12] dest LUI

U-即値[31:12] dest AUIPC

LUI（ロードアッパー即時）は、RV32Iと同じオペコードを使用します。

LUIは、20ビットのU-即値をレジスタrdのビット31-12に配置し、最下位の12ビットにゼロを配置します。

32ビットの結果は64ビットに符号拡張されます。

AUIPC（PCに即値の上位を追加）は、RV32Iと同じオペコードを使用します。

AUIPC（PCに即値の上位を追加）は、PC相対アドレスを作成するために使用され、Uタイプ形式を使用します。

AUIPCは、20ビットのU－即値に12の下位ゼロビットを付加し、その結果を64ビットに符号拡張し、それをpcに加え、結果をレジスタrdに格納します。

整数レジスタ - レジスタ演算

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct7 | rs2 | rs1 | funct3 | rd | opcode |  |

7 5 5 3 5 7

0000000 src2 src1 SLL/SRL dest OP

0100000 src2 src1 SRA dest OP

0000000 src2 src1 ADDW dest OP-32

0000000 src2 src1 SLLW/SRLW dest OP-32

0100000 src2 src1 SUBW/SRAW dest OP-32

ADDWおよびSUBWは、ADDおよびSUBと同様に定義されますが、32ビット値で動作し、符号付き32ビット結果を生成するRV64I専用の命令です。

オーバーフローは無視され、結果の下位32ビットは64ビットに符号拡張され、デスティネーション・レジスタに書き込まれます。

SLLW、SRLW、およびSRAWは、同様に定義されていますが、32ビット値で動作し、符号付き32ビット結果を生成するRV64I専用の命令です。

シフト量はrs2 [4：0]で与えられます。

4.3 ロードとストア命令

RV64Iはアドレス空間を64ビットに拡張します。

実行環境は、アドレス空間のどの部分にアクセスするのが正当であるかを定義します。

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | opcode |  |

12 5 3 5 7

offset[11:0] base width dest LOAD

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode |  |

7 5 5 3 5 7

offset[11:5] src base width offset[4:0] STORE

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

LW命令はメモリから32ビット値をロードし、これを64ビットに符号拡張してからRV64Iのレジスタrdに格納します。

一方、LWU命令は、RV64Iのメモリから32ビット値をゼロ拡張します。

LHおよびLHUは、8ビット値のLBおよびLBUと同様に、16ビット値に対しても同様に定義されます。

SD、SW、SH、およびSB命令は、レジスタrs2の下位ビットから64ビット、32ビット、16ビット、および8ビットの値をそれぞれメモリに格納します。

4.4 システム命令

RV64Iでは、CSR命令が64ビットCSRsを操作できます。

特に、RDCYCLE、RDTIME、およびRDINSTRETの疑似命令は、サイクル、時間、および開始カウンタの全64ビットを読み取ります。

したがって、RDCYCLEH、RDTIMEH、RDINSTRETH命令は不要であり、RV64Iでは不正です。

第5章

RV128I基本整数命令セット、

バージョン1.7

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

ベル と ストレッカー、ISCA-3、1976。

この章では、128ビットアドレス空間をサポートするRISC-V ISAの変形であるRV128Iについて説明します。

この変形は、既存のRV32IおよびRV64I設計の直接的な外挿(推定)です。

-----------------------------------------------------------------------------------------------------------------------------------

整数レジスタの幅を拡張する主な理由は、より大きなアドレス空間をサポートするためです。

64ビット以上の平なアドレス空間が必要な場合は明確ではありません。

執筆時点では、Top500ベンチマークで測定された世界最速のスーパーコンピュータは1PB以上のDRAMを搭載しており、すべてのDRAMが単一のアドレス空間に存在する場合は50ビット以上のアドレス空間が必要になります。

いくつかの倉庫規模のコンピュータの中には、既に大量のDRAMが搭載されており、新しい高密度個体不揮発性メモリや高速相互接続技術によって、さらに大きなメモリ空間の需要を牽引が求められる場合があります。

エクサスケール システム研究は、57ビットのアドレス空間を占める100PBのメモリシステムを対象としています。

過去の成長率では、2030年までに64ビット以上のアドレス空間が必要になる可能性があります。

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

現時点では128ビットアドレス空間の実際の使用に基づいて設計を進化させる必要があるため、RV128仕様を凍結していません。

↑ まだ変わることもあるかもよってこと

RV128Iは、整数レジスタを128ビットに拡張し、RV32Iの上にRV64Iを構築するのと同様に、RV64Iを構築したものです（すなわち、XLEN = 128）。

ほとんどの整数計算命令は、XLENビットで動作するように定義されているので変更されません。

レジスタの下位ビットの32ビット値で動作するRV64I "\* W"整数命令が保持され、そして、128ビット整数レジスタの下位ビットに保持されている64ビット値で動作する新しい「\* D」整数命令セットが追加されます。

「\* D」命令は、標準の32ビットエンコーディングで2つの主要なオペコード（OP-IMM-64およびOP-64）を消費します。

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

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

SQはSTOREメジャーオペコードに追加され、LQはMISC-MEMメジャーオペコードに追加されます。

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

--2018/05/19

第6章

整数乗算および除算のための "M"標準拡張、バージョン2.0

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

-----------------------------------------------------------------------------------------------------------------------------------

ローエンドの実装を単純化するために、または整数の乗除算演算が接続されたアクセラレータではあまり頻繁ではない、またはより適切に処理されるアプリケーションのために、整数の乗算と除算をベースから分離します。

↑基本命令セットには乗除算は組み込まずに別にした。そりゃ乗除算は面積食うからね。

6.1乗算演算

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct7 | rs2 | rs1 | funct3 | rd | opcode |  |

7 5 5 3 5 7

MULDIV multiplier multiplicand MUL/MULH[[SU] dest OP

MULDIV multiplier multiplicand MULW dest OP-32

MULはXLENビット\* XLENビットの乗算を行い、下位のXLENビットを送り先レジスタに配置します。

MULH、MULHU、およびMULHSUは同じ乗算を行いますが、符号付き\*符号付き、符号なし\*符号付き、および符号付き\*符号なし乗算の場合、フル2 \* XLENビット積の上位XLENビットを返します。

同じ製品の上位ビットと下位ビットの両方が必要な場合、推奨コードシーケンスは次のとおりです。：MULH [[S] U] rdh, rs1, rs2; MUL rdl, rs1, rs2（送り元レジスタ指定子は同じ順序でなければならず、rdhはrs1またはrs2と同じにすることはできません）。

マイクロアーキテクチャは、2つの別々の乗算を実行する代わりに、これらを1つの乗算演算に融合することができます。

MULWはRV64でのみ有効で、送り元レジスタの下位32ビットを乗算し、結果の下位32ビットの符号拡張を送り先レジスタに配置します。

MULを使用して64ビット積の上位32ビットを得ることができますが、符号付き引数は適切な32ビット符号付き値でなければなりませんし、符号なし引数は上位32ビットをクリアする必要があります。

6.2 除算操作

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct7 | rs2 | rs1 | funct3 | rd | opcode |  |

7 5 5 3 5 7

MULDIV divisor dividend DIV[U]/REM[U] dest OP

MULDIV divisor dividend DIV[U]W/REM[U]W dest OP-32

DIVとDIVUはXLENビットでXLENビットの符号付きおよび符号なし整数除算を実行します。

REMとREMUは、対応する除算演算の残りの部分を提供します。

商と剰余の両方が同じ除算から必要とされる場合、推奨されるコードシーケンスは次のとおりです。DIV [U] rdq, rs1,rs2; REM [U] rdr, rs1, rs2（rdqはrs1またはrs2と同じにすることはできません）。

マイクロアーキテクチャは、2つの別個の除算を実行する代わりに、これらを1つの除算演算に融合することができます。

ゼロ除算と除算オーバーフローの意味を表6.1に要約します。

0による除算の商は、すべてのビットがセットされています、すなわち、符号なし除算の場合は2XLEN-1、符号付き除算の場合は-1です。

ゼロによる除算の残りは被除数と等しいです。

符号付き除算のオーバーフローは、負の整数-2XLEN-1を-1で除算した場合にのみ発生します。

符号付き除算オーバーフローの商は被除数に等しく、余りはゼロです。

符号なし除算のオーバーフローは発生しません。

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

表6.1：ゼロ除算および除算オーバーフローのセマンティクス

-----------------------------------------------------------------------------------------------------------------------------------

私たちは、整数ゼロ除算の例外を発生させることを考慮しました。これらの例外は、ほとんどの実行環境でトラップを引き起こします。

しかし、これは標準ISA（浮動小数点例外フラグをセットし、デフォルト値を書き込むが、トラップは発生しません）の唯一の算術トラップであり、この場合、実行環境のトラップハンドラと対話する言語実装が必要となります。

さらに、ゼロ除算例外が即座の制御フロー変更を引き起こさなければならない言語規格(標準)では、各分岐演算に1つの分岐命令のみを追加する必要があり、この分岐命令は分割後に挿入することができ、 非常に予想通りに実行されず、ランタイムオーバーヘッドを少し追加します。

↑この文章はセンテンスが長いので係り受けがいまいちあやしいかも

除算回路を簡略化するために、符号なしと符号付きの両方の除算値をゼロで除算すると設定されたすべてのビットの値が返されます。

すべての1sの値は、最大の符号なし数値を表す符号なし除算に返される自然数と、単純な符号なし除算器実装の自然な結果の両方です。

符号付き除算は符号なし除算回路を使用して実装されることが多く、同じオーバーフロー結果を指定するとハードウェアが単純化されます。

DIVW命令とDIVUW命令はRV64でのみ有効で、rs1の下位32ビットをrs2の下位32ビットで除算し、それぞれ符号付き整数と符号なし整数として扱い、32ビット商をrdに配置し、64ビットに符号拡張します 。

REMW命令とREMUW命令はRV64でのみ有効で、対応する符号付きおよび符号なしの剰余演算をそれぞれ提供します。

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

--2018/05/19

-- 1ページ空けて次ページへ

第7章

アトミック命令のための "A" 標準拡張、バージョン2.0

！アトミックって何？ がここで明かされるのか？

-----------------------------------------------------------------------------------------------------------------------------------

このセクションは、現在のプログラミング言語のメモリモデルを効率的にサポートできるように、RISC-Vメモリモデルが現在改訂中であるため、多少古くなっています。

修正された基本メモリモデルには、少なくとも同じハートからの同じアドレスへのロードを並べ替えることができず、および命令間の構文データ依存性が尊重されることを含む、さらなる順序制約が含まれます。

標準的なアトミック命令拡張は、命令サブセット名 ”A” によって示され、同じメモリ空間で動作する複数のRISC-Vハート間の同期をサポートするために原子的に読み取り－変更－書き込み メモリの命令を含んでいます。

提供される2つの形式のアトミック命令は、ロードー予約/ストアー条件付き命令およびアトミック フェッチーアンドーオペレーション・メモリ命令です。

どちらの種類のアトミック命令は、順序付けなし、取得、解放、および逐次的に一貫性のあるセマンティクスを含む、様々なメモリ整合性順序をサポートします。

これらの命令は、RISC-VがRCscメモリ一貫性モデル[10]をサポートすることを可能にすることができます。

-----------------------------------------------------------------------------------------------------------------------------------

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

7.1アトミック命令の順序付けの指定

基本RISC-V ISAには緩やかなメモリモデルがあり、FENCE命令は追加の順序制約を課すために使用されます。

アドレス空間は、実行環境によってメモリ領域とI / O領域に分割され、FENCE命令は、これら2つのアドレス領域の一方または両方へのアクセスを順序付けるためのオプションを提供します。

リリース一貫性[10]のより効率的なサポートを提供するために、各アトミック命令は2つのビットaqとrlを持ち、他のRISC-Vハートで見られる追加のメモリ順序制約を指定します。

ビット順は、原子命令がどのアドレス・ドメインにアクセスしているかに応じて、2つのアドレス・ドメイン、メモリまたはI / Oのうちの1つにアクセスします。

順序付け制約は他のドメインへのアクセスには暗示されず、FENCE命令を使用して両方のドメイン間で順序付けする必要があります。

両方のビットがクリアされている場合、アトミックメモリ動作に追加の順序制約条件が課せられません。

aqビットのみがセットされている場合、アトミック・メモリ操作は獲得アクセスとして扱われ、

すなわち、このRISC-Vハートに対する後続のメモリ操作は、メモリ取得動作の前に行われることは観察できません。

rlビットだけがセットされている場合、アトミックメモリ操作は解放アクセスとして扱われ、

すなわち、解放メモリ操作は、このRISC-Vハート上の以前のメモリ操作の前に行われることは観察できません。

aqビットとrlビットの両方がセットされている場合、アトミックメモリ動作は逐次一致し、同じRISC-Vハート内で、以前のメモリ動作の前またはその後のメモリ動作の後に発生することは観測できず、 同一のアドレス領域に対するすべての逐次的に一貫した原子メモリ操作の同じグローバル順序で他のハートによって観察されることができる。

-----------------------------------------------------------------------------------------------------------------------------------

理論的には、aqビットとrlビットの定義は、グローバルストアのアトミック性を持たない実装を可能にします。

しかし、aqビットとrlビットの両方がセットされている場合、アトミック操作のための完全な逐次一貫性が必要です。これは、取得と解放の両方のセマンティクスに加えてグローバルストアの原子性を意味します。

実際には、ハードウェアシステムは、通常、シングルライタのキャッシュコヒーレンスプロトコルとともにローカルプロセッサの順序付け規則に組み込まれたグローバルストアのアトミック性で実装されています。

7.2 ロード予約/ストア条件付き命令

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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | aq | rl | rs2 | rs1 | funct3 | rd | opcode |  |

5 1 1 5 5 3 5 7

LR ordering 0 addr width dest AMO

SC ordering src addr width dest AMO

単一のメモリワードに対する複雑なアトミックメモリ操作は、ロード予約（LR）およびストア条件付き（SC）命令で実行されます。

LRはrs1のアドレスからワードをロードし、符号拡張された値をrdに配置し、メモリアドレスに予約を登録します。

SCは、そのアドレスに有効な予約がまだ存在する場合、rs1のアドレスにrs2のワードを書き込みます。

SCは、成功するとrdに0を、失敗した場合は0以外のコードを書き込みます。

-----------------------------------------------------------------------------------------------------------------------------------

比較とスワップ（CAS）とLR / SCの両方は、ロック-フリーのデータ構造を構築するのに使用できます。

広範な議論の後、私たちはいくつかの理由からLR / SCを選択しました：

1）CASは、データ値の変化をチェックするのではなく、アドレスへのすべてのアクセスを監視するため、LR / SCが回避するABAの問題を抱えています。

2）CASはまた、3つのソースオペランド（アドレス、比較値、スワップ値）、および異なるメモリシステムメッセージフォーマットをサポートするために新しい整数命令フォーマットを必要とし、これはマイクロアーキテクチャを複雑にします。

3）さらに、ABA問題を回避するために、他のシステムでは、カウンタをデータワードとともにテストしインクリメントできるように、ダブル幅のCAS（DW-CAS）を提供しています。

これには、5つのレジスタを読み込んで1つの命令に2つの命令を書き込む必要があります。かつ、新しい大規模なメモリシステムのメッセージタイプであり、実装をさらに複雑にします。

4）LR / SCは、CASを使用する2つのロードではなく、1つのロードしか必要としないため、多くのプリミティブのより効率的な実装を提供します。

（投機的計算のための値を得るためにCAS命令の前に1回ロードし、次にCAS命令の一部として2回目のロードを行い、値が更新前に変更されていないかどうかをチェックします）

CASに対するLR / SCの主な欠点はライブロックであり、これは、以下で説明するように、最終的なフォワード進捗のアーキテクチャ保証を避けています。

もう一つの懸念は、現在のx86アーキテクチャがDW-CASに与える影響が、DW-CASが基本的なマシンプリミティブであることを前提とした同期ライブラリやその他のソフトウェアの移植を複雑にするかどうかです。

考えられる緩和要因は、最近のx86へのトランザクショナルメモリ命令の追加であり、DW-CASからの移行を引き起こす可能性があります。

値1の障害コードは、不特定の障害をエンコードするために予約されています。

現時点では他の障害コードが予約されており、可搬ソフトウェアは障害コードがゼロではないと仮定する必要があります。

LRおよびSCは、メモリ内の64ビット（RV64のみ）または32ビットの自然なアラインメントで動作します。

ミスアラインアドレスは、ミスアラインアドレス例外を生成します。

-----------------------------------------------------------------------------------------------------------------------------------

SLT / SLTU命令に必要な既存のマルチプレクサを使用して簡単な実装がこの値を返すことができるように、”未指定” を意味する1の不合格コードを予約します。

より具体的な障害コードは、ISAの将来のバージョンまたは拡張で定義される可能性があります。

標準A拡張では、特定の制約付きLR / SCシーケンスが最終的に成功することが保証されています。

LR / SCシーケンスの静的コードと、障害の発生時にシーケンスを再試行するコードには、最大16個の整数命令を順次メモリに配置する必要があります。

シーケンスが最終的に成功することが保証されるために、LR命令とSC命令との間で実行される動的コードは、ロード、ストア、逆方向ジャンプまたは取り去られた逆方向分岐、FENCE、FENCEI、およびSYSTEM命令を除いて、ベース ”I” サブセットからの他の命令のみを含むことができる。

失敗したLR / SCシーケンスを再試行するコードは、後方ジャンプおよび/または分岐を含み、LR / SCシーケンスを繰り返すことができるが、そうでなければ同じ制約を有します。

SCは、実行された最新のLRと同じアドレスになければなりません。

これらの制約を満たさないLR / SCシーケンスは、いくつかの実装でいくつかの試みを完了するかもしれませんが、最終的な成功を保証するものではありません。

-----------------------------------------------------------------------------------------------------------------------------------

CASの利点の1つは、LR / SC原子配列が一部のシステムで無期限にライブロックできるのに対し、最終的にいくつかのハートが進行することを保証することです。

この懸念を避けるために、我々は、LR / SC原子配列に前方進行の構造的保証を追加しました。

LR / SCシーケンス内容の制限により、実装では、LR上のキャッシュラインをキャプチャし、短時間のリモートキャッシュ介入を保留することによってLR / SCシーケンスを完了することができます。

割込みおよびTLBミスにより、予約が失われる可能性がありますが、最終的にはアトミックシーケンスが完了することがあります。

命令キャッシュとTLBのサイズと結合性に関する過度の制限を避けるため、LR / SCシーケンスの長さを基本ISAの64個の連続した命令バイトに収めるように制限しました。

同様に、シーケンス内に他のロードやストアを許可しないことで、データキャッシュの結合性の制限が回避されました。

分岐とジャンプの制限により、シーケンスで費やせる時間が制限されます。

浮動小数点演算と整数乗算/除算は、適切なハードウェアサポートがない実装でこれらの命令のオペレーティングシステムのエミュレーションを単純化するために許可されませんでした。

実装は、各LR上のメモリ空間の任意のサブセットを予約することができ、複数のLR予約は、単一のハートのために同時にアクティブになることができます。

アドレスを予約するために、このハートのSCと最後のLRとの間に他のハートからアドレスへのアクセスが発生していないことが確認できれば、SCは成功することができます。

注：このLRはアドレス引数が異なるかもしれませんが、メモリサブセットの一部としてSCのアドレスを予約していることに注意してください。

このモデルに続いて、メモリ変換を伴うシステムでは、以前のLRが別の仮想アドレスを持つエイリアスを使用して同じロケーションを予約していた場合、SCは成功することができます。しかし、仮想アドレスが異なる場合には失敗することが許されます。

他のハートからアドレスへのメモリアクセスが観察可能である場合、またはこのハートに介在するコンテキストスイッチがある場合、またはその間にハートが特権例外復帰命令を実行した場合、SCは失敗する必要があります。

-----------------------------------------------------------------------------------------------------------------------------------

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

LR / SCは、ロックフリーのデータ構造を構築するために使用できます。

比較およびスワップ機能を実装するためのLR / SCの使用例を図7.1に示します。

インライン化されている場合、比較とスワップの機能は3つの命令しか必要としません。

＃ a0 メモリロケーションのアドレスを保持

＃ a1 期待値を保持

＃ a2 望ましい値を保持

＃ a0 戻り値、成功した場合は0、そうでない場合は !0 を保持

cas:

lr.w t0, (a0) ＃元の値をロードします。

bne t0, a1, fail ＃が一致しないので失敗します。

sc.w a0, a2, (a0) ＃更新を試みます。

jr ra ＃リターン。

fail:

li a0, 1 ＃リターンを失敗に設定します。

jr ra ＃リターン。

図7.1：LR / SCを使用した比較およびスワップ機能のサンプルコード

-- 2018/05/25

SC命令は、直前のLRの前に別のRISC-Vハートによって観察されることは決してできません。

LR / SCシーケンスの原子的性質のために、LRと成功したSCとの間には何らかのハートからのメモリ操作は発生していないことが確認できます。

LR / SCシーケンスは、SC命令のaqビットをセットすることによって、セマンティクスを獲得することができます。

LR / SCシーケンスは、LR命令のrlビットをセットすることにより、解除セマンティクスを与えることができます。

LR命令のaqビットとrlビットの両方をセットし、SC命令のaqビットをセットすることにより、LR / SCシーケンスは、他の逐次的に一貫したアトミック操作に関して順次整合します。

どちらのビットもLRとSCの両方に設定されていない場合、同じRISC-Vハートから周囲のメモリ操作の前後にLR / SCシーケンスが発生することが観察されます。

これは、LR / SCシーケンスを使用して並列リダクション操作を実装する場合に適しています。

どちらのビットもLRとSCの両方に設定されていない場合、同じRISC-Vハートから周囲のメモリ操作の前後にLR / SCシーケンスが発生することが観察されます。

これは、LR / SCシーケンスを使用して並列リダクション操作を実装する場合に適しています。

-----------------------------------------------------------------------------------------------------------------------------------

一般に、マルチワードの原子プリミティブが望ましいが、これがどのような形で取るべきかについてはまだ議論があり、進歩を保証することはシステムに複雑さを加えます。

我々の現在の考えは、任意の標準的な拡張子 ”T” として、元のトランザクショナルメモリ提案のラインに沿った限られた容量の小さなトランザクションメモリバッファを含むことです。

7.3 原子メモリ操作

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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | aq | rl | rs2 | rs1 | funct3 | rd | opcode |  |

5 1 1 5 5 3 5 7

AMOSWAP.W/D ordering src addr width dest AMO

AMOADD.W/D ordering src addr width dest AMO

AMOAND.W/D ordering src addr width dest AMO

AMOOR.W/D ordering src addr width dest AMO

AMOXOR.W/D ordering src addr width dest AMO

AMOMAX[U].W/D ordering src addr width dest AMO

AMOMIN[U].W/D ordering src addr width dest AMO

アトミックメモリ操作（AMO）命令は、マルチプロセッサ同期のための読み出し - 変更 - 書き込み動作を実行し、Rタイプの命令フ形式で符号化されます。

これらのAMO命令は、rs1のアドレスからデータ値を原子にロードし、その値をレジスタrdに配置し、ロードされた値とrs2の元の値に二項演算子を適用し、結果をrs1のアドレスに戻します。

AMOsは、メモリ内の64ビット（RV64のみ）または32ビットワードで動作することができます。

RV64では、32ビットのAMOsは常にrdに配置された値を符号拡張します。

rs1に保持されるアドレスは、オペランドのサイズに自然に整列されなければなりません（すなわち、64ビットワードの場合は8バイト整列、32ビットワードの場合は4バイト整列）。

アドレスが自然に整列されていない場合、不整合アドレス例外が生成されます。

サポートされている操作は、スワップ、整数加算、論理AND、論理OR、論理XOR、符号付きおよび符号なし整数の最大値と最小値です。

順序制約がない場合、これらのAMOsを使用して通常、戻り値はx0に書き込むことによって破棄される、並列リダクション演算を実装することができます。

-----------------------------------------------------------------------------------------------------------------------------------

我々は、LR / SCまたはCASよりも高度に並列なシステムに拡張するので、フェッチ・アンド・オペレーション・スタイルの原子プリミティブを提供しました。

単純なマイクロアーキテクチャでは、LR / SCプリミティブを使用してAMOsを実装できます。

さらに複雑な実装では、メモリコントローラにAMOsを実装し、宛先がx0のときに元の値を取り出すことを最適化できます。

AMOsのセットは、C11 / C ++ 11アトミックメモリ操作を効率的にサポートし、また、メモリの並列削減をサポートするために選択されました。

AMOsの別の使用法は、I / O空間内のメモリマップされたデバイスレジスタ（e..g、設定、クリア、またはトグルビット）にアトミック更新を提供することです。

マルチプロセッサ同期の実装を支援するために、AMOはオプションでリリース一貫性セマンティクスを提供します。

aqビットがセットされていれば、このRISC-Vハートの後のメモリ操作は、AMOの前に行われることはありません。

逆に、rlビットが設定されている場合、他のRISC-Vハーツは、このRISC-VハートのAMOに先行するメモリアクセスの前にAMOを観察しません。

-----------------------------------------------------------------------------------------------------------------------------------

AMOsは、C11およびC ++ 11メモリモデルを効率的に実装するように設計されました。

FENCE R、RW命令は取得操作とフェンスRWを実装するのに、Wはリリースを実装するのに十分ですが、対応するaqまたはrlビットが設定されたAMOsと比較して不必要な順序付けが必要です。

テスト-アンド-セット スピンロックによって保護されるクリティカルセクションのコードシーケンスの例を図7.2に示します。

注：最初のAMOはaqとマークされ、クリティカルセクションの前にロック取得を注文し、2番目のAMOはrlとマークされて、ロックの解放前にクリティカルセクションを注文することに注意してください。

li t0, 1 # Initialize swap value.

again:

amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.

bnez t0, again # Retry if held.

# ...

# Critical section.

# ...

amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.

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

-----------------------------------------------------------------------------------------------------------------------------------

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

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

この効果は、テスト-アンド-セット ロックと似ていますが、コード・パスは短くなります。

“A” 拡張の命令は、一貫性のあるロードおよびストアを連続して提供するためにも使用できます。

連続的に一貫した負荷は、aqとrlの両方を設定したLRとして実装できます。

連続的に一貫したストアは、古い値をx0に書き込み、aqとrlの両方を設定するAMOSWAPとして実装できます。

！ちょっと調べてみたけど 原子的、アトミック って (分割不可分) ということらしい

！ここの章では主にメモリについてのアクセスが記してあったように思える。

！あるハーツ(アクセス)での転送を細切れにせず、そのアクセスの間ははバスを確保するようなことなのかな

！sequentiallyとかとかあるしね

！で、ウィキペディアに不可分操作ってあって、アトミック操作とも書いてある

！[https://ja.wikipedia.org/wiki/%E4%B8%8D%E5%8F%AF%E5%88%86%E6%93%8D%E4%BD%9C](https://ja.wikipedia.org/wiki/不可分操作)

！以下、ウィキペディアより引用

！不可分操作は、以下の2つの条件を満たさなければならない。

！１．全操作が完了するまで、他のプロセスはその途中の状態を観測できない。

！２．一部操作が失敗したら組合せ全体が失敗し、システムの状態は不可分操作を行う前の状態に戻る。

！システムの他の部分から見て、操作の組合せが一度に成功したか失敗したように見える。

！途中の状態にアクセスすることはできない。このため不可分操作あるいはアトミック操作（= 原子操作）と呼ぶのである。

！原子ってこれ以上分割できないっていう意味で使ってるんですな。

！例えばAとBの2つのハーツ(タスクとかスレッド)が走ってて、Aが0～99番地まで書き込みを行っていて20番地まで

！書いたときに、Bが割り込んで50～59番地まで書いて、またAに戻って21～99番地まで書いたとすると、Bが書いた

！50～59は書きつぶされてしまう。

！

！この場合はAの書き込みが終わるまでロックかけてバスを占有しないといけない。

！こういう転送を原子(アトミック)転送 ってことで良いかな。

！

第8章

単精度浮動小数点の "F"標準拡張、

バージョン2.0

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

8.1 Fレジスタの状態

F拡張は、浮動小数点ユニットの動作モードと例外ステータスを含む32個のそれぞれ32ビット幅の浮動小数点レジスタf0〜f31、と浮動小数点制御およびステータスレジスタfcsrを追加します。

この追加の状態を図8.1に示します。

RISC-V ISAでは浮動小数点レジスタの幅を記述するためにFLENという用語を使用し、F単精度浮動小数点の拡張ではFLEN = 32を使用します。

ほとんどの浮動小数点命令は浮動小数点レジスタファイルを操作します。

浮動小数点ロード命令とストア命令は、浮動小数点値をレジスタとメモリ間で転送します。

また、整数レジスタファイルとの間で値を転送するための命令も提供されます。

-----------------------------------------------------------------------------------------------------------------------------------

整数と浮動小数点値の統一されたレジスタファイルを考えました。これは、ソフトウェアレジスタの割り当てと呼び出し規約を単純化し、ユーザーの全状態を減らすためです。

ただし、分割組織では、指定された命令幅でアクセス可能なレジスタの総数が増加し、広いスーパスカラ問題に対応する十分なregfileポートの準備が簡素化され、分離浮動小数点ユニットアーキテクチャがサポートされ、内部浮動小数点エンコーディングテクニックの使用が簡素化されます。

分割レジスタファイルアーキテクチャのコンパイラサポートと呼び出し規約は十分に理解されており、浮動小数点レジスタファイル状態でダーティビットを使用すると、コンテキストスイッチのオーバーヘッドを減らすことができます。

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8.1：RISC-V標準F拡張単精度浮動小数点状態

31 8 7 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| 予約 | 丸め方式 (frm) | 発生した例外 (fflags) |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NV | DZ | OF | UF | NX |

24 3 １ 1 1 1 1

図8.2：浮動小数点制御およびステータスレジスタ。

8.2浮動小数点制御および状態レジスタ

浮動小数点制御および状態レジスタfcsrは、RISC-V制御および状態レジスタ（CSR）です。

図8.2に示すように、浮動小数点算術演算の動的丸めモードを選択し、発生した例外フラグを保持する32ビットの読取り/書込みレジスタです。

fcsrレジスタは、基になるCSRアクセス命令で構築されたアセンブラ疑似操作であるFRCSR命令とFSCSR命令で読み書きできます。

FRCSRはfcsrを整数レジスタrdにコピーすることによってそれを読み取ります。

FSCSRは、元の値を整数レジスタrdにコピーし、次に整数レジスタrs1から取得した新しい値をfcsrに書き込むことによって、fcsrの値を入れ替えます。

fcsr内のフィールドは、異なるCSRアドレスを介して個別にアクセスすることもでき、これらのアクセスに対して別々のアセンブラ擬似操作が定義されます。

FRRM命令は、丸めモードフィールドfrmを読み取り、それを整数レジスタrdの最下位3ビットにコピーします。他のすべてのビットはゼロです。

FSRMは、元の値を整数レジスタrdにコピーし、整数レジスタrs1の最下位3ビットから得られた新しい値をfrmに書き込むことによって、frmの値を交換します。

FRFLAGSとFSFLAGSは、未払い例外フラグフィールドfflagsと同様に定義されています。

↑Accrued って Google翻訳とかだと 未払いとか未収ってなるけど、Webiloで辞書ひくと accure 生じる、発生する となる。

↑発生した例外フィールド くらいでいいかな

追加の疑似命令FSRMIとFSFLAGSIは、レジスタrs1の代わりに即値を使用して値を交換します。

fcsrのビット31-8は、10進浮動小数点の "L"標準拡張を含む他の標準拡張のために予約されています。

これらの拡張が存在しない場合、実装はこれらのビットへの書き込みを無視し、読み込み時にゼロの値を供給しなければならない。

標準ソフトウェアはこれらのビットの内容を保持する必要があります。

浮動小数点演算は、命令でエンコードされた静的な丸めモードまたはfrmに保持された動的な丸めモードのいづれかを使用します。

丸めモードは、表8.1に示すようにエンコードされます。

命令のrmフィールドの値が111の場合、frmに保持されている動的丸めモードが選択されます。

frmが無効な値（101-111）に設定されている場合、その後の動的丸めモードで浮動小数点演算を実行しようとすると、不正な命令トラップが発生します。

rmフィールドを持ついくつかの命令は、丸めモードの影響をそれにも関わらず受けません(受けないのもあります)。 rmフィールドをRNE（000）に設定する必要があります。

-----------------------------------------------------------------------------------------------------------------------------------

C99言語規格は、動的丸めモードレジスタの提供を効果的に義務づけています。

--2018/05/26

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

表8.1：丸めモードのエンコーディング

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

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

表8.2：発生した例外フラグの符号化。

-----------------------------------------------------------------------------------------------------------------------------------

標準で許可されているように、我々はベースISAの浮動小数点例外のトラップはサポートしていませんが、ソフトウェアのフラグを明示的にチェックする必要があります。

浮動小数点が発生した例外フラグの内容によって直接制御される分岐を追加することを検討しましたが、最終的にISAを単純に保つためにこれらの命令を省略することを選択しました。

8.3 NaNの生成と伝播

↑NaN って 確か不定状態のことだったよな

特に明記されている場合を除いて、浮動小数点演算の結果がNaNの場合、正規NaNになります。

正規のNaNは正の符号を持ち、MSB以外のすべての符号ビットをクリアします。つまり、静かなビットです。

単精度浮動小数点の場合、これはパターン0x7fc00000に対応します。

FMINとFMAXの場合、少なくとも1つの入力がシグナリングNaNである場合、または両方の入力が静かなNaNの場合、結果は正規のNaNになります。

1つのオペランドが静かなNaNであり、もう1つがNaNでない場合、結果は非NaNオペランドになります。

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

-----------------------------------------------------------------------------------------------------------------------------------

標準で推奨されているように、私たちはNaNペイロードを伝播すると考えましたが、この決定はハードウェアコストを増加させるでしょう。

さらに、この機能は標準ではオプションなので、移植可能なコードでは使用できません。

実装者は、非標準動作モードで有効にされた非標準拡張としてNaNペイロード伝播スキームを自由に提供できます。

ただし、上記の正規のNaNスキームは常にサポートされていなければならず、既定のモードでなければなりません。

-----------------------------------------------------------------------------------------------------------------------------------

私たちは例外レベルの場合には、標準レベルのデフォルト値を返す実装が必要です。これは、ユーザレベルのソフトウェアの部分に何ら介入しなくても可能です。（アルファ ISA浮動小数点トラップバリアとは異なります）。

私たちは例外的なケースの完全なハードウェア処理がより一般的になると考えており、他のアプローチを最適化するためにユーザーレベルのISAを複雑にしないようにしたいと考えています。

実装は、例外的なデフォルト値を提供するために、常にマシンモードのソフトウェアハンドラにトラップすることができます。

8.4非正常な算術

非正規数の演算は、IEEE 754-2008規格に従って処理されます。

IEEE標準の言い回しでは、丸めた後に微小が検出されます。

↑ tininess って とても小さいという意味みたいだが、どう訳すのがいいかな

-----------------------------------------------------------------------------------------------------------------------------------

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

8.5 単精度ロード命令とストア命令

浮動小数点ロードとストアは、整数ベースISAと同じベース+オフセットアドレッシングモードを使用し、レジスタrs1のベースアドレスと12ビット符号付きバイトオフセットを使用します。

FLW命令は、メモリから浮動小数点レジスタrdに単精度浮動小数点値をロードします。

FSWは、浮動小数点レジスタrs2から単精度値をメモリに格納します。

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | width | rd | opcode |  |

12 5 3 5 7

offset[11:0] base W dest LOAD-FP

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| imm[11:5] | rs2 | rs1 | width | imm[4:0] | opcode |  |

7 5 5 3 5 7

offset[11:5] src base W offset[4:0] STORE-FP

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

8.6 単精度浮動小数点計算命令

1つまたは2つのソースオペランドを使用する浮動小数点算術命令は、OP-FPの主要なオペコードでR型形式を使用します。

FADD.S、FSUB.S、FMUL.S、およびFDIV.Sは、それぞれrs1とrs2の単精度浮動小数点加算、減算、乗算、および除算を実行し、結果をrdに書き込みます。

FMIN.SおよびFMAX.Sは、それぞれrs1およびrs2のうちの小さい方または大きい方をそれぞれrdに書き込みます。

FSQRT.Sはrs1の平方根を計算し、結果をrdに書き込みます。

2ビット浮動小数点フォーマットフィールドfmtは、表8.3に示すようにエンコードされます。

F拡張のすべての命令についてS（00）に設定されます。

2ビット浮動小数点フォーマットフィールドfmtは、表8.3に示すようにエンコードされます。

F拡張のすべての命令についてS（00）に設定されます。

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

表8.3：フォーマットフィールドのエンコード

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

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

FADD/FSUB S src2 src1 RM dest OP-FP

FMUL/FDIV S src2 src1 RM dest OP-FP

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

FSQRT S 0 src RM dest OP-FP

浮動小数点の融合乗加算命令は、新しい標準命令フォーマットを必要とします。

↑融合乗加算って 乗算と加算が一緒になったっていう意味だよね。つまり積和演算のことね。昔はDSPならではの命令だった

R4型命令は、3つのソースレジスタ（rs1、rs2、rs3）とデスティネーションレジスタ（rd）を指定します。

この形式は、浮動小数点の融合乗加算命令でのみ使用されます。

融合乗加算命令は、rs1とrs2の値を乗算し、オプションで積を否定し、rs3の値を加算または減算して最終結果をrdに書き込みます。

FMADD.Sは、rs1×rs2+rs3を計算します。 FMSUB.Sは、rs1×rs2-rs3を計算します。 FNMSUB.Sは、-rs1×rs2+rs3を計算します。 FNMADD.Sは-rs1×rs2-rs3を計算します。

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

-----------------------------------------------------------------------------------------------------------------------------------

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

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| rs3 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

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

8.7単精度浮動小数点変換および移動命令

浮動小数点から整数への変換命令および整数から浮動小数点への変換命令は、OP-FPメジャーオペコード空間で符号化されています。

FCVT.W.SまたはFCVT.L.Sは、浮動小数点レジスタrs1の浮動小数点数を、整数レジスタrdの符号付き32ビットまたは64ビット整数にそれぞれ変換します。

FCVT.S.WまたはFCVT.S.Lは、整数レジスタrs1の32ビットまたは64ビットの符号付き整数をそれぞれ浮動小数点レジスタrdの浮動小数点数に変換します。

FCVT.WU.S、FCVT.LU.S、FCVT.S.WU、およびFCVT.S.LUバリアントは、符号なし整数値に変換されます。

FCVT.L [U] .SとFCVT.S.L [U]はRV32では不正です。

丸められた結果が宛先フォーマットで表現できない場合、それは最も近い値にクリッピングされ、無効フラグがセットされます。

表8.4に、FCVT.int.Sの有効な入力の範囲と無効な入力の動作を示します。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | FCVT.W.S | FCVT.WU.S | FCVT.L.S | FCVT.LU.S |
| 最小有効入力（丸め後）  最大有効入力（丸め後） | -231  231-1 | 0  232-1 | -263  263-1 | 0  264-1 |
| 範囲外の負入力時の出力  -∞時の出力  範囲外の正入力時の出力  +∞もしくはNaN時の出力 | -231  -231  231-1  231-1 | 0  0  232-1  232-1 | -263  -263  263-1  263-1 | 0  0  264-1  264-1 |

表8.4：浮動小数点から整数への変換のドメインと無効な入力の動作

すべての浮動小数点から整数への整数と浮動小数点への変換命令は、rmフィールドに従って丸めます。

浮動小数点レジスタは、例外を発生させないFCVT.S.W rd、x0を使用して、浮動小数点の正のゼロに初期化することができます。

--2018/05/29

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

FCVT.int.fmt S W[U]/L[U] src1 RM dest OP-FP

FCVT.fmt.int S W[U]/L[U] src1 RM dest OP-FP

浮動小数点から浮動小数点への符号注入命令FSGNJ.S、FSGNJN.S、およびFSGNJX.Sは、rs1からの符号ビットを除くすべてのビットをとる結果を生成します。

FSGNJの場合、結果の符号ビットはrs2の符号ビットです。 FSGNJNの場合、結果の符号ビットはrs2の符号ビットの反対です。 FSGNJXの場合、符号ビットはrs1とrs2の符号ビットの排他的論理和です。

サインイン命令では、浮動小数点例外フラグは設定されません。

注：FSGNJ.S rx,ry,ryはryをrxに移動します（アセンブラ疑似オペレーションFMV.S rx,ry）。 FSGNJN.S rx,ry,ryは、ryの否定をrxに移動します（アセンブラ疑似演算FNEG.S rx,ry）。FSGNJX.S rx,ry,ryはryの絶対値をrxに移動します（アセンブラ疑似演算FABS.S rx、ry）。

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

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

-----------------------------------------------------------------------------------------------------------------------------------

符号注入命令は、浮動小数点MV、ABS、NEGを提供するだけでなく、超越数学関数ライブラリでのIEEE copySign演算と符号操作を含むいくつかの他の演算をサポートします。

MV、ABS、およびNEGは単一のレジスタ・オペランドだけを必要としますが、FSGNJ命令は2つ必要ですが、ほとんどのマイクロアーキテクチャでは、これらの比較的まれな命令のためのレジスタ読取り回数の減少の恩恵を受けるために最適化を追加することはほとんどありません。

この場合であっても、マイクロアーキテクチャは、両方のソースレジスタがFSGNJ命令で同じであることを単に検出し、単一のコピーのみを読み取ることができます。

浮動小数点レジスタと整数レジスタの間でビットパターンを移動するための命令が提供されています。

FMV.X.Wは、IEEE 754-2008エンコーディングで表される浮動小数点レジスタrs1の単精度値を整数レジスタrdの下位32ビットに移動します。

RV64の場合、デスティネーションレジスタの上位32ビットは浮動小数点数の符号ビットのコピーで埋められます。

FMV.W.Xは、IEEE 754-2008標準エンコーディングでエンコードされた単精度値を、整数レジスタrs1の下位32ビットから浮動小数点レジスタrdに移動します。

ビットは転送で変更されません。特に、非標準NaNのペイロードは保持されます。

-----------------------------------------------------------------------------------------------------------------------------------

FMV.W.XおよびFMV.X.W命令は、以前はFMV.S.XおよびFMV.X.S.と呼ばれていました。

Wの使用は、それらの解釈を行わずに32ビットを動かす命令として、そのセマンティクスとより一貫しています。

NaN-ボクシングを定義した後、これはより明確になりました。

既存のコードを妨害しないように、WとSの両方のバージョンがツールでサポートされます。

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

FMV.X.W S ０ src 000 dest OP-FP

FMV.W.X S ０ src 000 dest OP-FP

-----------------------------------------------------------------------------------------------------------------------------------

基本浮動小数点ISAは、実装が浮動小数点フォーマットの内部再コーディングをレジスタに採用して、異常値の処理を簡素化し、場合によっては機能ユニットの待ち時間を短縮できるように定義されています。

この目的のために、ベースISAは、整数レジスタファイルを直接読み書きする変換および比較演算を定義することによって、浮動小数点レジスタにおける整数値の表現を避けます。

これにより、整数と浮動小数点レジスタ間の明示的な移動が必要な一般的なケースの多くが削除され、一般的な混合フォーマットコードシーケンスの命令数とクリティカルパスが削減されます。

--2018/05/30

8.8単精度浮動小数点比較命令

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

FLT.SおよびFLE.Sは、IEEE 754-2008標準がシグナリング比較と呼ぶものを実行します。つまり、どちらかの入力がNaNの場合、無効演算例外が発生します。

FEQ.Sは静かな比較を実行します：シグナリングNaN入力だけが無効な操作例外を引き起こします。

3つの命令すべてについて、どちらかのオペランドがNaNの場合、結果は0になります。

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

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

8.9単精度浮動小数点分類命令

FCLASS.S命令は、浮動小数点レジスタrs1の値を調べ、浮動小数点数のクラスを示す10ビットのマスクを整数レジスタrdに書き込みます。

マスクのフォーマットについては、表8.5を参照してください。

プロパティが真の場合はrdの対応するビットが設定され、それ以外の場合はクリアされます。

rdの他のビットはすべてクリアされます。

rdのちょうど1ビットが設定されることに注意してください。

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

FCLASS S 0 src 001 dest OP-FP

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

表8.5：FCLASS命令の結果のフォーマット

ー2018/06/03

！浮動小数点まで来た。この後は倍精度とか４倍精度のだね

！ちょっと別件が入ったので、RISC-V翻訳はペースダウンするのだ。

！やめるわけでばないので、気を長～くしてまってておくれ。

！

－1ページ置いて次へ

第9章

倍精度浮動小数点の「D」標準拡張、

バージョン2.0

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

D拡張は、基本単精度命令サブセットFに依存します。

9.1 Dレジスタの状態

D拡張は、32の浮動小数点レジスタf0〜f31を64ビットに広げます（図8.1のFLEN = 64）。

fレジスタは、9.2節で後述するように、32ビットまたは64ビットの浮動小数点値を保持できるようになりました。

-----------------------------------------------------------------------------------------------------------------------------------

FLENは、F、D、およびQ拡張のどれがサポートされているかに応じて、32,64、または128にすることができます。

H、F、D、およびQを含む4つまでの異なる浮動小数点精度がサポートされています。

半精度Hスカラー値は、Vベクトル拡張がサポートされている場合にのみサポートされます。

より狭い値の9.2 NaNのボクシング

複数の浮動小数点精度がサポートされている場合、NaN-ボクシングと呼ばれる処理では、n <FLENの狭いnビット型の有効な値がFLENビットのNaN値の下位nビットに表されます。

有効なNaNボックス値の上位ビットはすべて1sでなければなりません。

したがって、有効なNaNボックスnビット値は、より広いmビット値（n <m≦FLEN）と見なされるとき、負の静穏NaN（qNaN）として表示されます。

-----------------------------------------------------------------------------------------------------------------------------------

ソフトウェアは、浮動小数点レジスタに格納されている現在のデータタイプを知らないかもしれないが、レジスタ値を保存して復元することができなければならないため、より幅の広い演算を使用してより狭い値を転送した結果を定義する必要があります。

一般的なケースは、呼び出し先保存レジスタですが、バリアック、ユーザーレベルのスレッドライブラリ、仮想マシンの移行、およびデバッグなどの機能には、標準的な規則も望ましいものです。

浮動小数点nビット転送演算は、IEEE標準フォーマットで保持されている外部値をfレジスタとの間で移動し、浮動小数点ロードおよびストア（FLn / FSn）および浮動小数点移動命令（FMV.n.X / FMV.X.n）を含みます。

n <FLENのfレジスタへのより狭いnビット転送は、宛先fレジスタのすべての上位FLEN-nビットを1に設定することにより、有効なNaNボックス値を作成します。

浮動小数点レジスタからのより狭いnビット転送は、上位FLEN-nビットを無視してレジスタの下位nビットを転送します。

浮動小数点演算および符号インジェクション演算は、fレジスタに保持されたFLENビット値に基づいて結果を計算します。

n <FLENである狭いnビット演算は、入力オペランドがNaNボックスで正しくチェックされる、すなわち、すべての上位FLEN-nビットが1であることをチェックします。

そうであれば、入力のn最下位ビットが入力値として使用され、それ以外の場合、入力値はnビット正準NaNとして扱われます。

nビットの浮動小数点結果は、デスティネーションfレジスタの最下位nビットに書き込まれ、すべて1が最上位のFLEN-nビットに書き込まれ、正当なNaNボックス値が生成されます。

整数から浮動小数点への変換（例えば、FCVT.S.X）は、FLENビットのデスティネーションレジスタを満たすためにFLENより狭い任意の結果をNaNボックスに入れます。

より狭いnビット浮動小数点値から整数（例えば、FCVT.X.S）への変換は正当なNaNボクシングをチェックし、正当なnビット値でない場合には入力をnビット正準NaNとして扱います。

-----------------------------------------------------------------------------------------------------------------------------------

このドキュメントの以前のバージョンでは、幅の狭いオペランドの結果をより狭いオペランドの値を保存することを要求する以外は、より狭いオペランドまたはより広いオペランドの結果をオペレーションに渡す動作は定義されていませんでした。

新しい定義では、この実装固有の動作が削除されますが、浮動小数点ユニットのレコーディングされていない実装とレコード化された実装の両方に対応します。

新しい定義は、値が正しく使用されない場合にNaNを伝播することによってソフトウェアエラーを捕捉するのにも役立ちます。

コード化されていない実装では、すべての浮動小数点演算の入力と出力にオペランドをIEEE標準形式に展開してパックします。

非再構成の実装に対するNaN-ボクシングのコストは、主に、より狭い演算の上位ビットが正当なNaN-ボックス化された値を表しているかどうかをチェックし、結果の上位ビットにすべて1を書き込むことにあります。

レコーディングされた実装では、浮動小数点値を表現するためのより便利な内部形式を使用し、すべての値を正規化して保持できるように指数ビットが追加されています。

レコード化された実装のコストは主に内部型と符号ビットを追跡するために必要な余分なタグ付けですが、これは指数部の内部でNaNをコード化することによって新しい状態ビットを追加することなく行うことができます。

記録フォーマットの内外で値を転送するために使用されるパイプラインには小さな変更が必要ですが、データパスとレイテンシのコストは最小限です。

レコーディング処理では、いずれの場合もワイドオペランドの入力正規化値のシフトを処理しなければならず、NaNボックス値の抽出は、先頭の0ビットをスキップせずに先頭の1ビットをスキップする以外は正規化と同様の処理です。データパス多重化を共有することができます。

9.3倍精度ロード命令とストア命令

FLD命令は、メモリから浮動小数点レジスタrdに倍精度浮動小数点値をロードします。

FSDは浮動小数点レジスタの倍精度値をメモリに格納します。

-----------------------------------------------------------------------------------------------------------------------------------

倍精度値は、NaNボックス化された単精度値です。

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | width | rd | opcode |  |

12 5 3 5 7

offset[11:0] base D dest LOAD-FP

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| imm[11:5] | rs2 | rs1 | width | imm[4:0] | opcode |  |

7 5 5 3 5 7

offset[11:5] src base D offset[4:0] STORE-FP

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

9.4倍精度浮動小数点計算命令

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

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

FADD/FSUB D src2 src1 RM dest OP-FP

FMUL/FDIV D src2 src1 RM dest OP-FP

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

FSQRT D 0 src RM dest OP-FP

↑単精度浮動小数点とほぼいっしょ D のところだけ違う

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| rs3 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

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

9.5倍精度浮動小数点変換および移動命令

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

FCVT.D.WまたはFCVT.D.Lは、整数レジスタrs1の32ビットまたは64ビットの符号付き整数をそれぞれ浮動小数点レジスタrdの倍精度浮動小数点数に変換します。

FCVT.WU.D、FCVT.LU.D、FCVT.D.WU、およびFCVT.D.LUバリアントは、符号なし整数値に変換されます。

FCVT.L [U] .DとFCVT.D.L [U]はRV32では不正です。

FCVT.int.Dの有効な入力の範囲と無効な入力の動作は、FCVT.int.Sの場合と同じです。

すべての浮動小数点から整数への整数と浮動小数点への変換命令は、rmフィールドに従って丸めます。

注FCVT.D.W [U]は常に正確な結果を生成し、丸めモードの影響を受けません。

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

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

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

倍精度から単精度および単精度から倍精度への変換命令FCVT.S.DおよびFCVT.D.Sは、OP-FPメジャーオペコード空間でエンコードされ、ソースおよびデスティネーションの両方が浮動小数点レジスタです。

rs2フィールドはソースのデータ型をエンコードし、fmtフィールドはデスティネーションのデータ型をエンコードします。

FCVT.S.DはRMフィールドに従って丸めます。 FCVT.D.Sは決してラウンドしません。

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

FCVT.S.D S D src RM dest OP-FP

FCVT.D.S D S src RM dest OP-FP

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

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

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

RV64の場合のみ、浮動小数点レジスタと整数レジスタの間でビットパターンを移動するための命令が提供されます。

FMV.X.Dは、浮動小数点レジスタrs1の倍精度値を整数レジスタrdのIEEE 754-2008標準エンコーディングの表現に移動します。

FMV.D.Xは、IEEE 754-2008標準エンコーディングでエンコードされた倍精度値を整数レジスタrs1から浮動小数点レジスタrdに移動します。

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

FMV.X.D D 0 src 000 dest OP-FP

FMV.D.X D 0 src 000 dest OP-FP

9.6倍精度浮動小数点比較命令

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

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

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

9.7倍精度浮動小数点分類命令

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

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

FCLASS D 0 src 001 dest OP-FP

-- 2018/07/08

-- 1ページ空き、次ページへ

第10章

クワッド精密浮動小数点の「Q」標準拡張、

バージョン2.0

この章では、IEEE 754-2008算術標準に準拠した128ビットバイナリ浮動小数点命令のQ標準拡張について説明します。

128ビットまたは4倍精度の2進浮動小数点命令サブセットの名前は "Q"で、RV64IFDが必要です。

浮動小数点レジスタは、単精度、倍精度、または4倍精度浮動小数点値（FLEN = 128）のいずれかを保持するように拡張されました。

セクション9.2で説明したNaN-boxingスキームが再帰的に拡張され、倍精度値内で単精度値がNaNボックス化されるようになりました。倍精度値自体は4倍精度値のNaNボックス化されています。

10.1四倍精度ロード命令とストア命令

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

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | width | rd | opcode |  |

12 5 3 5 7

offset[11:0] base Q dest LOAD-FP

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| imm[11:5] | rs2 | rs1 | width | imm[4:0] | opcode |  |

7 5 5 3 5 7

offset[11:5] src base Q offset[4:0] STORE-FP

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

10.2クワッド精密計算命令

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

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

表10.1：フォーマットフィールドのエンコード

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

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

FADD/FSUB Q src2 src1 RM dest OP-FP

FMUL/FDIV Q src2 src1 RM dest OP-FP

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

FSQRT Q 0 src RM dest OP-FP

↑単精度浮動小数点、倍精度浮動小数点とほぼいっしょ Q のところだけ違う

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| rs3 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

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

10.3四倍精度変換および移動命令

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

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

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

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

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

FCVT.S.Q S Q src RM dest OP-FP

FCVT.Q.S Q S src RM dest OP-FP

FCVT.D.Q D Q src RM dest OP-FP

FCVT.Q.D Q D src RM dest OP-FP

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

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

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

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

-----------------------------------------------------------------------------------------------------------------------------------

RV128はQ拡張でFMV.X.QとFMV.Q.Xをサポートしています。

10.4 4倍精度浮動小数点比較命令

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

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

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

10.5四倍精度浮動小数点分類命令

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

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |  |

5 2 5 5 3 5 7

FCLASS Q 0 src 001 dest OP-FP

-- 1ページ空き、次ページへ

第11章

小数点浮動小数点バージョン「L」の標準拡張、バージョン0.0

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

11.1 10進浮動小数点レジスタ

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

-----------------------------------------------------------------------------------------------------------------------------------

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

-- 1ページ空き、次ページへ

第12章

圧縮された命令のための "C"標準拡張、バージョン2.0

この章では、一般的な操作に短い16ビットの命令エンコーディングを追加することにより、静的および動的コードサイズを削減する、RISC-V標準圧縮命令セット拡張「C」の現在のドラフト提案について説明します。

C拡張は、基本ISA（RV32、RV64、RV128）のいずれかに追加することができ、これらのいずれかをカバーするために総称「RVC」を使用します。

通常、プログラム内のRISC-V命令の50％〜60％をRVC命令に置き換えることができるため、コードサイズを25％〜30％削減できます。

12.1概要

RVCは以下の場合に、一般的な32ビットRISC-V命令のより短い16ビットバージョンを提供する簡単な圧縮方式を使用します。

・即時オフセットまたはアドレス指定が小さいか、または

・レジスタの1つがゼロレジスタ（x0）、ABIリンクレジスタ（x1）、またはABIスタックポインタ（x2）、または

・デスティネーションレジスタと第1のソースレジスタが同一で​​あるか、または

・使用されるレジスタは8つの最も一般的なレジスタです。

C拡張は、他のすべての標準命令拡張と互換性があります。

C拡張は、16ビット命令を32ビット命令と自由に混在させることができ、後者は任意の16ビット境界で開始することができます。

C拡張を追加すると、JAL命令とJALR命令は命令のミスアライン例外を発生しなくなります。

-----------------------------------------------------------------------------------------------------------------------------------

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

圧縮された命令エンコーディングはRV32C、RV64C、RV128Cで共通ですが、表12.3に示すように、いくつかのオペコードはベースISAの幅に応じて異なる目的で使用されます。

たとえば、RV32Cは同じオペコードを使用して単精度浮動小数点値のロードと格納を圧縮するのに対して、64ビット整数値のロードと格納を圧縮するには、より広いアドレス空間のRV64CとRV128Cバリアントが追加のオペコードを必要とします。

同様に、RV128Cでは、128ビット整数値のロードと格納をキャプチャするために追加のオペコードが必要ですが、RV32CとRV64Cの倍精度浮動小数点値のロードと格納には同じオペコードが使用されます。

C拡張が実装されている場合は、関連する標準浮動小数点拡張（Fおよび/またはD）も実装されているときは常に、適切な圧縮浮動小数点ロードおよびストア命令を提供する必要があります。

さらに、RV32Cには、RV64CおよびRV128CのADDIWを圧縮するために同じオペコードが使用される、短距離サブルーチンコールを圧縮する圧縮ジャンプおよびリンク命令が含まれています。

-----------------------------------------------------------------------------------------------------------------------------------

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

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

したがって、RV32Cでの圧縮サポートを提供する動機です。

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

さまざまなベースレジスタ幅に対して異なる目的でオペコードを再利用するとドキュメンテーションが複雑になりますが、複数のベースISAレジスタ幅をサポートするデザインであっても実装の複雑さへの影響は小さいです。

圧縮された浮動小数点ロードおよびストア・バリアントは、より広い整数ロードおよびストアと同じレジスタ指定子を有する同じ命令フォーマットを使用します。

RVCは、各RVC命令がベースISA（RV32I / E、RV64I、またはRV128I）または存在する場合はFおよびD標準拡張のいずれかで単一の32ビット命令に展開されるという制約の下で設計されました。

この制約を採用すると、2つの主な利点があります。

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

-----------------------------------------------------------------------------------------------------------------------------------

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

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

-----------------------------------------------------------------------------------------------------------------------------------

コード長を改善するために、可変長命令セットが長く使用されてきました。

たとえば、1950年代後半に開発されたIBM Stretch [6]には、32ビットおよび64ビット命令を含むISAがありました。ここでは、32ビット命令の一部は完全な64ビット命令の圧縮バージョンでした。

Stretchは、インデックスレジスタの1つのみを参照できる短い分岐命令を使用して、より短い命令フォーマットのいくつかでアドレス可能なレジスタのセットを制限するという概念も採用しました。

後のIBM 360アーキテクチャ[3]は、16ビット、32ビット、または48ビットの命令フォーマットを使用した単純な可変長命令エンコーディングをサポートしていました。

1963年にCDCはRISCアーキテクチャの前身であるCray設計のCDC 6600 [28]を導入しました。このアーキテクチャでは、2つの長さ、15ビットと30ビットの命令を持つレジスタ豊富なロードストアアーキテクチャが導入されました。

後のCray-1デザインは、16ビットと32ビットの命令長を持つ非常に似た命令フォーマットを使用していました。

1980年代の初期のRISC ISAは、ワークステーション環境では合理的で、コードサイズよりも性能が優れていましたがしかし組込みシステムでは合理的でありませんでした。

したがって、ARMとMIPSは、その後、標準の32ビット幅の命令の代わりに代替の16ビット幅の命令セットを提供することにより、より小さなコードサイズを提供するISAのバージョンを作成しました。

圧縮されたRISC ISAは、開始ポイントに対して約25〜30％のコードサイズを縮小し、80x86よりも大幅に小さいコードを生成しました。

この結果は、可変長CISC ISAが16ビットおよび32ビットフォーマットのみを提供していたRISC ISAsよりも小さくなければならないということだったため、いくつかのことを驚かせました。

元のRISC ISAsは、これらの計画外圧縮命令を含むために十分なオペコード空間を自由に残さなかったので、完全な新しいISAとして開発されました。

これは、コンパイラが独立した圧縮ISA用に異なるコードジェネレータを必要とすることを意味していました。

最初の圧縮RISC ISA拡張（例えば、ARM ThumbおよびMIPS16）は固定の16ビット命令サイズのみを使用しており、静的なコードサイズは大幅に削減されましたが、動的な命令数が増加をもたらし、元の固定幅の32ビット命令サイズと比較してパフォーマンスが低下しました。

これにより、パフォーマンスは純粋な32ビット命令に似ていましたが、(たとえば、ARM Thumb2、microMIPS、PowerPC VLEなど)、16ビットと32ビットの混在した命令長の圧縮RISC ISAデザインの第2世代が開発され大幅なコードサイズが削減されました。

残念なことに、圧縮されたISAのこれらの異なる世代は、相互に互換性がなく、元の圧縮されていないISAと互換性がなく、ドキュメンテーション、実装、およびソフトウェアツールのサポートが大幅に複雑になります。

一般的に使用されている64ビットISAのうち、現在PowerPCとmicroMIPSのみが圧縮命令フォーマットをサポートしています。

スタティックコードサイズと動的命令フェッチ帯域幅が重要なメトリックであるため、モバイルプラットフォーム（ARM v8）で最も一般的な64ビットISAに圧縮命令フォーマットが含まれていないことは驚くべきことです。

大規模システムでは静的コードサイズは大きな問題ではありませんが、商用ワークロードを実行しているサーバーでは命令フェッチ帯域幅が大きなボトルネックになりがちです。

25年間の見通しから知見を得て、RISC-Vは、最初から圧縮された命令をサポートするように設計されており、RVCがベースISAの上に簡単に拡張されるように（多くの他の拡張機能と共に）追加されています。

RVCの哲学は、組み込みアプリケーションのコードサイズを削減し、命令キャッシュのミスが少なくなるため、すべてのアプリケーションのパフォーマンスとエネルギー効率を向上させることです。

Watermanは、RVCが命令ビットを25％〜30％削減し、命令キャッシュミスを20％〜25％削減すること、または命令キャッシュサイズを2倍にするのとほぼ同じパフォーマンスに影響することを示しています[33]。

12.2圧縮された命令フォーマット

表12.1に8つの圧縮命令フォーマットを示します。

CR、CI、CSSは32個のRVIレジスタを使用できますが、CIW、CL、CS、CBはわずか8個に制限されています。

表12.2にレジスタx8〜x15に対応する一般的なレジスタを示します。

スタックポインタをベースアドレスレジスタとして使用するロード命令とストア命令は別々のバージョンが存在することに注意してください。スタックへの保存と復元は非常に一般的であるため、CIおよびCSS形式を使用して32のすべてのデータレジスタにアクセスできるようにします。

CIWは、ADDI4SPN命令のために8ビットの即値を供給します。

-----------------------------------------------------------------------------------------------------------------------------------

RISC-V ABIは頻繁に使用されるレジスタをレジスタx8〜x15にマップするように変更されました。

これにより、連続して自然にアライメントされたレジスタ番号のセットを有することによって解凍デコーダが簡略化され、16個の整数レジスタしか持たないRV32Eサブセットベース仕様とも互換性があります。

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

-----------------------------------------------------------------------------------------------------------------------------------

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

このフォーマットは、すべての命令の同じ場所に2つのレジスタソース指定子のビットを保持するように設計されていて、宛先レジスタフィールドは移動することができます。

完全な5ビットデスティネーションレジスタ指定子が存在する場合、それは32ビットRISC-Vエンコーディングと同じ場所にあります。

イミディエイトが符号拡張されている場合、符号拡張は常にビット12から始まります。

イミディエイトフィールドは、ベース仕様のようにスクランブルされており、必要な即時マルチプレクサの数を減らしています。

-----------------------------------------------------------------------------------------------------------------------------------

即値フィールドは、順次命令ではなく命令フォーマットでスクランブルされるので、可能な限り多くのビットがすべての命令の同じ位置にあり、それにより実装が単純化される。

例えば、即値ビット17-10は、常に同じ命令ビット位置から供給される。

5つの他の即値ビット（5,4,3,1および0）は、わずか2つのソース命令ビットを有し、4つ（9,7,6および2）は3つのソースを有し、1つ（8）は4つのソースを有します。

多くのRVC命令では、ゼロ値のイミディエイトは禁止され、x0は有効な5ビットのレジスタ指定子ではありません。

これらの制約は、より少ないオペランドビットを必要とする他の命令のための符号化空間を解放します。

– 2018/07/15

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

表12.1：圧縮された16ビットRVC命令フォーマット。

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

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

12.3ロードとストア命令

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

RVCには、ロードとストアの2種類があります。

ABIスタック・ポインタx2をベース・アドレスとして使用し、任意のデータ・レジスタをターゲットにすることができます。

もう1つは8つのベースアドレスレジスタの1つと8つのデータレジスタの1つを参照できます。

スタックポインタベースのロードとストア

15 13 12 11 7 6 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct3 | imm | rd | imm | op |  |

3 1 5 5 2

C.LWSP offset[5] dest≠0 offset[4:2][7:6] C2

C.LDSP offset[5] dest≠0 offset[4:3][8:6] C2

C.LQSP offset[5] dest≠0 offset[4][9:6] C2

C.FLWSP offset[5] dest offset[4:2][7:6] C2

C.FLDSP offset[5] dest offset[4:3][8:6] C2

これらの命令はCI形式を使用します。

C.LWSPはメモリからレジスタrdに32ビット値をロードします。

スタックポインタx2に4でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。

lw rd, offset [7:2]（x2）に展開されます。

C.LDSPは、メモリからレジスタrdに64ビット値をロードするRV64C / RV128C専用命令です。

スタックポインタx2に8でスケーリングされたゼロ拡張オフセットを加算することによって、実効アドレスを計算します。

ld rd, offset [8:3]（x2）に展開されます。

C.LQSPは、メモリからレジスタrdに128ビットの値をロードするRV128C専用の命令です。

スタックポインタx2に16でスケーリングされたゼロで拡張されたオフセットを加算することによって、実効アドレスを計算します。

lq rd、offset [9:4]（x2）に展開されます。

C.FLWSPは、メモリから浮動小数点レジスタrdに単精度浮動小数点値をロードするRV32FC専用命令です。

スタックポインタx2に4でスケーリングしたゼロ拡張オフセットを加算することにより、実効アドレスを計算します。

それはflw rd, offset [7:2]（x2）に展開されます。

C.FLDSPは、メモリから浮動小数点レジスタrdに倍精度浮動小数点値をロードするRV32DC / RV64DC専用命令です。

スタックポインタx2に8でスケーリングされたゼロ拡張オフセットを加算することによって、実効アドレスを計算します。

fld rd, offset [8:3]（x2）に展開されます。

15 13 12 7 6 2 1 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| funct3 | imm | rs2 | op |  |

3 6 5 2

C.SWSP offset[5:2][7:6] src C2

C.SDSP offset[5:3][8:6] src C2

C.SQSP offset[5:4][9:6] src C2

C.FSWSP offset[5:2][7:6] src C2

C.FSDSP offset[5:3][8:6] src C2

これらの手順では、CSS形式を使用します。

C.SWSPはレジスタrs2の32ビット値をメモリに格納します。

スタックポインタx2に4でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。

sw rs2, offset [7:2]（x2）に展開されます。

C.SDSPは、レジスタrs2の64ビット値をメモリに格納するRV64C / RV128C専用の命令です。

これは、ゼロで拡張されたオフセットを8でスケーリングしたものをスタックポインタx2に加算することによって実効アドレスを計算します。

sd rs2、offset [8：3]（x2）に展開されます。

C.SQSPは、レジスタrs2の128ビット値をメモリに格納するRV128C専用の命令です。

これは、スタックポインターx2にゼロで拡張されたオフセットを16でスケーリングしたものを加算することによって実効アドレスを計算します。

sq rs2, offset [9:4]（x2）に展開されます。

C.FSWSPは浮動小数点レジスタrs2の単精度浮動小数点値をメモリに格納するRV32FC専用命令です。

スタックポインタx2に4でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。

fsw rs2, offset [7:2]（x2）に展開されます。

C.FSDSPは、浮動小数点レジスタrs2の倍精度浮動小数点値をメモリに格納するRV32DC / RV64DC専用の命令です。

これは、ゼロで拡張されたオフセットを8でスケーリングしたものをスタックポインタx2に加算することによって実効アドレスを計算します。

fsd rs2, offset [8:3]（x2）に展開されます。

– 2018/07/16

-----------------------------------------------------------------------------------------------------------------------------------

関数の入口/出口でのレジスタ保存/復元コードは、静的コードサイズのかなりの部分を表します。

RVCのスタックポインタベースの圧縮ロードとストアは、動的命令帯域幅を削減してパフォーマンスを向上させながら、セーブ/リストアスタティックコードサイズを2倍に減らすのに有効です。

他のISAで保存/復元コードのサイズをさらに減らすために使用される一般的なメカニズムは、ロード・マルチプルおよびストア・マルチプル・インストラクションです。

RISC-Vにこれらを採用することを検討しましたが、これらの手順には次のような欠点がありました。：

・これらの命令はプロセッサの実装を複雑にします。

・仮想メモリシステムでは、一部のデータアクセスが物理メモリに常駐する可能性があり、実行できないものもあります。これは部分的に実行される命令に対して新しい再起動メカニズムを必要とします。

・残りのRVC命令とは異なり、複数ロードと複数ストアに相当するIFDはありません。

・RVC命令の残りの部分とは異なり、コンパイラは、命令を生成し、それらを保存して格納する可能性を最大にするためにレジスタを割り当てるために、これらの命令を認識しなければならず、それらは保存され、順番に復元されるからです。

・シンプルなマイクロアーキテクチャの実装は、負荷の周りに他の命令をスケジューリングして複数の命令を格納する方法を制限し、潜在的なパフォーマンスの低下につながります。

・シーケンシャルなレジスタ割り付けの要望は、CIW、CL、CS、およびCBフォーマット用に選択された機能レジスタと競合する可能性があります。

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

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

レジスタベースのロードおよびストア

15 13 12 10 9 7 6 5 4 2 1 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct3 | imm | rs1’ | imm | rd’ | op |  |

3 3 3 2 3 2

C.LW offset[5:3] base offset[2|6] dest C0

C.LD offset[5:3] base offset[7:6] dest C0

C.LQ offset[5|4|8] base offset[7:6] dest C0

C.FLW offset[5:3] base offset[2|6] dest C0

C.FLD offset[5:3] base offset[7:6] dest C0

これらの命令は、CL形式を使用します。

C.LWはメモリからレジスタrd'に32ビット値をロードします。

レジスタrs1'のベースアドレスに4でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。

lw rd'、offset [6：2]（rs1'）に展開されます。

C.LDは、メモリからレジスタrd'に64ビット値をロードするRV64C / RV128C専用命令です。

レジスタrs1'のベースアドレスに8でスケーリングしたゼロ拡張オフセットを加算して実効アドレスを計算します。

ld rd'、offset [7：3]（rs1'）に展開されます。

C.LQは、メモリからレジスタrd'に128ビットの値をロードするRV128C専用命令です。

レジスタrs1'のベースアドレスに16でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。

lq rd'、offset [8：4]（rs1'）に展開されます。

C.FLWは、単精度浮動小数点値をメモリから浮動小数点レジスタrd'にロードするRV32FC専用命令です。

レジスタrs1'のベースアドレスに4でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。

fld rd'、offset [6：2]（rs1'）に展開されます。

C.FLDは、メモリから浮動小数点レジスタrd'に倍精度浮動小数点値をロードするRV32DC / RV64DC専用命令です。

レジスタrs1'のベースアドレスに8でスケーリングしたゼロ拡張オフセットを加算して実効アドレスを計算します。

fld rd'、offset [7：3]（rs1'）に展開されます。

15 13 12 10 9 7 6 5 4 2 1 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct3 | imm | rs1’ | imm | rs2’ | op |  |

3 3 3 2 3 2

C.SW offset[5:3] base offset[2|6] src C0

C.SD offset[5:3] base offset[7:6] src C0

C.SQ offset[5|4|8] base offset[7:6] src C0

C.FSW offset[5:3] base offset[2|6] src C0

C.FSD offset[5:3] base offset[7:6] src C0

これらの命令は、CSフォーマットを使用します。

C.SWはレジスタrs2'の32ビット値をメモリに格納します。

レジスタrs1'のベースアドレスに4でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。

sw rs2'、offset [6：2]（rs1'）に展開されます。

C.SDは、レジスタrs2'の64ビット値をメモリに格納するRV64C / RV128C専用の命令です。

レジスタrs1'のベースアドレスに8でスケーリングしたゼロ拡張オフセットを加算して実効アドレスを計算します。

sd rs2'、offset [7：3]（rs1'）に展開されます。

C.SQはレジスタrs2'の128ビット値をメモリに格納するRV128C専用命令です。

レジスタrs1'のベースアドレスに16でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。

sq rs2'、offset [8：4]（rs1'）に展開されます。

C.FSWは浮動小数点レジスタrs2'の単精度浮動小数点値をメモリに格納するRV32FC専用命令です。

レジスタrs1'のベースアドレスに4でスケーリングされたゼロ拡張オフセットを加算して実効アドレスを計算します。

fsw rs2'、offset [6：2]（rs1'）に展開されます。

C.FSDは、浮動小数点レジスタrs2'の倍精度浮動小数点値をメモリに格納するRV32DC / RV64DC専用の命令です。

レジスタrs1'のベースアドレスに8でスケーリングしたゼロ拡張オフセットを加算して実効アドレスを計算します。

fsd rs2'、offset [7：3]（rs1'）に展開されます。

--

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

12.4制御転送命令

RVCは無条件ジャンプ命令と条件付き分岐命令を提供します。

基本RVI命令の場合と同様に、すべてのRVC制御転送命令のオフセットは2バイトの倍数になります。

15 13 12 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| funct3 | imm | op |  |

3 11 2

C.J offset[11|4|9:8|10|6|7|3:1|5] C1

C.JAL offset[11|4|9:8|10|6|7|3:1|5] C1

これらの命令はCJ形式を使用します。

C.Jは無条件制御転送を行います。

オフセットは符号拡張され、ジャンプターゲットアドレスを形成するためにPCに追加されます。

従って、C.Jは±2KiBの範囲を対象とすることができます。

C.Jはjal x0、offset [11：1]に展開されます。

C.JALはC.Jと同じ操作を実行するが、リンクレジスタx1にジャンプ（pc + 2）に続く命令のアドレスを追記するRV32C専用命令です。

C.JALはjal x1、offset [11：1]に展開されます。

15 12 11 7 6 2 1 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| funct4 | rs1 | rs2 | op |  |

4 5 5 2

C.JR src≠0 0 C2

C.JALR src≠0 0 C2

These instructions use the CR format.

C.JR (jump register) performs an unconditional control transfer to the address in register rs1.

C.JR expands to jalr x0, rs1, 0.

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

C.JALR expands to jalr x1, rs1, 0.

-----------------------------------------------------------------------------------------------------------------------------------

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

15 13 12 10 9 7 6 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct3 | imm | rs1’ | imm | op |  |

3 3 3 5 2

C.BEQZ offset[8|4:3] src offset[7:6|2:1|5] C1

C.BNEZ offset[8|4:3] src offset[7:6|2:1|5] C1

これらの命令はCB形式を使用します。

C.BEQZは、条件付き制御転送を実行する。

オフセットは符号拡張され、PCに追加されて分岐先アドレスを形成します。

したがって、±256Bの範囲をターゲットにすることができます。

C.BEQZは、レジスタrs1 'の値がゼロの場合に分岐を取ります。

beq rs1 '、x0、offset [8：1]に展開されます。

C.BNEZも同様に定義されますが、rs1 'にゼロ以外の値が含まれている場合は分岐が必要です。

bn rs1 '、x0、offset [8：1]に展開されます。

12.5整数計算命令

RVCは、整数演算および定数生成のためのいくつかの命令を提供します。

整数定数生成命令

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

15 13 12 11 7 6 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct3 | imm[5] | rd | Imm[4:0] | op |  |

3 1 5 5 2

C.LI imm[5] dest≠0 imm[4:0] C1

C.LUI nzuimm[17] dest≠{0,2} nzuimm[16:12] C1

C.LIは、符号拡張された6ビットの即値immをレジスタrdにロードします。

C.LIは、rd≠x0のときのみ有効です。

C.I.Iは、追加、x0、imm [5：0]に展開されます。

C.LUIは、デスティネーションレジスタのビット17-12に非ゼロの6ビット即値フィールドをロードし、下位12ビットをクリアし、ビット17をデスティネーションのすべての上位ビットに符号拡張します。

C.LUIは、rd≠{x0、x2}のときとイミディエートがゼロに等しくないときにのみ有効です。

C.LUIはlui rd、nzuimm [17:12]に拡張されます。

整数レジスタ - 即値演算

これらの整数レジスタ即値操作は、CI形式でエンコードされ、非x0整数レジスタおよび6ビット即値に対して操作を実行します。

即値はゼロにすることはできません。

15 13 12 11 7 6 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct3 | imm[5] | rd/rs1 | Imm[4:0] | op |  |

3 1 5 5 2

C.ADDI nzimm[5] dest nzimm[4:0] C1

C.ADDIW imm[5] dest≠0 imm[4:0] C1

C.ADDI16SP nzimm[9] 2 nzimm[4|6|8:7|5] C1

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

C.ADDIは、非ゼロ符号拡張6ビット即値をレジスタrdの値に加算し、その結果をrdに書き込みます。

C.ADDIは、addi rd、rd、nzimm [5：0]に拡大します。

C.ADDIWはRV64C / RV128Cのみの命令で、同じ計算を実行しますが32ビットの結果を生成し、結果を64ビットに符号拡張します。

C.ADDIWは、追加、rd、imm [5：0]に拡張されます。

即値はC.ADDIWの場合はゼロであり、これはsext.w rdに対応します。

C.ADDI16SPはオペコードをC.LUIと共有しますが、宛先フィールドはx2であす。

C.ADDI16SPは、非ゼロ符号拡張6ビット即値をスタック・ポインタ（sp = x2）の値に加算します。ここで、イミディエートは、範囲内の16の倍数（-512,496）を表すようにスケーリングされます。

C.ADDI16SPは、プロシージャのプロローグとエピローグでスタックポインタを調整するために使用されます。

addi x2、x2、nzimm [9：4]に展開されます。

-- プロローグ、エピローグ っていうのがいまいちわからない、何をさすんだろう。

-- -512,496 って 16進で FFF8\_2E10、正としても 0007\_D1F0 なのでなんか中途半端。どっから出てきた数字だろう。

– あ、いや違った、もしかすると -512 と 496 かな、FE00 と 01F0 なのでこっちが正解っぽい。496は512-16だし。

-----------------------------------------------------------------------------------------------------------------------------------

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

15 13 12 5 4 2 1 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| funct3 | imm | rd’ | op |  |

3 8 3 2

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

C.ADDI4SPNは、CIW形式のRV32C / RV64Cのみの命令で、ゼロで拡張された非ゼロの即値を4でスケーリングした値をスタックポインタx2に加算し、その結果をrd’に書き込みます。

この命令は、スタック割り当て変数へのポインタを生成するために使用され、addi rd0、x2、nzuimm [9：2]に展開されます。

15 13 12 11 7 6 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct3 | shamt[5] | rd/rs1 | shamt[4:0] | op |  |

3 1 5 5 2

C.SLLI shamt[5] dest≠0 shamt[4:0] C2

C.SLLIは、レジスタrdの値の論理左シフトを実行し、結果をrdに書き込むCI形式の命令です。

シフト量はshamtフィールドにエンコードされます。shamt [5]はRV32Cではゼロでなければなりません。

RV32CおよびRV64Cの場合、シフト量はゼロでない必要があります。

RV128Cでは、64のシフトをエンコードするためにシフト量ゼロが使用されます。

C.SLLIは、shamt = 0のRV128Cを除いて、slli rd、rd、shamt [5：0]に展開され、slli rd、rd、64に展開されます。

15 13 1 2 11 10 9 7 6 2 1 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct3 | shamt[5] | funct2 | rd’/rs1’ | shamt[4:0] | op |  |

3 1 2 3 5 2

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

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

C.SRLIは、レジスタrd 'の値の論理右シフトを実行し、その結果をrd'に書き込むCB形式の命令です。

シフト量はshamtフィールドにエンコードされます。shamt [5]はRV32Cではゼロでなければなりません。

RV32CおよびRV64Cの場合、シフト量はゼロでない必要があります。

RV128Cでは、64のシフトをエンコードするためにシフト量ゼロが使用されます。

さらに、シフト量はRV128Cに対して符号拡張されているため、有効なシフト量は1-31,64,96-127です。

C.SRLIは、srli rd '、rd'、shamt [5：0]に展開され、ただしshamt = 0のRV128Cはsrli rd '、rd'、64に展開されます。

C.SRAIはC.SRLIと同様に定義されますが、代わりに算術右シフトを実行します。

C.SRAIはsrai rd '、rd'、shamt [5：0]に展開されます。

-----------------------------------------------------------------------------------------------------------------------------------

左シフトは、通常、右シフトより頻繁で、左シフトはアドレス値を拡大するために頻繁に使用されるためです。

したがって、右シフトはより少ない符号化空間が与えられ、他のすべての直後が符号拡張された符号化象限に配置されます。

RV128の場合、即座に6ビットシフト量を符号拡張することが決定されました。

デコードの複雑さを軽減することは別として、128ビットアドレスポインタの上位部分に配置されたタグの抽出を可能にするために、96-127の右シフト量が64-95よりも有用であると考えています。

RV128Cは、RV32CおよびRV64Cと同じポイントでフリーズしないため、128ビットのアドレス空間コードの一般的な使用を評価できます。

15 13 1 2 11 10 9 7 6 2 1 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct3 | imm[5] | funct2 | rd’/rs1’ | imm[4:0] | op |  |

3 1 2 3 5 2

C.ANDI imm[5] C.ANDI dest imm[4:0] C1

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

C.ANDIはandi rd '、rd'、imm [5：0]に展開されます。

整数レジスタ - レジスタ演算

15 12 11 7 6 2 1 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| funct4 | rd/rs1 | rs2 | op |  |

4 5 5 2

C.MV dest≠0 src≠0 C2

C.ADD dest≠0 src≠0 C2

これらの命令はCR形式を使用します。

C.MVはレジスタrs2の値をレジスタrdにコピーします。

C.MVは、add rd、x0、rs2に展開される。

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

C.ADDは、add rd、rd、rs2に展開されます。

15 10 9 7 6 5 4 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct6 | rd'/rs’ | funct | rs2’ | op |  |

6 3 2 3 2

C.AND dest C.AND src C1

C.OR dest C.OR src C1

C.XOR dest C.XOR src C1

C.SUB dest C.SUB src C1

C.ANDW dest C.ANDW src C1

C.SUBW dest C.SUBW src C1

これらの命令は、CSフォーマットを使用します。

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

C.ANDはrd '、rd'、rs2 'に展開されます。

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

C.ORはrd '、rd'、rs2 'に展開されます。

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

C.XORはxor rd '、rd'、rs2 'に展開されます。

C.SUBは、レジスタrd 'の値をレジスタrd'の値から減算し、その結果をレジスタrd 'に書き込む。

C.SUBはsub rd '、rd'、rs2 'に展開されます。

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

C.ADDWはaddw rd '、rd'、rs2 'に展開されます。

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

C.SUBWはsubw rd '、rd'、rs2 'に展開されます。

-----------------------------------------------------------------------------------------------------------------------------------

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

定義違法命令

15 13 12 11 7 6 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |  |

3 1 5 5 2

0 0 0 0 0

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

-----------------------------------------------------------------------------------------------------------------------------------

メモリ空間のゼロエンドまたは存在しない部分を実行しようとする試みをトラップするのに役立つ不正な命令であることをすべてゼロ命令で予約します。

すべてゼロでない値は、非標準拡張では再定義されるべきではありません。

同様に、存在しないメモリ領域に見られる別の共通の値をキャプチャするには、すべてのビットが1（RISC-V可変長符号化方式の非常に長い命令に対応）に設定された命令を不正なものとして予約します。

NOP命令

15 13 12 11 7 6 2 1 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct3 | imm[5] | rd/rs1 | imm[4:0] | op |  |

3 1 5 5 2

C.NOP 0 0 0 C1

C.NOPは、PCを前進させることを除いて、ユーザーが見ることのできる状態を変更しないCI形式の命令です。

C.NOPはc.addi x0、0として符号化され、addi x0、x0、0に展開されます。

-----------------------------------------------------------------------------------------------------------------------------------

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

ブレークポイント命令

15 12 11 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| funct4 | 0 | op |  |

4 10 2

C.EBREAK 0 C2

デバッガは、デバッグ環境にコントロールを戻すために、ebreakに展開されるC.EBREAK命令を使用することができます。

C.EBREAKはオペコードをC.ADD命令と共有しますが、rdとrs2が両方ともゼロであるため、CR形式も使用できます。

12.6 LR / SCシーケンスにおけるC命令の使用

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

12.7 RVC命令セットリスト

表12.3に、RVCの主要なオペコードのマップを示します。

下位2ビットがセットされたオペコードは、基本ISAの命令を含む16ビットよりも広い命令に対応する。

いくつかの命令は特定のオペランドに対してのみ有効です。 無効な場合、オペコードが将来の標準拡張のために予約されていることを示すRES、 オペコードが非標準拡張用に予約されていることを示すNSE。 オペコードが将来の標準マイクロアーキテクチャヒントのために予約されていることを示すためにHINTを使用します。

HINTとマークされた命令は、ヒントが効果を持たない実装ではno-opsとして実行する必要があります。

-----------------------------------------------------------------------------------------------------------------------------------

HINT命令は、パフォーマンスに影響する可能性があるがアーキテクチャ状態に影響を与えないマイクロアーキテクチャヒントの将来の追加をサポートするように設計されています。

単純な実装がHINTエンコーディングを無視し、HINTをアーキテクチャ状態を変更しない通常の操作として実行できるように、HINTエンコーディングが選択されています。

たとえば、宛先レジスタがx0の場合、C.ADDはHINTで、5ビットのrs2フィールドはHINTの詳細をエンコードします。

しかし、簡単な実装では、HINTをレジスタx0への加算として単に実行するだけで、効果はありません。

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

表12.3：RVCオペコードマップ

表12.4-12.6に、RVC命令を示します。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 000 | 0 | | | 0 | 00 | Illegal instruction |
| 000 | Nzuimm[5:4|9:6|2|3] | | | rd’ | 00 | C.ADDI4SPN(RES,nzuimm=0) |
| 001 | uimm[5:3] | rs1’ | uimm[7:6] | rd’ | 00 | C.FLD(RV32/64) |
| 001 | Uimm[5:4|8] | rs1’ | uimm[7:6] | rd’ | 00 | C.LQ(RV128) |
| 010 | uimm[5:3] | rs1’ | uimm[2|6] | rd’ | 00 | C.LW |
| 011 | uimm[5:3] | rs1’ | uimm[2|6] | rd’ | 00 | C.FLW(RV32) |
| 011 | uimm[5:3] | rs1’ | uimm[7:6] | rd’ | 00 | C.LD(RV64/128) |
| 100 | ー | | | | 00 | Reserved |
| 101 | uimm[5:3] | rs1’ | uimm[7:6] | rs2’ | 00 | C.FSD(RV32/64) |
| 101 | uimm[5:4|8] | rs1’ | uimm[7:6] | rs2’ | 00 | C.SQ(RV128) |
| 110 | uimm[5:3] | rs1’ | uimm[2|6] | rs2’ | 00 | C.SW |
| 111 | uimm[5:3] | rs1’ | uimm[2|6] | rs2’ | 00 | C.FSW(RV32) |
| 111 | uimm[5:3] | rs1’ | uimm[7:6] | rs2’ | 00 | C.SD(RV64/128) |

表12.4：RVC、Quadrant 0の命令一覧

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

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

表12.5：RVC、Quadrant 1の命令一覧

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

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

表12.6：RVC、Quadrant 2の命令リスト

-- 1ページ空き、次ページへ

第13章

"B"ビット操作のための標準拡張、バージョン0.0

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

-----------------------------------------------------------------------------------------------------------------------------------

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

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

-- 1ページ空き、次ページへ

第14章

動的に翻訳された言語用の "J"標準拡張、バージョン0.0

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

-----------------------------------------------------------------------------------------------------------------------------------

多くの一般的な言語は、通常、JavaやJavascriptを含む動的翻訳を介して実装されています。

これらの言語は、ダイナミックチェックとガベージコレクションのための追加のISAサポートから恩恵を得ることができます。

-- 1ページ空き、次ページへ

第15章

トランザクションメモリのための "T"標準拡張、バージョン0.0

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

-----------------------------------------------------------------------------------------------------------------------------------

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

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

-- 1ページ空き、次ページへ

第16章

パック℃-SIMD命令の "P"標準拡張命令、バージョン0.1

-----------------------------------------------------------------------------------------------------------------------------------

第5回RISC-Vワークショップでの議論では、大規模な浮動小数点SIMD演算のV拡張を標準化することを支持して、浮動小数点レジスタのこのパックドSIMD提案を破棄したいとの要望が示されました。

しかし、小さなRISC-V実装の整数レジスタで使用するためのパックドSIMD固定小数点演算に関心がありました。

この章では、RISC-V用の標準的なパックドSIMD拡張の概要を説明します。

将来のパックドSIMD拡張の標準セットに対して、命令サブセット名 "P"を予約しました。

他の拡張機能の多くは、整数ユニットとは別のワイドデータレジスタとデータパスを利用して、パックドSIMD拡張を構築することができます。

-----------------------------------------------------------------------------------------------------------------------------------

Lincoln Labs TX-2 [9]で初めて導入されたPacked-SIMD拡張は、データ並列コードでより高いスループットを提供する一般的な方法となっています。

以前の商用マイクロプロセッサのインプリメンテーションには、Intel i860、HP PA-RISC MAX [19]、SPARC VIS [29]、MIPS MDMX [12]、PowerPC AltiVec [8]、Intel x86 MMX / SSE [24,26] が含まれていおり、最近のデザインにはIntel x86 AVX [20]やARM Neon [11]などが含まれています。

この章では、packed-SIMDを追加するための標準的なフレームワークについて説明しますが、このような設計には積極的に取り組んでいません。

我々の意見では、パックドSIMD設計は、既存のワイド・データパス・リソースを再利用する際の妥当な設計ポイントであり、

重要な追加リソースをデータ並列実行に専念させる場合は、従来のベクタ・アーキテクチャに基づく設計がより良い選択であり、V拡張を使用する必要があります。

RISC-VパックドSIMD拡張命令は、浮動小数点レジスタ（f0〜f31）を再利用します。

これらのレジスタは、FLEN = 32〜FLEN = 1024の幅を持つように定義できます。

標準浮動小数点命令サブセットは、幅32ビット（ "F"）、64ビット（2D "）、または128ビット（" Q "）のレジスタを必要とします。

-----------------------------------------------------------------------------------------------------------------------------------

浮動小数点レジスタは、パックドSIMD値の代わりに使用するのが当然です、整数レジスタ（PA-RISCおよびAlphaパックSIMD拡張）は、制御レジスタおよびアドレス値の整数レジスタを解放し、SIMD浮動小数点実行のためのスカラー浮動小数点ユニットの再利用を簡素化し、デカップリングされた整数/浮動小数点ハードウェア設計に自然につながります。

浮動小数点ロードおよびストア命令エンコーディングはまた、より広いパックドSIMDレジスタを処理するための空間を有します。

ただし、パックドSIMD値に浮動小数点レジスタを再利用すると、浮動小数点値のコード化された内部フォーマットを使用するのが難しくなります。

既存の浮動小数点ロード命令とストア命令は、メモリからfレジスタにさまざまなサイズのワードをロードして格納するために使用されます。

ベースISAは32ビットと64ビットのロードとストアをサポートしますが、LOAD-FP命令とSTORE-FP命令のエンコードでは、表16.1に示すように8種類の異なる幅をエンコードできます。

パックドSIMD操作で使用する場合、ハードウェアに非自然にアライメントされたロードおよびストアをサポートすることが望ましいです。

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

表16.1：LOAD-FPとSTORE-FPの幅エンコーディング

パックドSIMD計算命令は、f個のレジスタにパックされた値で動作します。

各値は、8ビット、16ビット、32ビット、64ビット、または128ビットのいずれでもかまいません。また、整数と浮動小数点の両方の表現をサポートできます。

例えば、64ビットのパックドSIMD拡張は、各レジスタを1×64ビット、2×32ビット、4×16ビット、または8×8ビットのパックド値として扱うことができます。

-----------------------------------------------------------------------------------------------------------------------------------

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

-- 2018/08/05

第17章

ベクトル操作のための "V"標準拡張、バージョン0.2

この章では、RISC-Vベクトル命令セット拡張の提案を示します。

ベクトル拡張は、構成可能なベクトルユニットをサポートし、アーキテクチャベクタレジスタの数とサポートされている要素の幅を使用可能な最大ベクトル長とトレードオフします。

ベクトル拡張は、同じバイナリコードが、物理的なベクトル記憶容量とデータパスの並列性が異なるさまざまなハードウェア実装で効果的に動作するように設計されています。

-----------------------------------------------------------------------------------------------------------------------------------

ベクトル拡張は、1970年代にSeymour Crayによって導入されたベクトルレジスタアーキテクチャのスタイルに基づいており、以前のパックドSIMDアプローチとは対照的に、1957年にLincoln Labs TX-2で導入され、現在ほとんどの商用命令セットで採用されています。

ベクトル命令セットには、Berkeley T0およびVIRAMベクタ・マイクロプロセッサ、MIT Scaleベクトル・スレッド・プロセッサ、Berkeley MavenおよびHwachaプロジェクトを含む、以前の研究プロジェクトで開発された多くの機能が含まれています。

17.1ベクトル単位の状態

追加のベクトルユニットアーキテクチャ状態は、32個のベクトルデータレジスタ（v0〜v31）、8個のベクトル述部レジスタ（vp0〜vp7）、およびXLENビットのWARLベクトル長CSR、vlで構成されています。

さらに、ベクトルユニットの現在の構成は、以下で説明するように、セットベクトル構成CSRs（vcmaxw、vctype、vcnpred）に保持されます。

実装は、vcmaxwおよびvcnpredレジスタに保持されている現在のコンフィギュレーションの使用可能な最大ベクトル長（MVL）を決定します。

また、3ビット固定小数点丸めモードCSR vxrm、および単一ビット固定小数点飽和ステータスCSR vxsatもあります。

17.2要素のデータ型と幅

V拡張でサポートされているデータ型と操作は、ベーススカラーISAとサポートされている拡張に依存し、8ビット、16ビット、32ビット、64ビット、128ビットの整数と固定小数点データ型16ビット、32ビット、64ビット、および128ビットの浮動小数点タイプ（それぞれF16、F32、F64、およびF128）をサポートしています。

V拡張が追加されるとき、表17.2で定義されているように、サポートされているスカラー型によって暗示されたベクトルデータ要素型をサポートしなければなりません。

サポートされている最大要素幅：

ELEN = max（XLEN; FLEN）

|  |  |  |
| --- | --- | --- |
| CSR 名 | 番号 | 基本ISA |
| vl | 0x020 | RV32, RV64, RV128 |
| vxrm | 0x020 | RV32, RV64, RV128 |
| vxsat | 0x020 | RV32, RV64, RV128 |
| vcsr | 0x020 | RV32, RV64, RV128 |
| vcnpred | 0x020 | RV32, RV64, RV128 |
| vcmaxw | 0x020 | RV32, RV64, RV128 |
| vcmaxw1 | 0x020 | RV32 |
| vcmaxw2 | 0x020 | RV32, RV64 |
| vcmaxw3 | 0x020 | RV32 |
| vctype | 0x020 | RV32, RV64, RV128 |
| vctype1 | 0x020 | RV32 |
| vctype2 | 0x020 | RV32, RV64 |
| vctype3 | 0x020 | RV32 |
| vctypev0 | 0x020 | RV32, RV64, RV128 |
| vctypev1 | 0x020 | RV32, RV64, RV128 |
| ... |  |  |
| vctypev31 | 0x020 | RV32, RV64, RV128 |

表17.1：ベクター拡張CSRs。

|  |  |
| --- | --- |
| サポートされる浮動小数点幅 | |
| RV32I | X8, X16, X32 |
| RV64I | X8, X16, X32, X64 |
| RV128I | X8, X16, X32, X64, X128 |

表17.2：基本整数ISAとサポートされている浮動小数点数の拡張に応じてサポートされるデータ要素の幅。

   指定された浮動小数点幅をサポートすると、すべてのより狭い浮動小数点幅のサポートが要求されることに注意してください。

-----------------------------------------------------------------------------------------------------------------------------------

ハードウェアでサポートされているデータ型がスカラ命令とベクトル命令の両方でサポートされている場合、ベクトル化のコンパイラサポートは大幅に簡素化されます。

浮動小数点サポートを持つマシンにベクトル拡張を追加すると、IEEE標準の半精度16ビット浮動小数点データ型がサポートされます。

これには、セクション「??」で説明されている一連のスカラー半精度命令が含まれます。

スカラー倍精度命令は、他の浮動小数点精度のテンプレートに従いますが、これまで使用されていなかったfmtフィールドのエンコーディング10を使用しています。

-----------------------------------------------------------------------------------------------------------------------------------

ハーフ精度の主な利点は、オペレーションごとの制御オーバーヘッドを償却するベクトル命令を使用する場合に得られるため、ベクトル拡張の一部としてスカラー半精度浮動小数点型のみをサポートします。

別個のスカラー半精度浮動小数点拡張をサポートしていないため、標準命令セットの数が減少します。

17.3ベクトル構成レジスタ（vcmaxw、vctype、vcp）

使用前にベクターユニットを構成する必要があります。

各アーキテクチャベクタデータレジスタ（v0〜v31）は、そのベクタデータレジスタの各要素で許可される最大ビット数で構成されます。または、他のアーキテクチャベクタデータレジスタの物理ベクタ格納を解放するために無効にすることもできます。

利用可能なベクトル述語レジスタの数は、独立して設定することもできます。

利用可能なMVLは構成設定に依存しますが、MVLは常に同じ構成パラメータに対して同じ値を持つ必要があります。

実装では、サポートされているすべての構成設定に対して、少なくとも4つの要素のMVLを提供する必要があります。

各ベクトルデータレジスタの現在の最大幅は、表17.3のようにエンコードされたvcmaxw CSRの別々の4ビットフィールドに保持されます。

|  |  |
| --- | --- |
| 幅 | エンコーディング |
| 無効 | 0000 |
| 8 | 1000 |
| 16 | 1001 |
| 32 | 1010 |
| 64 | 1011 |
| 128 | 1100 |

表17.3：vcmaxwフィールドのエンコーディング それ以外の値はすべて予約されています。

-----------------------------------------------------------------------------------------------------------------------------------

以前のいくつかのベクトルマシンは、より多くの短いベクトルまたはより短い数の長いベクトル、特に富士通VPシリーズ[21]に物理ベクトルレジスタ記憶装置を構成する能力を有していた。

さらに、各ベクトルデータレジスタには、表17.4に示すようにエンコードされたvctype CSRの4ビットフィールドに保持される関連するダイナミックタイプフィールドがあります。

ベクトルデータレジスタのダイナミックタイプフィールドは、そのベクトルデータレジスタの対応するvcmaxwフィールドの値と等しいか小さい幅を持つ型だけを保持するように制約されています。

vctypeを変更してもMVLは変更されません。

-----------------------------------------------------------------------------------------------------------------------------------

ベクタデータレジスタは、ベクトルファンクションコールをサポートするために、要素の最大幅と現在の要素のデータ型の両方を持っています。以下に説明するように、呼び出し元は呼び出し先が必要とする型を知りません。

構成時間を短縮するために、vcmaxwフィールドへの書き込みは、対応するvctypeフィールドも書き込みます。

vcmaxwフィールドには、表17.4のタイプエンコーディングから取得した値を書き込むことができますが、表17.3に示す幅情報のみがvcmaxwフィールドに記録され、完全な型の情報は対応するvctypeフィールドに記録されます。

実装でサポートされている幅を超えるvcmaxwフィールドを書き込もうとすると、不正な命令例外が発生します。

実装では、要求された値より大きなvcmaxw値を記録することができます。

具体的には、実装では、vcmaxwフィールドをサポートされている最大の幅に固定することを選択できます。

|  |  |  |
| --- | --- | --- |
| 型 | vctypeエンコーディング | vcmaxw同等 |
| 無効 | 0000 | 0000 |
| F16 | 0001 | 1001 |
| F32 | 0010 | 1010 |
| F64 | 0011 | 1011 |
| F128 | 0100 | 1100 |
| X8 | 1000 | 1000 |
| X16 | 1001 | 1001 |
| X32 | 1010 | 1010 |
| X64 | 1011 | 1011 |
| X128 | 1100 | 1100 |

表17.4：vctypeフィールドのエンコーディング

3番目の列には、vcmaxwフィールドに書き込むときに保存される値が表示されます。

それ以外の値はすべて予約されています。

要求された値より大きい値。

具体的には、実装では、vcmaxwフィールドをサポートされている最大の幅に固定することを選択できます。

サポートされていない型または現在のvcmaxw幅を超える型をvctypeフィールドに書き込もうとすると、例外が発生します。

vcmaxwレジスタ内のフィールドへの書き込みは、ベクトル単位を構成し、すべてのベクトルデータレジスタをゼロにし、すべてのベクトル述部レジスタを設定し、ベクトル長レジスタvlを最大サポートベクトル長に設定します。

vctypeフィールドへの書き込みは、関連するベクトルデータレジスタのみをゼロにし、他のベクトルユニットの状態はそのままにします。

対応するvcmaxw値よりも多くのビットを必要とする型をvctypeフィールドに書き込もうとすると、不正な命令例外が発生します。

-----------------------------------------------------------------------------------------------------------------------------------

セキュリティホールを防ぎ、異なる実装が物理的なベクトルレジスタ記憶をどのように管理するかの違いを避けるために、ベクトルレジスタは再設定時にゼロにされます。

順序どおりの実装では、おそらく上書きされるまで、レジスタごとにフラグビットを使用して、各ソースのガベージ値ではなく0をmuxします。

インオーダー・マシンでは、予測またはMVL未満のベクトル長による部分書込みはこのゼロ化を複雑にしますが、

これらの場合は、ハードウェアのリード・モディファイ・ライトを採用し、エレメントごとにゼロ・ビットを追加し、

または設定後の最初の書き込みアクセスが部分的な場合は、マシンモードのトラップハンドラへのトラップです。

アウト・オブ・オーダー・マシンは、初期リネーム・テーブルを物理ゼロ・レジスタにポイントすることができます。

RV128では、vcmaxwは32個の4ビット幅のフィールドを保持する単一のCSRです。 ビット（4N + 3）-（4N）は、ベクタデータレジスタNの最大幅を保持します。

RV64では、vcmaxw2 CSRがvcmaxwの上位64ビットにアクセスできます。

RV32では、vcmaxw1 CSRがvcmaxwのビット63-32へのアクセスを提供し、vcmax3 CSRはビット127-96へのアクセスを提供します。

vcnpred CSRには、0〜8の有効なアーキテクチャ述語レジスタの数を示す単一の4ビットWLRLフィールドが含まれています。

vcnpredに書き込むと、すべてのベクタデータレジスタがゼロになり、可視ベクタプレディケートレジスタのすべてのビットがセットされ、ベクタ長レジスタvlがサポートされている最大ベクタ長に設定されます。

vcnpredに8より大きい値を書き込もうとすると、不正な命令例外が発生します。

|  |  |
| --- | --- |
| AVL 値 | vl 設定 |
| ＡＶＬ ≧ 2MVL | MVL |
| 2MVL > AVL > MVL | |AVL/2| |
| MVL ≧ AVL | AVL |

表17.5：要求されたアプリケーションベクトル長（AVL）および現在の最大ベクトル長（MVL）に基づいて、ベクトル長レジスタvlを設定するsetvl命令の操作。

17.4ベクターの長さ

アクティブなベクトルの長さは、XLENビットのWARLベクトルの長さCSR vlに保持されます。これは、0〜MVLの値を保持するだけです。

最大コンフィギュレーションレジスタ（vcmaxwまたはvcnpred）への書き込みは、vlをMVLで初期化します。

vctypeへの書き込みはvlに影響しません。

アクティブなベクトルの長さは通常、setvl命令で書き込まれます。setvl命令は、vl CSR番号へのcsrrw命令としてエンコードされます。

csrrwのsource引数は、要求されたアプリケーションのベクトル長（AVL）を符号なしのXLENビット整数として指定します。

setvl命令は、表17.5に従ってvlに割り当てる値を計算します。

-----------------------------------------------------------------------------------------------------------------------------------

vlレジスタを設定するルールは、ストリップミニループの最後の2回の反復にわたってベクトルパイプラインをいっぱいにします。

クレイ設計のマシンでも同様のルールが使用されていました[7]。

この計算の結果は、setvl命令の結果としても返されます。

通常のcsrrw命令とは異なり、整数レジスタrdに書き込まれる値は元のCSR値ではなく、変更された値であることに注意してください。

-----------------------------------------------------------------------------------------------------------------------------------

インプリメンテーションで定義されたベクトルの長さは、少なくともストリップラインループを制御するための特別な "ロードベクトルカウントと更新"（VLVCU）命令を使用したIBM 3090 Vector Facility [5]に帰着しました。

ここに含まれるsetvl命令は、Asanovic [4]によって導入されたより単純なsetvlr命令に基づいています。

setvl命令は、典型的には、次のループ反復で動作するベクトル要素の数を設定するために、ストリップミニループの各反復の開始時に使用されます。

現在のMVLは、すべてのビットが設定された（最大の符号なし整数）ソース引数を使用してsetvlを実行することによって取得できます。

vl = 0のとき、どのベクトル命令に対しても要素演算は実行されません。

17.5ラピッドコンフェッション命令

vcmaxw、vctype、およびvcnpredを特定の構成に設定するには、いくつかの指示が必要です。

ベクトルユニットの設定を高速化するために、vcmaxw、vctype、およびvncpred構成レジスタに複数のフィールドを設定したエンコードされた即値を使用して、CSRへの書き込みとしてエンコードされた特殊なvcfg命令が追加されます。

vcfgd命令は、図17.2に示すようにエンコードされたレジスタ値を受け取り、対応するMVLをデスティネーションレジスタに返すCSRRWとしてエンコードされます。

対応するvcfgdi命令は、コンフィグレーションを設定するために5ビットの即値を取り、宛先レジスタにMVLを戻すCSRRWIとしてエンコードされます。

＃ベクトル - ベクトル32ビット加算ループ。

＃正しいタイプで構成されたベクトルユニットを仮定します。

＃a0はNを保持するa1は結果ベクトルへのポインタを保持する

＃a2は最初のソースベクトルへのポインタを保持する

＃a3は、第2のソースベクトルへのポインタを保持する。

ループ：setvl t0、a0

        vld v0、a2 ＃最初のベクトルをロードする

        sll t1、t0、2 ＃バイトを掛ける

        a2、t1 ＃バンプポインタを追加

        vld v1、a3 ＃第2のベクトルをロードする

        a3、t1 ＃バンプポインタを追加

        vadd v0、v1 ＃要素を追加する

        sub a0、t0 ＃デクリメント要素が完了しました

        vst v0、a1 ＃結果ベクトルを格納する

        add a1、t1 ＃バンプポインタ

        bnez a0、loop ＃これ以上は？

図17.1：ベクトルベクトル加算ループの例。

-----------------------------------------------------------------------------------------------------------------------------------

vcfgdiの主な用途の1つは、memcpyルーチンとmemsetルーチンで使用する1バイトの要素ベクトルでベクトルユニットを構成することです。

単一の命令で、これらの操作のためにベクトルユニットを構成することができます。

また、vcfgd命令はvcnpredレジスタもクリアするので、述語レジスタは割り当てられません。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | F64 | F32 | F16 | X32 | X16 | X8 | RV32 |

2 5 5 5 5 5 5

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | F128 | X64 | F64 | F32 | F16 | X32 | X16 | X8 | RV64 |

24 5 5 5 5 5 5 5 5

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | X128 | F128 | X64 | F64 | F32 | F16 | X32 | X16 | X8 | RV64 |

83 5 5 5 5 5 5 5 5 5

図17.2：サポートされている各タイプの5ビットベクタレジスタ番号を保持する異なるベースISAのvcfgd値のフォーマット。

フィールドには、そのタイプにベクタ・レジスタが割り当てられていないことを示す0、またはすべてが右より大きいベクトル・レジスタ番号が含まれている必要があります。

2つの非ゼロフィールドの間のすべてのベクトルレジスタ番号は、より高いベクトルレジスタ番号を有するタイプに割り当てられます。

vcfgd値は、各データ型のベクトルレジスタの数を指定し、サポートされるデータ型ごとに1つずつ、5ビットのフィールドに分割されます。

フィールドの値0は、そのタイプのレジスタが割り当てられていないことを示します。

ゼロ以外の値は、最も高いベクトルを示します。

vcfgd値の各5ビットフィールドは、ベクタレジスタがその型に割り当てられていないことを示すゼロか、関連する型を含む最上位のベクトルレジスタを示す下位ビット位置のすべてのフィールドより大きいベクトルレジスタ番号を含む必要があります。

このエンコーディングは、最も狭い必要タイプに割り当てられたベクトルレジスタ（v0およびv1）が少なくとも2つ必要であることを除けば、ベクトルレジスタのデータタイプへの任意の割り当てをコンパクトに表すことができます。

割り当ての例を図17.3に示します。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | F64 | F32 | F16 | X32 | X16 | X8 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 18 | 12 | 0 | 1 | 0 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| ベクタ レジスタ | vcmaxw | vctype | 型 |
| v31-v19 | 0000 | 0000 | 無効 |
| v18-v13 | 1011 | 0011 | F64 |
| v12-v2 | 1010 | 0010 | F32 |
| v1-v0 | 1010 | 1010 | X32 |

図17.3：構成を設定するためのvcfgd値の使用例

CSRRWエンコーディングとCSRRWIエンコーディングをそれぞれ使用して、vcnpredレジスタにソース値を書き込み、新しいMVLを返す別々のvcfgp命令とvcfgpi命令が用意されています。

これらの書込みは、ベクタデータレジスタもクリアし、割り当てられた述語レジスタのすべてのビットをセットし、vl = MVLにセットします。

vcfgdの後にvcfgpまたはvcfgpi命令を使用して、ベクトルユニットのリコンフィギュレーションを完了することができます。

vcgfdにゼロ引数が指定されている場合、有効なレジスタなしでベクトルユニットがコンフィグレーションされ、MVLの場合は0が返されます。

この状態では、構成レジスタvcmaxwおよびvcnpredにのみ、直接またはvcfgd、vcfgdi、vcfgp、またはvcfgpi命令を介してアクセスできます。

他のベクトル命令では、不正な命令例外が発生します。

ベクトルレジスタの個々のタイプをすばやく変更するために、各ベクトルデータレジスタnにはvctypevnというvctypeフィールドにアクセスするための専用のCSRアドレスがあります。

vcfgtおよびvcfgti命令は、タイプフィールドを更新して元の値を返す通常のCSRRWおよびCSRRWI命令のためのアセンブラ疑似命令です。

vcfgti命令は、通常、1つの命令で前のタイプを記録している間に、所望のタイプに変更するために使用され、vcfgt命令は、保存されたタイプに戻すために使用されます。

stripmined loop stripmined って辞書引くと、「除去によって地表の近くで採鉱される」。 除去によって地表の近くで採鉱されるループってよくわからない。

predicate registers そのまま訳すと 述語レジスタとか述部レジスタなんだけどなにか良い訳はないか。

ちょっとこの ベクター操作の章 は難しいな。

-- 1ページ空き、次ページへ

第18章

ユーザレベル割り込みのための "N"標準拡張、バージョン1.1

-----------------------------------------------------------------------------------------------------------------------------------

これはNエクステンションのより完全な執筆のためのプレースホルダーであり、議論の基礎を形成するものです。

この章では、RISC-Vユーザーレベルの割り込みと例外処理を追加するための提案を示します。

N拡張が存在し、外部実行環境が指定された割り込みおよび例外をユーザレベルに委譲した場合、ハードウェアは外部実行環境を呼び出さずに制御をユーザレベルトラップハンドラに直接転送することができます。

-----------------------------------------------------------------------------------------------------------------------------------

ユーザーレベルの割り込みは主に、MモードとUモードのみのセキュアな組み込みシステムをサポートすることを目的としていますが、ユーザーレベルのトラップ処理をサポートするUnixライクなオペレーティングシステムを実行するシステムでもサポートできます。

Unix環境で使用する場合、ユーザーレベルの割り込みは従来の信号処理を置き換えることはできませんが、ガベージコレクションの障壁、整数オーバーフロー、浮動小数点トラップなどのユーザーレベルのイベントを生成するさらなる拡張のビルディングブロックとして使用できます 。

18.1追加のCSRs

Nエクステンションをサポートするために追加されたユーザーが表示するCSRsを表18.1に示します。

|  |  |  |
| --- | --- | --- |
| 番号 | 名 | 説明 |
| 0x000 | ustatus | ユーザ状態レジスタ。 |
| 0x004 | uie | ユーザ割り込みイネーブルレジスタ。 |
| 0x005 | utvec | ユーザトラップハンドラのベースアドレス。 |
| 0x040 | uscratch | ユーザトラップハンドラのためのスクラッチレジスタ。 |
| 0x041 | uepc | ユーザー例外プログラムカウンタ。 |
| 0x042 | ucause | ユーザトラップの原因。 |
| 0x043 | utval | ユーザーのアドレスまたは命令が正しくありません。 |
| 0x044 | uip | ユーザー割り込みの保留中です。 |

表18.1：N拡張のCSRs

18.2ユーザーステータスレジスタ（ustatus）

ustatusレジスタは、図18.1のようにフォーマットされたXLENビットのリード/ライト・レジスタです。

ustatusレジスタは、ハートの現在の動作状態を追跡し、制御します。

XLEN 5 4 3 1 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WPRI | UPIE | WPRI | UIE |  |

XLEN-5 1 3 1

図18.1：ユーザモードステータスレジスタ（ustatus）。

ユーザー割込み許可ビットUIEは、クリア時にユーザーレベルの割込みをディスエーブルします。

UIEの値は、ユーザーレベルのトラップが取られたときにUPIEにコピーされ、UIEの値はゼロに設定されて、ユーザーレベルのトラップハンドラのアトミック性を提供します。

-----------------------------------------------------------------------------------------------------------------------------------

以前の特権モードを保持するためのUPPビットはありません。これは、ユーザーモードのみであるためです。

URET命令はUモードのトラップから戻るために使用され、URETはUPIEをUIEにコピーしてからUPIEを設定します。

-----------------------------------------------------------------------------------------------------------------------------------

UPIEは、UPIE / UIEスタックがポップされた後に設定され、割り込みを有効にし、コーディングエラーを捕捉するのに役立ちます。

18.3その他のCSR

残りのCSRは、MモードとSモード用に定義されたトラップ処理レジスタと同様に機能します。

-----------------------------------------------------------------------------------------------------------------------------------

より完全な執筆を続けてください。 (もっと詳しく書いてください。)

18.4 N拡張命令

URET命令は、MRETおよびSRETに類似の機能を実行するために追加されています。

18.5コンテキストスワップオーバーヘッドの削減

ユーザーレベルの割り込み処理レジスタは、ユーザーレベルのコンテキストにかなりの状態を追加しますが、通常の使用ではほとんどアクティブではありません。

特に、uepc、ucause、およびutvalは、トラップハンドラの実行中のみ有効です。

FSフィールドとXSフィールドのフォーマットに従って、mstatusとsstatusにNSフィールドを追加して、値が公開されていないときにコンテキストスイッチオーバーヘッドを減らすことができます。

URETを実行すると、uepc、ucause、およびutvalが初期状態に戻ります。

– 2018/08/10

第19章

RV32 / 64G命令セットリスト

RISC-Vプロジェクトの1つの目標は、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す。

この目的のために、基本ISA（RV32IまたはRV64I）と選択された標準拡張（IMAFD）の組み合わせを「汎用ISA」として定義し、命令セット拡張のIMAFDの組み合わせに略語Gを使用します。

この章では、オペコードマップとRV32GおよびRV64Gの命令セットリストを示します。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Inst[4:2] | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| Inst[6:5] |  |  |  |  |  |  |  | (＞ 32b) |
| 00 | LOAD | LOAD-FP | custom-0 | MISC-MEM | OP-IMM | AUIPC | OP-IMM-32 | 48b |
| 01 | STORE | STORE-FP | custom-1 | AMO | OP | LUI | OP-32 | 64b |
| 10 | MADD | MSUB | NMSUB | NMADD | OP-FP | reserved | custom-2/rv128 | 48b |
| 11 | BRANCH | JALR | reserved | JAL | SYSTEM | reserved | custom-3/rv128 | ≧ 80b |

表19.1：RISC-Vベースオペコードマップ、inst [1：0] = 11

表19.1にRVGの主要なオペコードのマップを示します。

3ビット以上の下位ビットがセットされた主要なオペコードは、32ビットを超える命令長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カスタム命令セット拡張のために避けるべきです。

カスタム-0およびカスタム-1とマークされた主要なオペコードは、将来の標準拡張によって回避され、ベースの32ビット命令フォーマット内のカスタム命令セット拡張によって使用することが推奨されます。

カスタム-2 / rv128およびカスタム-3 / rv128とマークされたオペコードは、将来のRV128での使用のために予約されていますが、標準の拡張では避けられるため、RV32およびRV64のカスタム命令セット拡張にも使用できます。

我々はRV32GとRV64Gが広範な汎用コンピューティングのための単純で完全な命令セットを提供すると信じています。

追加のハードウェアの複雑さを伴いますが、パフォーマンス、コードサイズ、およびエネルギー効率を向上させるために、第12章で説明したオプションの圧縮命令セットを追加できます（RV32GCおよびRV64GCの作成）。

IMAFDCを超えてさらに命令セット拡張に移行するにつれて、追加された命令はよりドメイン特有である傾向があり、例えばマルチメディアやセキュリティのような制限されたクラスのアプリケーションにのみ利益をもたらします。

ほとんどの商用ISAとは異なり、RISC-V ISA設計は、基本ISAと広く適用可能な標準拡張を、これらのより特殊な追加から明確に分離しています。

第21章では、RISC-V ISAに拡張機能を追加する方法について、より広範な議論を行っています。

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct7 | rs2 | rs1 | funct3 | rd | opcode | R-型 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | opcode | I-型 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| imm[11:5] | rs2 | rs1 | funct3 | Imm[4:0] | opcode | S-型 |

|  |  |  |  |
| --- | --- | --- | --- |
| imm[31:12] | rd | opcode | U-型 |

|  |  |  |  |
| --- | --- | --- | --- |
| imm[20|10:1|11|19:12] | rd | opcode | J-型 |

RV32Iベース命令セット

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| 0000 | pred | | succ | rs1 | 000 | 00000 | 0001111 | FENCE |
| 0000 | 0000 | | 0000 | rs1 | 001 | 00000 | 0001111 | FENCE.I |
| 000000000000 | | | | rs1 | 000 | 00000 | 1110011 | ECALL |
| 000000000001 | | | | rs1 | 000 | 00000 | 1110011 | EBREAK |
| csr | | | | rs1 | 001 | rd | 1110011 | CSRRW |
| csr | | | | rs1 | 010 | rd | 1110011 | CSRRS |
| csr | | | | rs1 | 011 | rd | 1110011 | CSRRC |
| csr | | | | zimm | 101 | rd | 1110011 | CSRRWI |
| csr | | | | zimm | 110 | rd | 1110011 | CSRRSI |
| csr | | | | zimm | 111 | rd | 1110011 | CSRRCI |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct7 | rs2 | rs1 | funct3 | rd | opcode | R-型 |
| imm[11:0] | | rs1 | funct3 | rd | opcode | I-型 |
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opccode | S-型 |

RV64Iベース命令セット（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 |

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）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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 |

RV32A標準拡張

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 00010 | aq | r1 | 00000 | rs1 | 010 | rd | 0101111 | LR.W |
| 00011 | aq | r1 | rs2 | rs1 | 010 | rd | 0101111 | SC.W |
| 00001 | aq | r1 | rs2 | rs1 | 010 | rd | 0101111 | AMOSWAP.W |
| 00000 | aq | r1 | rs2 | rs1 | 010 | rd | 0101111 | AMOADD.W |
| 00100 | aq | r1 | rs2 | rs1 | 010 | rd | 0101111 | AMOXOR.W |
| 01100 | aq | r1 | rs2 | rs1 | 010 | rd | 0101111 | AMOAND.W |
| 01000 | aq | r1 | rs2 | rs1 | 010 | rd | 0101111 | AMOOR.W |
| 10000 | aq | r1 | rs2 | rs1 | 010 | rd | 0101111 | AMOMIN.W |
| 10100 | aq | r1 | rs2 | rs1 | 010 | rd | 0101111 | AMOMAX.W |
| 11000 | aq | r1 | rs2 | rs1 | 010 | rd | 0101111 | AMOMINU.W |
| 11100 | aq | r1 | rs2 | rs1 | 010 | rd | 0101111 | AMOMAXU.W |

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct7 | | rs2 | rs1 | funct3 | rd | opcode | R-型 |
| fs3 | funct2 | r2 | rs1 | funct3 | rd | opcode | R4-型 |
| imm[11:0] | | | rs1 | funct3 | rd | opcode | I-型 |
| imm[11:5] | | rs2 | rs1 | funct3 | imm[4:0] | opccode | S-型 |

RV64A標準拡張（RV32Aに加えて）

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 00010 | aq | r1 | rs2 | rs1 | 011 | rd | 0101111 | LR.D |
| 00011 | aq | r1 | rs2 | rs1 | 011 | rd | 0101111 | SC.D |
| 00001 | aq | r1 | rs2 | rs1 | 011 | rd | 0101111 | AMOSWAP.D |
| 00000 | aq | r1 | rs2 | rs1 | 011 | rd | 0101111 | AMOADD.D |
| 00100 | aq | r1 | rs2 | rs1 | 011 | rd | 0101111 | AMOXOR.D |
| 01100 | aq | r1 | rs2 | rs1 | 011 | rd | 0101111 | AMOAND.D |
| 01000 | aq | r1 | rs2 | rs1 | 011 | rd | 0101111 | AMOOR.D |
| 10000 | aq | r1 | rs2 | rs1 | 011 | rd | 0101111 | AMOMIN.D |
| 10100 | aq | r1 | rs2 | rs1 | 011 | rd | 0101111 | AMOMAX.D |
| 11000 | aq | r1 | rs2 | rs1 | 011 | rd | 0101111 | AMOMINU.D |
| 11100 | aq | r1 | rs2 | rs1 | 011 | rd | 0101111 | AMOMAXU.D |

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 |
| 0010100 | | rs2 | rs1 | 010 | rd | 1010011 | FSGNJX.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 |

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| funct7 | | rs2 | rs1 | funct3 | rd | opcode | R-型 |
| fs3 | funct2 | r2 | rs1 | funct3 | rd | opcode | R4-型 |
| imm[11:0] | | | rs1 | funct3 | rd | opcode | I-型 |
| imm[11:5] | | rs2 | rs1 | funct3 | imm[4:0] | opccode | S-型 |

RV64F標準拡張（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 |

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 |
| 0010101 | | rs2 | rs1 | 010 | rd | 1010011 | FSGNJX.D |
| 0010101 | | rs2 | rs1 | 000 | rd | 1010011 | FMIN.D |
| 0010101 | | rs2 | rs1 | 001 | rd | 1010011 | FMAX.D |
| 0100000 | | 00000 | rs1 | rm | rd | 1010011 | FCVT.S.D |
| 0100001 | | 00001 | 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 |

表19.2：RISC-Vの命令リスト

表19.3に、現在CSRsアドレスが割り当てられているCSRsを示します。

タイマー、カウンタ、および浮動小数点CSRsは、この仕様で定義されている唯一のCSRsです。

|  |  |  |  |
| --- | --- | --- | --- |
| 番号 | 特権 | 名 | 説明 |
| 浮動小数点コントロールとステータスレジスタ | | | |
| 0x001 | 読み／書き | fflags | 浮動小数点発生例外。 |
| 0x002 | 読み／書き | frm | 浮動小数点動的丸めモード |
| 0x003 | 読み／書き | 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 | 読み限定 | cycle | RDCYCLE命令のサイクルカウンタ。 |
| 0xC01 | 読み限定 | time | RDTIME命令のタイマ。 |
| 0xC02 | 読み限定 | instret | RDINSTRET命令の命令リタイヤカウンタ。 |
| 0xC80 | 読み限定 | cycleh | cycleの上位32ビット、RV32Iのみ。 |
| 0xC81 | 読み限定 | timeh | timeの上位32ビット、RV32Iのみ。 |
| 0xC82 | 読み限定 | insreth | instretの上位32ビット、RV32Iのみ。 |

表19.3：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20章

RISC-Vアセンブリプログラマーズハンドブック

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20.1に、xとfレジスタのアセンブラニーモニックと標準呼び出し規約の役割を示します。

|  |  |  |  |
| --- | --- | --- | --- |
| レジスタ | ABI 名 | 説明 | セーバー |
| x0 | zero | ハードワイヤードゼロ | ー |
| x1 | ra | リターン 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 | 広域 ポインタ | ー |
| x4 | tp | スレッドポインタ | ー |
| x5 | t0 | 一時的/代替リンク・レジスタ | 呼び出し元 |
| x6-7 | t1-2 | 一時変数 | 呼び出し元 |
| x8 | s0/fp | 保存レジスタ/フレームポインタ | 呼び出し先 |
| x9 | s1 | 保存レジスタ | 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 | a2-11 | 保存レジスタ | 呼び出し先 |
| x28-31 | t3-6 | 一時変数 | 呼び出し元 |
| f0-7 | ft0-7 | FP 一時変数 | 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 一時変数 | 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 一時変数 | 呼び出し元 |

表20.1：RISC-V整数と浮動小数点レジスタのアセンブラニーモニック。

表20.2と表20.3に、標準のRISC-V疑似命令のリストを示します。

|  |  |  |
| --- | --- | --- |
| 疑似命令 | 基本命令(s) | 意味 |
| la rd, symbol | auipc rd, symbol[31:12]  addi rd, rd, symbol[11:0] | ロードアドレス |
| l{b|h|w|d} rd, symbol | auipc rd, symbol[31:12]  l{b|h|w|d} rd, symbol[11:0](rd) | ロード広域 |
| s{b|h|w|d} rd, symbol, rt | auipc rt, symbol[31:12]  s{b|h|w|d} rd, symbol[11:0](rt) | ストア高域 |
| fl{w|d} rd, symbol, rt | auipc rt, symbol[31:12]  flfw|dg rd, symbol[11:0](rt) | 浮動小数点ロード広域 |
| fs{w|d} rd, symbol, rt | auipc rt, symbol[31:12]  fsfw|dg rd, symbol[11:0](rt) | 浮動小数点ストア広域 |
| nop | addi x0, x0, 0 | 操作なし |
| li rd, immediate | 無数のシーケンス | ロード即値 |
| mv rd, rs | addi rd, rs, 0 | コピーレジスタ |
| 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 | sub 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, r | 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 | ＜ セロの時セット |
| sgtz rd, rs | slt rd, rs, x0 | ＞ ゼロの時セット |
| fmv.s rd, rs | fsgnj.s rd, rs, rs | 単精度レジスタのコピー |
| fabs.s rd, rs | fsgnjx.s rd, rs, rs | 単精度絶対値 |
| fneg.s rd, rs | fsgnjn.s rd, rs, rs | 単精度否定 |
| fmv.d rd, rs | fsgnj.d rd, rs, rs | 倍精度レジスタのコピー |
| 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 |
| j offset | jal x0, offset | ジャンプ |
| jal offset | jal x1, offset | ジャンプとリンク |
| jr rs | jalr x0, rs, 0 | ジャンプ レジスタ |
| jalr rs | jalr x1, rs, 0 | ジャンプとリンクレジスタ |
| ret | jalr x0, x1, 0 | サブルーチンから復帰 |
| call offset | auipc x6, offset[31:12]  jalr x1, x6, offset[11:0] | 遠く離れたサブルーチンを呼び出す |
| tail offset | auipc x6, offset[31:12]  jalr x0, x6, offset[11:0] | テールコール遠く離れたサブルーチン |
| fence | fence iorw, iorw | すべてのメモリとI / O上のフェンス |

表20.2：RISC-V疑似命令

|  |  |  |
| --- | --- | --- |
| 疑似命令 | 基本命令(s) | 意味 |
| rdinstret[h] rd | csrrs rd, instret[h], x0 | 退職カウンターを読む |
| rdcycle[h] rd | csrrs rd, cycle[h], x0 | サイクルカウンターを読む |
| rdtime[h] rd | csrrs rd, time[h], x0 | リアルタイムクロックを読む |
| csrr rd, csr | csrrs rd, csr, x0 | CSRを読む |
| csrw csr, rs | csrrw x0, csr, rs | CSRに書く |
| csrs csr, rs | csrrs x0, csr, rs | CSRのビットをセット |
| csrc csr, rs | csrrc x0, csr, rs | CSRのビットをクリア |
| csrwi csr, imm | csrrwi x0, csr, imm | CSRに即値を書く |
| csrsi csr, imm | csrrsi x0, csr, imm | CSRに即値でビットをセット |
| csrci csr, imm | csrrci x0, csr, imm | CSRに即値でビットをクリア |
| frcsr rd | csrrs rd, fcsr, x0 | FP 制御/状態レジスタを読む |
| fscsr rd, rs | csrrw rd, fcsr, rs | FP 制御/状態レジスタを交換 |
| fscsr rs | csrrw x0, fcsr, rs | FP 制御/状態レジスタに書く |
| frrm rd | csrrs rd, frm, x0 | FP丸めモードを読む |
| fsrm rd, rs | csrrw rd, frm, rs | FP 丸めモードを交換 |
| fsrm rs | csrrw x0, frm, rs | FP 丸めモードに書く |
| fsrmi rd, imm | csrrwi rd, frm, imm | FP 即値で丸めモードを交換 |
| fsrmi imm | csrrwi x0, frm, imm | FP 即値で丸めモードに書く |
| frflags rd | csrrs rd, fflags, x0 | FP 例外フラグを読む |
| fsflags rd, rs | csrrw rd, fflags, rs | FP 例外フラグを交換 |
| fsflags rs | csrrw x0, fflags, rs | FP 例外フラグに書く |
| fsflagsi rd, imm | csrrwi rd, fflags, imm | FP 即値で例外フラグを交換 |
| fsflagsi imm | csrrwi x0, fflags, imm | FP 即値で例外フラグに書く |

表20.3：コントロールとステータスレジスタにアクセスするための疑似命令

-- 1ページ空き、次ページへ

第21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ことに加えて、RISCVのもう1つの目標は、より専門化された命令セット拡張またはよりカスタマイズされたアクセラレータの基礎を提供することです。

命令エンコーディング空間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ISAツールチェーンのソフトウェア開発作業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実装に対して、おそらく多くの非標準の命令セット拡張とともに、完全なソフトウェアサポートを引き続き提供することが目的です。

この章では、独立したグループによって開発された命令セット拡張を管理するスキームとともに、ベースRISC-V ISAを拡張するさまざまな方法について説明します。

このボリュームは、ユーザーレベルのISAのみを扱いますが、2番目のボリュームで説明されているスーパーバイザーレベルの拡張にも同じアプローチと用語が使用されています。

21.1拡張用語

このセクションでは、RISC-V拡張を記述するための標準用語を定義します。

標準と非標準の拡張

RISC-Vプロセッサの実装では、基本整数ISA（RV32IまたはRV64I）をサポートする必要があります。

さらに、実装は、1つ以上の拡張をサポートすることができます。

私たちは、拡張を2つの広いカテゴリーに分けます：

標準対非標準。

・標準的な拡張は、一般に有用で、他の標準拡張と競合しないように設計されています。

現在、このマニュアルの他の章で説明されている”MAFDQLCBTPV”は、完全または計画的な標準拡張です。

・非標準拡張は高度に特殊化されている可能性があり、他の標準または非標準拡張と競合する可能性があります。

私たちは、さまざまな非標準的な拡張が時間の経過とともに開発され、最終的には標準拡張に拡張されるものもあると

予想しています。

エンコード空間と接頭辞

命令符号化空間は、基本ISAまたはISA拡張が符号化されるいくつかの命令ビット数です。

RISC-Vはさまざまな命令長をサポートしますが、単一の命令長でもさまざまなサイズのエンコード空間が利用できます。

例えば、ベースISAは30ビットの符号化空間（32ビット命令のビット31-2）内に定義され、原子拡張子「A」は25ビットの符号化空間内にあります（ビット31-7）。

プリフィックスという用語は、命令エンコーディング空間の右側のビットを参照するために使用されます。（RISC-Vはリトルエンディアンであるため、右のビットはより早い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ます）。

標準ベースISAエンコーディングの接頭辞は、32ビットワードのビット1〜0に保持される2ビットの「11」フィールドであり、

標準的な原子拡張「A」のプレフィックスは、AMOメジャーオペコードを表す32ビットワードのビット6-0に保持される7ビットの「0101111」フィールドであり、符号化フォーマットの特徴は、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フィールドが、32ビット命令フォーマットのメジャーオペコードビットと連続していないが、22ビット命令空間のプレフィックスの一部と考えられることです。

命令エンコーディング空間は任意のサイズにすることができますが、より小さい共通サイズのセットを採用することで、独自に開発した拡張機能を単一のグローバルエンコーディングに簡単にまとめることができます。

RISC-Vの推奨サイズを表21.1に示します。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| サイズ | 使用法 | ＃ 標準的な命令長で利用可能 | | | |
| 16-bit | 32-bit | 48-bit | 64-bit |
| 14-bit | 圧縮16ビット符号化の象限 | 3 |  |  |  |
| 22-bit | ベース32ビットエンコーディングのマイナーオペコード |  | 28 | 220 | 235 |
| 25-bit | ベース32ビットエンコーディングにおける主要オペコード |  | 32 | 217 | 232 |
| 30-bit | ベース32ビットエンコーディングの象限 |  | 1 | 212 | 227 |
| 32-bit | 48ビットエンコーディングのマイナーオペコード |  |  | 210 | 225 |
| 37-bit | 48ビットエンコーディングの主なオペコード |  |  | 32 | 220 |
| 40-bit | 48ビットエンコーディングの象限 |  |  | 4 | 217 |
| 45-bit | 64ビットエンコーディングのサブマイナーオペコード |  |  |  | 212 |
| 48-bit | 64ビットエンコーディングのマイナーオペコード |  |  |  | 29 |
| 52-bit | 64ビットエンコーディングの主要オペコード |  |  |  | 32 |

表21.1：推奨される標準RISC-V命令の符号化空間サイズ。

グリーンフィールドとブラウンフィールドエクステンション

グリーンフィールドとブラウンフィールドの拡張

我々は、新しい命令エンコーディング空間を占有し始める拡張を記述するためにグリーンフィールド拡張という用語を使用し、したがって、プレフィックスレベルでのみ符号化の競合を引き起こす可能性があります。

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空間内の既存の符号化に適合する拡張を記述します。

ブラウンフィールド拡張は、特定のグリーンフィールドの親エンコーディングに必ず結びついており、同じグリーンフィールドの親エンコーディングへの複数のブラウンフィールドエクステンションが存在する場合があります。

たとえば、基本ISAsは30ビット命令空間のグリーンフィールド符号化であり、FDQ浮動小数点拡張はすべて親ベースISA 30ビット符号化空間に追加するブラウンフィールド拡張です。

標準のA拡張が、完全な32ビット基本命令エンコーディングの左端のビットに新たに空の25ビットのエンコーディング空間を定義するときに、緑のフィールドエンコーディングを持つとみなしていることに、その標準的な接頭語がベースISAの30ビットの符号化空間内にそれを置くとしても、注意してください。

単一の7ビットプレフィックスだけを変更すると、A拡張を別の30ビットエンコーディング空間に移動することができますが、エンコーディング空間自体ではなく、プレフィックスレベルでの競合を心配する必要があります。

|  |  |  |
| --- | --- | --- |
|  | 状態を追加します | 新しい状態はありません |
| 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21.2：標準的な命令セット拡張の2次元の特徴付け。

表21.2は、単純な2次元タクソノミーに置かれた基底と標準拡張を示します。

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どうかであり、もう1つの軸は、拡張がアーキテクチャの状態を追加するかどうかです。

グリーンフィールド拡張の場合、命令符号化空間のサイズはカッコで示されています。

ブラウンフィールド拡張の場合、それが作成する拡張子（グリーンフィールドまたはブラウンフィールド）の名前はカッコで示されています。

追加のユーザレベルのアーキテクチャ状態は、通常、スーパーバイザレベルのシステムへの変更、または場合によって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、完全な別の基本エンコーディングと見なされることに注意してください。

標準互換のグローバルエンコーディング

実際のRISC-V実装のISAを完全またはグローバルにエンコードするには、含まれているすべての命令エンコーディング空間に一意の競合しないプレフィックスを割り当てる必要があります。

ベースとすべての標準エクステンションには、グローバルエンコーディングで共存できるように標準プレフィックスが割り当てられています。

標準と互換性のあるグローバルなエンコーディングは、ベースと含まれるすべての標準エクステンションに標準プレフィックスがあるものです。

標準と互換性のあるグローバルなエンコーディングには、含まれている標準拡張と競合しない非標準の拡張を含めることができます。

関連する標準拡張がグローバル・エンコーディングに含まれていない場合、標準互換グローバル・エンコーディングは、非標準拡張に標準接頭辞を使用することもできます。

言い換えれば、標準拡張は、標準互換グローバル符号化に含まれていればその標準プレフィックスを使用しなければならないが、そうでなければそのプレフィックスは自由に再割り当てできます。

これらの制約により、共通ツールチェーンは、RISC-V標準互換の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。

保証された非標準のエンコーディング空間

独自のカスタム拡張の開発をサポートするために、エンコーディング空間の一部は、標準の拡張では使用されないことが保証されています。

21.2 RISC-V拡張設計の考え方

私たちは、独立して開発された多数の拡張をサポートするつもりです。これは、拡張エンティティ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ることによって、これらを標準互換のグローバルエンコーディングにパックするツールを提供することです。

一部の拡張機能は、既存の拡張機能の茶色のフィールド拡張としてより自然に実装され、親の緑色のフィールド拡張に割り当てられるプレフィックスを共有します。

標準エクステンションプレフィックスは、コア機能のエンコーディングにおいて偽の非互換性を避け、さらに難解な拡張のカスタムパッキングを可能にします。

RISC-V拡張を異なる標準互換のグローバル符号化に再パックするこの機能は、さまざまな方法で使用できます。

1つのユースケースは、重要なアプリケーションドメインからカーネルを実行するように設計された高度に特殊化されたカスタムアクセラレータを開発することです。

これらは基本整数ISA以外のすべてを削除し、手元にあるタスクに必要な拡張機能だけを追加することができます。

ベースISAは、ハードウェアの実装に最小限の要件を課すように設計されており、32ビットの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。

別の使用事例は、新しいタイプの命令セット拡張のための研究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努力を費やしたくないかもしれないので、単純な32ビットの固定幅の命令エンコーディングを使ってその拡張を試作したいと考えています。

ただし、この新しい拡張機能は、32ビットスペースの標準拡張と共存するには大きすぎます。

研究の実験で標準拡張のすべてが必要ない場合、標準互換のグローバルコードは、未使用の標準拡張を削除し、そのプレフィックスを再利用して、提案された拡張を非標準の場所に置いて研究プロトタイプの設計を簡単に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存在するベースおよび任意の標準拡張をターゲットにすることができます。

命令セット拡張が評価され、洗練されれば、より大きな可変長符号化空間にパックすることができ、すべての標準との衝突を避けることができます。

以下のセクションでは、新しい命令セット拡張を使用して実装を開発するためのますます洗練された戦略について説明します。

これらは主に、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、教育的な、または実験的なアーキテクチャでの使用を目的としています。

21.3固定幅の32ビット命令フォーマット内の拡張

このセクションでは、基本固定幅32ビット命令フォーマットのみをサポートする実装に拡張機能を追加する方法について説明します。

-----------------------------------------------------------------------------------------------------------------------------------

最も単純な固定幅の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プロトタイプで一般的に使用されることが予想されます。

使用可能な30ビット命令エンコーディング空間

標準的な符号化では、利用可能な30ビットの命令符号化空間（2ビットの接頭辞00,01、および10を有するもの）のうちの3つが、オプションの圧縮命令拡張を可能にするために使用されまず。

しかし、圧縮された命令セット拡張が必要でない場合、これら3つのさらなる30ビット符号化空間が利用可能になります。

これにより、利用可能なエンコーディング空間が32ビット形式で四倍になります。

利用可能な25ビット命令コード空間

25ビットの命令符号化空間は、基本および標準の拡張符号化における主要なオペコードに対応しています。

カスタムエクステンション用に明示的に予約された4つの主要なオペレーションコード（表19.1）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ます。

これらのうちの2つは、RV128ベースのエンコーディング（OP-IMM-64とOP-64）で使用するために予約されていますが、RV32とRV64の標準または非標準の拡張に使用できます。

RV64用に予約された2つのオペコード（OP-IMM-32およびOP-32）は、RV32への標準および非標準の拡張にのみ使用できます。

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（LOAD-FP、STORE-FP、MADD、MSUB、NMSUB、NMADD、OP-FP）用に予約された7つの主要オペコードは、拡張機能に再利用できます。

同様に、AMOメジャーオペコードは、標準アトミック拡張が必要な​​い場合に再利用することができます。

インプリメンテーションが32ビットよりも長い命令を必要としない場合は、さらに4つの主要なオペコードが利用可能です（表19.1では灰色でマークされています）。

基本RV32Iエンコーディングは11の主要オペコードと3つの予約オペコードを使用し、最大18の拡張を利用できます。

ベースRV64Iエンコーディングは、13の主要オペコードと3つの予約オペコードを使用し、最大16の拡張を利用できます。

利用可能な22ビット命令エンコーディング空間

22ビットの符号化空間は、基本および標準の拡張エンコーディングのfunct3マイナーオペコード空間に対応します。

いくつかの主要なオペレーションコードには、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があり、利用可能ないくつかの22ビットエンコーディングスペースが残っています。

通常、メジャーオペコードは、命令の残りのビットに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、メジャーオペコードのオペランドフォーマットに従ってハードウェアデコードを単純化する必要があります。

その他のスペース

特定の主要なオペコードの下ではより小さいスペースが使用でき、すべてのマイナーオペコードが完全に埋められるわけではありません。

21.4整列した64ビット命令拡張の追加

基本32ビット固定幅命令フォーマットには大きすぎる拡張のためのスペースを提供する最も簡単な方法は、自然に整列した64ビット命令を追加することです。

実装では32ビットの基本命令フォーマットをサポートする必要がありますが、必要に応じて32ビットのNOP命令をアライメントパディングとして使用して、命令フェッチを簡略化するために64ビットの命令を64ビットの境界に配置する必要があります。

標準ツールの使用を簡素化するために、64ビット命令は図1.1のようにエンコードする必要があります。

しかし、実装は、32ビット命令の標準エンコーディングを保持しながら、64ビット命令の非標準命令長エンコーディングを選択する可能性があります。

例えば、圧縮された命令が必要でない場合、64ビット命令は、命令の最初の2ビットの1つ以上のゼロビットを使用して符号化することができます。

-----------------------------------------------------------------------------------------------------------------------------------

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が期待されます。

21.5 VLIWエンコーディングサポート

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、VLIWエンコーディングはいくつかの代替アプローチを使用して拡張として追加できます。

どのような場合でも、標準のソフトウェアツールを使用できるように、ベースの32ビットエンコーディングをサポートする必要があります。

固定サイズ命令グループ

最も簡単なアプローチは、VLIW演算が符号化される、自然にアライメントされた単一の命令フォーマット（例えば、128ビット）を定義することです。

従来のVLIWでは、このアプローチはNOPs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VLIWコードサイズ拡張をVLIWアクセラレーション関数に限定してベース32ビット命令もサポートする必要があります。

エンコードされた長さのグループ

別のアプローチは、図1.1の標準長符号化を使用して並列命令グループを符号化し、NOPsをVLIW命令から圧縮することができるようにすることです。

例えば、64ビット命令は2つの28ビット演算を保持することができ、96ビット命令は3つの28ビット演算を保持することができ、以下同様である。

あるいは、48ビット命令は1つの42ビット演算を保持することができ、96ビット命令は2つの42ビット演算を保持することができ、以下同様である。

このアプローチは、単一オペレーションを保持する命令に対してベースISAエンコードを保持する利点を有するが、VLIW命令内のオペレーションに新しい28ビットまたは42ビットエンコーディングを必要とし、より大きなグループに対してミスアライン命令フェッチを必要とするという欠点を有します。

1つの単純化は、VLIW命令がある特定のマイクロアーキテクチャ上重要な境界（例えば、キャッシュラインまたは仮想メモリページ）にまたがることを許さないことです。

固定サイズ命令バンドル

Itaniumに似た別のアプローチは、並列動作グループが符号化されているより大きな、自然に整列した固定命令バンドルサイズ（例えば、128ビット）を使用することです。

これは命令のフェッチを簡略化するが、複雑さをグループ実行エンジンに移行します。

RISC-V互換を維持するためには、基本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

上記のアプローチのいずれも、VLIW命令内の個々の演算についてRISC-Vエンコーディングを保持しません。

さらに別のアプローチは、固定幅32ビット符号化で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を使用して、「グループの終わり」が設定されていればそれを示すことができますが、2番目のビットはクリアであれば断定の下で実行を示すことができます。

VLIW拡張を認識しないツールによって生成された標準RISC-V 32ビット命令は、プレフィックスビットが両方ともセットされ（11）、したがって、各命令がグループの終わりにあり、断定されていない正しいセマンティクスを有します。

↑ここはよくわからない、プレフィックスビットによってVLIW拡張の最後がわかるってことかな。VLIWって超長命令語だから、長々とした命令の最後がわからないといけないよね。かな。

このアプローチの主な欠点は、ベースISAが、積極的なVLIWシステムで通常必要とされる複雑な述語サポートを欠いており、標準の30ビット符号化空間でより多くの述語レジスタを指定するためのスペースを追加することが難しいことです。

-- 1ページ空き、次ページへ

第22章

ISAサブセット命名規則

この章では、ハードウェア実装に含まれる一連の命令またはアプリケーションバイナリインタフェース（ABI）で使用される命令のセットを簡潔に記述するために使用されるRISC-V ISAサブセット命名方式について説明します。

-----------------------------------------------------------------------------------------------------------------------------------

RISC-V ISAは、さまざまな実験命令セット拡張を使用してさまざまな実装をサポートするように設計されています。

私たちは、整理された命名方式がソフトウェアツールとドキュメントを簡素化することを発見しました。

22.1大文字小文字の区別

ISAの命名文字列は、大文字と小文字を区別しません。

22.2基本整数ISA

RISC-V ISA文字列は、RV32I、RV32E、RV64I、またはRV128Iで始まり、サポートされているアドレス空間サイズを基本整数ISAのビットで示します。

22.3命令拡張機能名

標準のISA拡張には、単一の文字で構成される名前が与えられます。

たとえば、整数基底への最初の4つの標準拡張は次のとおりです。

  整数乗除算の場合は「M」、アトミックメモリ命令の場合は「A」、単精度浮動小数点命令の場合は「F」、倍精度浮動小数点命令の場合は「D」です。

すべてのRISC-V命令セットバリアントは、基本整数接頭辞を含まれる拡張の名前と連結することによって簡潔に記述することができます。

たとえば、「RV64IMAFD」と入力します。

"IMAFD"のベースとエクステンションを表す略語 "G"を定義しました。これは私たちの標準的な汎用ISAを表すことを意図しています。

RISC-V ISAへの標準拡張には、4倍精度浮動小数点の場合は「Q」、16ビット圧縮命令フォーマットの場合は「C」などの他の予約文字が与えられます。

22.4バージョン番号

命令セットが時間の経過と共に拡張または変更される可能性があることを認識し、サブセット名に続くサブセットバージョン番号をエンコードします。

バージョン番号は、「p」で区切られたメジャーバージョン番号とマイナーバージョン番号に分かれています。

マイナーバージョンが "0"の場合、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、後方互換性の喪失を意味しますが、マイナーバージョン番号のみの変更は後方互換性が必要です。

たとえば、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、「RV64I1p0M1p0A1p0F1p0D1p0」、簡潔には「RV64I1M1A1F1D1」、またはより簡潔には「RV64G1」と書かれています。

G ISAサブセットは、「RV64I2p0M2p0A2p0F2p0D2p0」、またはより簡潔に「RV64G2」と書くことができます。

2番目のリリースではバージョン番号付け方式を導入しました。これは永続的な標準になる予定です。

したがって、標準サブセットのデフォルトバージョンを本書の時点で存在するものと定義します。たとえば、「RV32G」は「RV32I2M2A2F2D2」に相当します。

22.5非標準拡張名

非標準のサブセットは、単一の "X"の後に文字で始まる名前とオプションのバージョン番号を使用して名前が付けられます。

例えば、 "Xhwacha"は、HwachaベクトルフェッチISA拡張の名前です。 "Xhwacha2"と "Xhwacha2p0"という名前はバージョン2.0と同じです。

非標準の拡張子は、単一のアンダースコアで他の複数文字の拡張子から分離する必要があります。

例えば、非標準拡張ArgleとBargleを持つISAの名前は "RV64GXargle\_Xbargle"です。

22.6スーパーバイザレベルの命令サブセット

標準スーパバイザ命令サブセットは、第2巻で定義されていますが、接頭辞として "S"を使用し、その後に文字とオプションのバージョン番号で始まるスーパバイザサブセット名が続きます。

スーパーバイザー拡張は、単一のアンダースコアで他の複数文字拡張と分離する必要があります。

22.7スーパーバイザレベルの拡張

スーパバイザ・レベルのISAへの非標準的な拡張は、 "SX"接頭辞を使用して定義されます。

22.8サブセット命名規則

表22.1に標準化されたサブセット名の要約を示します。

|  |  |
| --- | --- |
| サブセット | 名 |
| 標準汎用ISA | |
| 整数 | I |
| 整数乗算と除算 | M |
| アトミック | A |
| 単精度浮動小数 | F |
| 倍精度浮動小数点 | D |
| 一般 | G = IMAFD |
| 標準ユーザーレベル拡張 | |
| 四倍精度浮動小数点 | Q |
| 10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 | P |
| ベクトル拡張 | V |
| ユーザーレベルの割り込み | N |
| 非標準ユーザーレベル拡張 | |
| 非標準拡張子 “abc " | Xabc |
| スタンダードスーパーバイザレベルISA | |
| スーパーバイザ拡張 "def" | Sdef |
| 非標準スーパーバイザレベル拡張 | |
| スーパーバイザ拡張 "ghi" | SXghi |

表22.1：標準ISAサブセット名

テーブルはまた、サブセット名が名前文字列に現れなければならない標準的な順序を定義します。例えば、RV32IMAFDQCは正当であるが、RV32IMAFDCQはそうではないなど、名前の文字列の最初から最後までを表の上から下に示します。

-- 1ページ空き、次ページへ

第23章

歴史と謝辞

23.1 ISAマニュアルのリビジョン1.0の歴史

RISC-V ISAと命令セットマニュアルは、以前のいくつかのプロジェクトを構築します。

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UC バークレーとICSIのT0（急流-0）ベクトルマイクロプロセッサプロジェクトに戻っています。

T0はMIPS-II ISA、クレステ・アサノビック を主アーキテクト、RTLデザイナー、ブライアン キングスブリとバートランド イリソウ を主要なVLSI実装者とするベクトルプロセッサーでした。

ICSIのディビッド ジョンソンは、T0 ISAの設計、特に監督者モード、およびマニュアルテキストに大きく貢献しました。

ジョン ハウザー氏は、T0 ISA設計についても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（ソフトウェア制御アーキテクチャ）は、T0プロジェクトインフラストラクチャに基づいて構築され、スーパーバイザレベルのインターフェイスを洗練し、分岐遅延スロットを削除することでMIPSスカラISAから離脱しました。

ロニー クラシンスキーとクリストファー バテンは、MITの スケール ベクター - スレッドプロセッサの主要アーキテクトでした。マーク ハンプトンは、GCCベースのコンパイラ・インフラストラクチャとスケール用ツールを移植しました。

MIT 6.371 導入 VLSI システムクラスの新バージョンを2002年秋に教授する際に、T0 MIPSスカラー・プロセッサー仕様（MIPS-6371）の軽く編集されたバージョンを使用し、クリス ターマンと クレステ・アサノビック を講師にしました。

クリス ターマンはクラスのラボ用資料のほとんどを寄稿しました（TAはありませんでした！）。

6.371クラスは2005年春にアルビンドと クレステ・アサノビック によって教えられたMITのトライアル6.884 複雑なデジタルデザインクラスに進化し、春期クラス6.375となりました。

SMIPSという名前のスケール MIPSベースのスカラーISAの縮小バージョンが6.884 / 6.375で使用されました。

クリストファー バテンは、これらのクラスの早期提供のためのTAであり、SMIPS ISAをベースとしたかなりの量のドキュメンテーションとラボの資料を開発しました。

この同じSMIPSラボの材料は、ジョン ウォウルズニック、クレステ・アサノビック、ジョン ラッツアロが教えたUC バークレイ 秋l 2009 CS250 VLSIシステム設計クラスのTA ユンサップ リーによって改訂されました。

Maven（ベクトルスレッドエンジンの順応性のある配列)）プロジェクトは、第2世代のベクトル型アーキテクチャでした。

そのデザインは、2007年夏にUCバークレー校で交換奨学生を迎えたときにクリストファー バテンによって率いられました。

日立製作所の客員研究員青木秀隆は、初期のMaven ISAとマイクロアーキテクチャ設計についてかなりのフィードバックを寄せました。

Mavenインフラストラクチャはスケールインフラストラクチャに基づいていましたが、Maven ISAはスケール化されたMIPS ISAバリアントから、統一された浮動小数点および整数レジスタファイルでさらに遠ざかりました。

Mavenは、代替のデータ並列アクセラレータでの実験をサポートするように設計されています。

ユンサップ リーは様々なMavenベクトルユニットの主な実装者でしたが、ライマス アビ ゼニスは様々なMavenスカラーユニットの主な実装者でした。

ユンサップ リーとクリストファー バテンは、新しいMaven ISAで動作するようにGCCを移植しました。

クリストファー チェリオは、Mavenの伝統的なベクトル命令セット（ "フラッド"）バリアントの初期定義を提供しました。

これらすべての以前のプロジェクトの経験に基づいて、RISC-V ISAの定義は2010年夏に開始されました。

RISC-V 32ビット命令サブセットの初期バージョンは、ユンサップ リーをTAとして迎えたUC バークレー 秋 2010 CS250 VLSIシステム設計クラスで使用されました。

RISC-Vは、以前のMIPSからインスピレーションを得たデザインからクリーンなブレークです。

ジョン ハウザーは、サインイン命令と浮動小数点値の内部再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。

23.2 ISAマニュアルのRevision 2.0の歴史

図23.1に示すように、いくつかのシリコン製作を含め、RISC-Vプロセッサの複数の実装が完了しています。

|  |  |  |  |
| --- | --- | --- | --- |
| 名 | テープアウト日 | プロセス | ISA |
| Raven-1 | 2011/05/29 | ST 28nm FDSOI | RV64G1 Xhwacha1 |
| EOS14 | 2012/04/01 | IBM 45nm SOI | RV64G1p1 Xhwacha2 |
| EOS16 | 2012/08/17 | IBM 45nm SOI | RV64G1p1 Xhwacha2 |
| Raven-2 | 2012/08/22 | ST 28nm FDSOI | RV64G1p1 Xhwacha2 |
| EOS18 | 2013/02/06 | IBM 45nm SOI | RV64G1p1 Xhwacha2 |
| EOS20 | 2013/07/03 | IBM 45nm SOI | RV64G1p99 Xhwacha2 |
| Raven-3 | 2013/09/26 | ST 28nm SOI | RV64G1p99 Xhwacha2 |
| EOS22 | 2014/03/07 | IBM 45nm SOI | RV64G1p9999 Xhwacha3 |

表23.1：製造されたRISC-Vテストチップ。

製造される最初のRISC-VプロセッサはVerilogで作成され、2011年にRaven-1テストチップとしてSTの試作28nm FDSOI処理技術で製造されました。

2つのコアは クレステ・アサノビック によってアドバイスされたユンサップ リーとアンドリュー ウオーターマンによって開発され、一緒に製作されました：

 1）エラー検出フリップフロップを備えたRV64スカラーコア、および

2）64ビット浮動小数点ベクトルユニットが接続されたRV64コア。

最初のマイクロアーキテクチャは、未熟なデザインライブラリで設計を完了するのに利用できる短い時間のため、非公式に「列車事故」と呼ばれていました。

続いて、インオーダー デカップリングされたRV64コア用のクリーンなマイクロアーキテクチャが、クレステ・アサノビック によってアドバイスされたアンドリュー ウオーターマン、ライマス アビ ゼニス、およびユンサップ リーによって開発され、鉄道のテーマを続けて、ジョージ・スティーブンソンの蒸気機関車の設計が成功した後、「ロケット」というコードネームが付けられました。

ロケットはUC バークレーで開発された新しいハードウェア設計言語チセルで書かれました。

ロケットで使用されるIEEE浮動小数点ユニットは、ジョン ハウザー、アンドリュー ウオーターマン、ブライアン リチャードによって開発されました。

ロケットはこれまでより洗練されて開発されており、Photonicsプロジェクトでは、28nm FDSOI（Raven-2、Raven-3）で2回、IBM 45nm SOI技術（EOS14、EOS16、EOS18、EOS20、EOS22）で5回製造されています。

ロケット設計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作業が進行中です。

EOS14-EOS22チップには、クレステ アサノビックによってアドバイスされたユンサップ リー、アンドリューウォーターマン、フィ ボー、アルバート オウ、クアン グエン、スティーブン トゥイグによって開発された64ビットIEEE浮動小数点ベクトルユニットHwachaの初期バージョンが含まれています。

EOS16-EOS22チップには、クレステ アサノビックによってアドバイスされたヘンリー コックと アンドリューウォーターマン によって開発されたキャッシュコヒーレンスプロトコルを備えたデュアルコアが含まれています。

EOS14シリコンは1.25GHzで正常に動作します。

EOS16シリコンは、IBMパッドライブラリのバグに悩まされていました。

EOS18とEOS20は1.35GHzで正常に動作します。

Raven(ワタリガラス)テストチップの貢献者には、ユンサップ リー、アンドリューウ ォーターマン 、ライマス アビ ゼニス、ブライアン ジマー 、ヤハ クアク、ルジカジェフ ティーチ、ミロバン ブラゴエビック、アルベルト プゲリ、スティーブン ベイリー、ベン ケラー、パイ フェン チウ、ブライアン リチャーズ、ボリヴォエ ニコリック、クレステ アサノビック。

EOSテストチップの貢献者には、ユンサップ リー、ライマス アビ ゼニス、アンドリューウ ォーターマ、ヘンリー コック、フィ ボー、ダイウェイ リー、チェン サン、アルバート オウ、クアン グエン、スティーブン トゥイグ、ウラジミール ストヤノビッチ、クレステ アサノビック が含まれます。

アンドリューウ ォーターマン とユンサップ リー は、開発中の金色モデルとして使用され、米国の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の名前を付けて、C ++ ISAシミュレータ「スパイク」を開発しました。

スパイク(犬釘)はBSDのオープンソースプロジェクトとして公開されました。

アンドリューウ ォーターマン は、RISC-V圧縮命令セット[33]の暫定的な設計で修士論文を修了しました。

RISC-VのさまざまなFPGA実装が完了しました。主にPar Labプロジェクトリサーチの統合デモの一部として完成しました。

最大のFPGAデザインには、研究オペレーティングシステムを実行する3つのキャッシュコヒーレントなRV64IMAプロセッサがあります。

FPGA実装の貢献者には、アンドリューウ ォーターマン、ユンサップ リー、ライマス アビ ゼニス、クレステ アサノビック が含まれます。

RISC-Vプロセッサは、UC バークレーのいくつかのクラスで使用されています。

ロケットは2011年秋にクラスプロジェクトの基盤としてCS250を使用し、ブライアン ジマー はTAとして使用されました。

2012年春の学部のCS152クラスでは、チセルを使って、 "トーマスタンクエンジン"と友達が住む島の後ろに "ソドー島"という名前の教育用RV32プロセッサーを書きました。

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および5段階のパイプライン化されたコアが含まれており、BSDライセンスで一般に公開されています。

このスイートは2013年春のCS152、TAとしてのユンサップ・リー、そしてTAとしてのエリック・ラブとの2014年春に更新され、再び使用されました。

クリストファー チェリオは、CS152クラスで使用されていたパイプラインのビジュアライゼーションを伴う、BOM（バークレーアウトオブオーダーマシン）と呼ばれる順序外のRV64デザインを開発しました。

CS152クラスは、アンドリュー ウォーターマンとヘンリー クックによって開発されたロケットコアのキャッシュコヒーレントバージョンも使用していました。

2013年の夏、ロケットコアにカスタムアクセラレータを追加することを簡単にするために、RoCC（ロケットカスタムコプロセッサ）インターフェイスが定義されました。

ロケットとRoCCインタフェースは、RoCCインタフェースに構築されたいくつかの学生アクセラレータプロジェクトとともに、ジョナサン バックラックが指導する秋の2013 CS250 VLSIクラスで幅広く使用されていました。

HwachaベクトルユニットはRoCCコプロセッサとして書き直されました。

バークレーの2人の大学生であるクアン グレンとアルバート オーは、2013年春にLinuxをRISC-V上で動作させることに成功しました。

コリン・シュミットtは、2014年1月にRISC-V 2.0のLLVMバックエンドを正常に完成しました。

ブルースペックのダリウス・ラッドは、2014年3月にGCCポートにソフトフロートのABIサポートを提供しました。

ジョン・ハウザーは、浮動小数点の分類命令の定義に貢献しました。

私たちは、Verilogのトミー ソーンとリシュリュー ニキルのブルースペックのものを含む、他のいくつかのRISC-Vコア実装について認識しています。

謝辞

ISAバージョン2.0仕様の草案についてのコメントのために、クリストファー F バテン、プリストン ブリッグス、クリストファーチェリオ、デビッド チスナール、ステファン フリューデンバーガー、ジョン ハウザー、ベン ケラー、リシュリュー ニキル、マイケル テイラー、トミー ソーン、ロバート ワトソンに感謝します。

23.3 改訂2.1の履歴

RISC-V ISAの取り込みは、2014年5月に凍結されたバージョン2.0が導入されて以来、非常に迅速であり、このような短い履歴セクションに記録するには多すぎる活動をしています。

多分最も重要なのは、2015年8月に非財団RISC-V財団が結成されたことです。

財団は現在公式のRISC-V ISA標準の管理を引き継いでおり、オフィシャルウェブサイトriscv.orgはRISC-V標準に関するニュースと最新情報を入手するのに最適な場所です。

謝辞

バージョン2.0の仕様については、スコット バーマー、アレン J バウム、クリストファーチェリオ、デビッド チスナール 、ポールクレイトン、パーマー ダベルト、ヤン グレイ、マイケル ハンブルク、ジョン ハウザーのおかげで感謝しています。

23.4改訂2.2の履歴

謝辞

ジェイコブ バッハメイヤー、アレックス ブラッドベリー、デビッド ホーナー、ステファン オレア、ジョセフ マイヤーズから、バージョン2.1の仕様に関するコメントがあり、ありがとうございました。

23.5資金調達

RISC-Vアーキテクチャの開発と実装は、以下のスポンサーによって部分的に資金提供されています。

* パーラボ：マイクロソフト（アワード ＃024263）とIntel（アワード ＃024894）による資金提供と、U.C ディスカバリー（アワード ＃DIG07-10227）。

パーラボがNokia、NVIDIA、Oracle、Samsungに加わったことから、さらなるサポートが得られました。

* プロジェクトイシス：DoEアワードDE-SC0003624
* ASPIREラボ：DARPA PERFECTプログラム、アワードHR0011-12-2-0016。

DARPA POEMプログラム アワードHR0011-11-C-0100。

セミコンダクター・リサーチ・コーポレーションから資金提供を受けているSTARnetのセンターである未来アーキテクチャー研究センター（C-FAR）のセンター。

ASPIRE産業スポンサー、Intel、ASPIREの関係者、Google、Hewlett Packard Enterprise、Huawei、Nokia、NVIDIA、Oracle、Samsungからの追加サポート

本書の内容は、必ずしも米国政府の立場や方針を反映するものではなく、正式な裏書を推論すべきではありません。

-- 1ページ空き、次ページへ

参考文献

[1] RISC-V ELF psABI仕様。

https://github.com/riscv/riscv-elf-psabi-doc/

[2] 32ビットマイクロプロセッサのIEEE標準。

IEEE Std。 1754-1994,1994。

[3] G. M. アムダール、G.A. ブラウ、およびJr. F. P. ブルックス。 IBM System / 360のアーキテクチャ

IBM ジャーナル of R＆D、8（2）、1964。

[4] クレステ アサノビック。ベクトルマイクロプロセッサ。博士論文、カリフォルニア大学バークレー校、1998年5月

テックリポート UCB / CSD-98-1014として入手可能です。

[5] W. ブッフホルツ。 IBM システム / 370ベクタ・アーキテクチャ

IBM システムズ ジャーナル、25（1）：51-62、1986。

[6] ワーナー ブックホルツ、編集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。

Mcグロウ-ヒル ブック 社、1962年。

[7] クレイ Inc. クレイアセンブリ言語（CAL）for クレイ X1 システムズリファレンスマニュアル、1.1版、2003年6月。

[8] K.ディフェンドロフ、P.K. デュビー、R.ホックスプラング、およびH.スケール。

PowerPCのAltiVec拡張により、メディア処理が高速化されます。

IEEE Micro、20（2）：85-95,2000。

-- 調べたら 「AltiVecとは、1998年にMotorolaによって発表された、CPUのマルチメディア拡張機能のことである。」との事

[9] ジョン M. ランクオビチとH. フィリップ パターソン。

リンカーンTX-2コンピュータの機能説明。

1957年2月、カリフォルニア州ロサンゼルスのウエスタンジョイントコンピューター会議で。

[10] クーロス ハラチオルー、ダニエル レノスキー、ジェームズ ラウドン、フィリップPhillip ギボンス、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整合性とイベントの順序

第17回国際コンピュータシンポジウム講演予稿集、15頁（26頁、1990年）。

[11] J. グッドエーカーおよびA.N.スロス。

並列性とARM命令セットアーキテクチャ

コンピュータ、38（7）：42-50,2005。

[12] リンリー グエンナップ。

デジタル、MIPSはマルチメディア拡張を追加します。

マイクロプロセッサーレポート、1996年。

[13] ティモシー H. ヘイルとジェームズ E. スミス。選択的なデュアルパス実行。

技術報告、ウィスコンシン大学 - マディソン、1996年11月。

[14] ANSI / IEEE Std 754-2008、浮動小数点演算のIEEE標準、2008。

[15] マリノス G.H. ケートヴェニス、ロバート W. シャバーン、Jr.、ディビッド A. パターソン、およびカルロo H. シークイン。

RISC IIマイクロアーキテクチャ

プロセッシング VLSI 83 カンファレンス、1983年8月。

[16] ヘスン キム、オヌア ムトル、ジャレッド スターク、エール N. パット。欲しいブランチ：

条件付き分岐と述語を結合して適応型述語実行を行う。

第38回IEEE / ACMマイクロシンポジウム国際シンポジウム予稿集、マイクロ 38、43-54頁、2005年。

[17] A. クラウザー、T. オウスチン、D. グルンバルド、およびB. カルダー。

非宣言命令セットアーキテクチャのための動的ハンモック予測。

PACT'98、ワシントン、DC、USA、1998年、並列アーキテクチャとコンパイル技術に関する1998年国際会議の議事録。

[18] デイビッド D. リー、シン I. コン、マーク D. ヒル、ジョージ S. テイラー、ディビッド A. ホッジス、

ランディ H.カッツ、ディビッド 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 - 第1部：

コプロセッサインタフェースを備え、シンボリック処理をサポートするRISCマイクロプロセッサ。

IEEE JSSC、24（6）：1688-1698、1989年12月。

[19] R.B. リー。 MAX-2とのサブワード並列処理。IEEE マイクロ、16（4）：51-59、1996年8月。

[20] クリス・ロモント。インテル®アドバンスト・ベクトル拡張機能の紹介。インテルホワイトペーパー、2011。

[21] 三浦健一と内田啓一郎。

FACOMベクトルプロセッサーシステム：VP-100 / VP-200。

カワリック編集者、高速計算に関するNATO高度研究ワークショップ議事録、第F7巻。

スプリンガー-フェラーク、1984。IEEEチュートリアルスーパーコンピュータ：設計と応用。

カイ ウォン（編集者）、pp59-73。

[22] オープンコアーズ。OpenRISC 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

[23] ディビッド A. パターソンとカルロ H. シークイン。RISC I：縮小命令セットVLSIコンピュータ。

ISCAでは、443頁-458,1981。

[24] A. ペレグおよびU. ワイザー。IntelアーキテクチャへのMMXテクノロジ拡張。

IEEE マイクロ、16（4）：42-50、1996年8月。

[25] ラヴィ ラジワとジェームズ R. グッドマン。

投機的なロック・エリート：並行性の高いマルチスレッド実行を可能にする。

第34回ACM / IEEE国際シンポジウム講演予稿集、マイクロ34、294〜305頁。

IEEEコンピュータ社会、2001年。

[26] S.K. ラマン、V.ペントコフスキイ、およびJ.ケーシャヴァ。Pentium-IIIプロセッサにストリーミングSIMD拡張を実装する。

IEEE マイクロ、20（4）：47-57,2000。

[27] バララム シナロイ、R. カルラ、WJ シュタルケ、HQ リ、R. カルゴニ、JA バン ノーストランド、BJ ロンチェッティ 、

J. ストウエッチリ、J.レーンストラ、GL ガスリー、DQ グエン、B. ブラナー、CF マリノ、E. レッター、およびP. ウィリアムズ。

IBM POWER7マルチコア・サーバー・プロセッサー。

IBM 研究開発のジャーナル、55（3）：1-1、2011。

[28] ジェームズ E. ソーントン。制御データ6600の並列操作。

1964年10月27-29日の論文集、秋の共同コンピュータ会議、パートII：

超高速コンピュータシステム、AFIPS'64（秋、パートII）、33-40頁、1965年。

[29] M. トランブレー、J.M.オ・コナー、V.ナラヤナン、および リアン ヒー。

VISは新しいメディア処理をスピードアップします。

IEEE マイクロ、16（4）：10-20、1996年8月。

[30] マルク トランブレー、ジェフリー チャン、シャリエンダー チョードリー、アンドリュー W. コニグリアロ、

およびシン ション ティ。

MAJCアーキテクチャ：並列性とスケーラビリティの統合

IEEE Micro、20（6）：12-25、2000を参照のこと。

[31] J. ツェンおよびK. アサノビッチ。エネルギー効率に優れたレジスタアクセス。

Proc。第13回集積回路およびシステム設計シンポジウム、377-384ページ、マナウス、ブラジル、2000年9月。

[32] ディビッド アンガー、リッキー ブラウ、ペーター フォレイ、ダイン サンプルス、ディビッド パターソン。

SOARのアーキテクチャ：RISC上のスモールトーク。 ISCA、188-197ページ、アン アーバー、MI、1984。

[33] 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縮小

修士論文、カリフォルニア大学バークレー校、2011

[34] 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

[35] アンドリュー ウォーターマン、ユンサップ リー、ディビッド A. パターソン、クレステ アサノビック。

RISC-V命令セットマニュアル、第1巻：ベースユーザレベル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

[36] アンドリュー ウォーターマン、ユンサップ リー、ディビッド A. パターソン、クレステ アサノビック。

RISC-V命令セットマニュアル、第1巻：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、2014年5月

-- 最終頁 ここまで。

-- 2018/08/13