

RISC-V プロセッサ トレース
バージョン 1.0
4d009c4de4c68d547adb4adec307a438feb3d815

ガジンダー・パネサー, アイアン・ロバートソン
<gajinder.panesar@ultrasoc.com>, <iain.robertson@ultrasoc.com>
ウルトラソック テクノロジーズ株式会社

2020 年 4 月 13 日

日本語訳 @shibatchii

2020 年 6 月 7 日

内容

1.	導入.....	1
1.1	用語集.....	2
1.2	命名法.....	3
2.	分岐トレース	5
2.1	命令デルタ・トレースの概念.....	6
2.1.1	順次命令.....	6
2.1.2	不連続性の PC.....	6
2.1.3	分岐	6
2.1.4	割り込みと例外.....	6
2.1.5	同期	7
2.1.6	トレースの終了.....	7
2.2	オプションおよびランタイム構成可能モード.....	7
2.2.1	デルタアドレスモード	8
2.2.2	フルアドレスモード	8
2.2.3	暗黙的な例外モード	9
2.2.4	順次推定ジャンプモード	9
2.2.5	暗黙的な復帰モード	9
2.2.6	分岐予測モード	10
2.2.7	ジャンプターゲットキャッシュモード	11
3.	エンコーダーインターフェイスへのハート.....	13

3.1	インターフェイスの要件	13
3.1.1	ジャンプ分類とターゲット推論	15
3.1.2	関係の between RISC-V コアとエンコーダー	16
3.2	命令インターフェース	16
3.2.1	単一退職の簡素化	20
3.2.2	代替の複数リタイアインターフェース構成	20
3.2.3	オプションのサイドバンド信号	20
3.2.4	デバッグモジュールからのトリガ出力の使用	22
3.2.5	リタイアメントシーケンスの例	22
4.	フィルタリング	23
5.	エンコーダ出力パケットのトレース	25
5.1	フォーマット 3 パケット	27
5.2	フォーマット 3 サブフォーマット 0 - 同期	27
5.2.1	フォーマット 3 分岐フィールド	27
5.3	フォーマット 3 サブフォーマット 1 - 例外	28
5.3.1	フォーマット 3 の tvalepc フィールド	28
5.4	フォーマット 3 サブフォーマット 2 - コンテキスト	28
5.5	フォーマット 3 サブフォーマット 3 - サポート	28
5.5.1	フォーマット 3 サブフォーマット 3 qual_status フィールド	29
5.6	フォーマット 2 パケット	31
5.6.1	フォーマット 2 通知フィールド	31
5.6.2	フォーマット 2 通知フィールドとアップディスクフィールド	32
5.6.3	形式 2 のレポートと詳細	33
5.7	フォーマット 1 パケット	34
5.7.1	フォーマット 1 アップディスクフィールド	34
5.7.2	フォーマット 1 branch_map フィールド	34

5.7.3	フォーマット 1 のイステータスおよびイデプスフィールド	36
5.8	フォーマット 0 パケット	37
5.8.1	フォーマット 0 サブフォーマットフィールド	37
5.8.2	フォーマット 0 branch_fmt フィールド	41
5.8.3	フォーマット 0 のイステータスフィールドとイデプスフィールド	41
6.	リファレンスアルゴリズム	43
6.1	フォーマットの選択	44
6.2	再同期	46
6.3	複数のリタイアに関する考慮事項	47
7.	パラメータと検出	49
7.1	エンコーダパラメータの検出	51
7.2	ipxact の説明例	52
8.	今後の方向性	57
8.1	データトレース	57
8.2	高速プロファイルリング	57
8.3	命令間サイクルカウント	57
8.4	トランスポート	58
9.	デコーダー	59
9.1	デコーダ擬似コード	59
10.	コードとパケットの例	67
11.	奥付	71

図一覧

5.1 カプセル化されたパケットフォーマットの例	25
6.1 命令差分トレースアルゴリズム	45

表一覧

3.1 命令インターフェイス信号	18
3.2 命令インターフェイス信号 - ブロックごとに複数のリタイア	19
3.3 命令インターフェイス信号 - ブロックあたりの単一のリタイア	19
3.4 コンテキストタイプの <code>ctype</code> 値と対応するアクション	19
3.5 オプションのサイドバンドエンコーダ入力信号	21
3.6 オプションのサイドバンドエンコーダ出力信号	21
3.7 デバッグモジュールトリガーサポート (<i>mcontrol</i> アクション)	22
3.8 例 1 : 9 命令は 4 サイクルにわたって退避され、2 つのブランチ	22
5.1 パケット形式 3、サブフォーマット 0	27
5.2 パケット形式 3、サブフォーマット 1	28
5.3 パケット形式 3、サブフォーマット 2	29
5.4 パケット形式 3、サブフォーマット 3	30
5.5 パケット形式 2	31
5.6 パケット形式 1 - アドレス、ブランチ マップ	35
5.7 パケット形式 1 - アドレスなし、ブランチマップ	36
5.8 パケット形式 0、サブフォーマット 0 - アドレスなし、ブランチカウント	37
5.9 パケット形式 0、サブフォーマット 0 - アドレス、ブランチカウント	38
5.10 パケット形式 0、サブフォーマット 1 - ジャンプ ターゲット インデックス、ブランチ マップ	39
5.11 パケット形式 0、サブフォーマット 1 - ジャンプ ターゲット インデックス、ブランチ マップなし	40
7.1 エンコーダへのパラメータ	50

7.2 必要とされる属性	51
7.3 オプションのフィルタリング属性	51
7.4 その他の推奨属性	52

第 1 章

1. 導入

複雑なシステムでは、プログラムの動作を理解することは容易ではありません。当然のことながら、このようなシステムでは、ソフトウェアが期待どおりに動作しないことがあります。これは、他のコアとの相互作用、ソフトウェア、周辺機器、リアルタイムイベント、実装不良、または上記の全ての組み合わせなど、多くの要因が原因である可能性があります。

デバッガーを使用して、実行中のシステムの動作を監視することは、常に、これは邪魔であるために使用できるわけではありません。プログラム実行の可視性を提供することは重要です。これは、膨大な量のデータをシステムに押し込まずに行う必要があります。

これを実現する方法の 1 つは、プロセッサ ブランチ トレースを使用することです。

これは、既知の開始アドレスからの実行を追跡し、プログラムによって取得されたアドレスデルタに関するメッセージを送信することによって機能します。これらのデルタは通常、ジャンプ、コール、リターン、ブランチのタイプの命令によって導入されますが、割り込みと例外もデルタのタイプです。

概念的には、システムには次の基本コンポーネントが 1 つ以上含まれています。

- プロセッサブランチトレースなどを正常に作成するために、すべての関連情報を出力する命令トレースインターフェースを備えたコア。これは高帯域幅のインターフェイスです：ほとんどの実装では、コア実行クロック サイクルごとに大量のデータ（命令アドレス、命令タイプ、コンテキスト情報、.）を提供します。
- この命令トレース インターフェイスに接続し、情報を低帯域幅トレース パケットに圧縮するハードウェア エンコーダー。
- 送信する伝送チャネル、またはこれらのトレース パケットを格納するメモリ。
- デコーダは、通常は外部 PC 上のソフトウェアであり、トレース パケットを取り込み、元の hart で実行されているプログラム バイナリの知識を持ってプログラム フローを再構築します。このデコードステップは、hart の実行中に、オフラインまたはリアルタイムで行うことができます。

RISC-V では、すべての命令が無条件に実行されるか、または少なくともそれらの実行はプログラムバイナリに基づいて決定することができます。デルタ間の命令は、すべて順次実行されると仮定することができます。

このため、トレース内の順次命令を報告する必要はなく、分岐が取られたかどうか、および取られた間接分岐またはジャンプのアドレスのみを報告します。プログラム カウンタが実行バイナリから判断できない量だけ変更された場合、トレース デコーダには宛先アドレス（次の有効な命令のアドレス）を指定する必要があります。この例は、間接分岐またはジャンプで、次の命令アドレスは、プログラムバイナリに組み込まれた定数ではなく、レジスタの内容によって決定されます。

割り込みは、通常、特定の命令またはイベントの結果として意図的にではなく、プログラムの実行に非同期的に発生します。例外は、通常は特定の命令アドレスにリンクしても、同じ方法で考えることができます。デコーダーは、一般的に命令シーケンス内で割り込みが発生する場所を認識しないため、トレース エンコーダーは、通常のプログラム フローが停止したアドレスを報告する必要があります。割り込みまたは例外が発生した場合、またはプロセッサが停止した場合は、最後の命令が事前にトレースに含まれている必要があります。

このドキュメントでは、入力ポート（RISC-V コアとエンコーダ間の信号）、圧縮プランチ トレース アルゴリズム、および圧縮プランチ トレース情報のカプセル化に使用されるパケット形式を指定する機能を提供します。

1.1 用語集

以下の用語は、本仕様において特定の意味を持ちます。

- **ATB:** アームトレースバス
- **branch:** 実行フローを条件付きで変更する命令
- **CSR:** 制御/ステータスレジスタ
- **decoder:** エンコーダーによって出力されたトレース パケットを受け取り、RISC-V のハートによって実行されるコードの実行フローを再構築するソフトウェアの一部
- **delta:** メモリに連續して配置された 2 つの命令間の差以外のプログラムカウンタの変化
- **discontinuity:** 'デルタ' の別の名前（上記を参照）
- **ELF:** 実行可能およびリンク可能なフォーマット
- **encoder:** RISC-V の hart から命令実行情報を取り込み、トレース パケットに変換するハードウェア
- **exception:** RISC-V のハルトの命令に関連付けられた実行時に発生する異常な状態
- **hart:** RISC-V ハードウェア スレッド
- **interrupt:** RISC-V のキーが予期しない制御の転送を経験する原因となる外部非同期イベント

- **ISA:** 命令セットアーキテクチャ
- **jump:** 無条件に実行フローを変更する命令
- **direct jump:** PC を定数値で変更して実行フローを無条件に変更する命令
- **indirect jump:** PC を計算値に変更して実行フローを無条件に変更する命令
- **inferable jump:** ジャンプオペコード内に埋め込まれた定数を介してターゲットアドレスが提供されるジャンプ
- **uninferable jump:** 推論できないジャンプ(上記参照)
- **LSB:** 最下位ビット
- **MSB:** 最上位ビット
- **packet:** エンコーダーによって生成されたエンコードされたトレース情報の原子(不可分)単位
- **PC:** プログラムカウンター
- **program counter:** 実行されている命令のアドレスを含むレジスタ
- **retire:** マシンの状態が更新されたときの命令の実行の最終段階（”コミット” または ”卒業” とも呼ばれます）
- **trap:** 例外または割り込みによって発生したトラップハンドラへの制御の転送
- **updiscon:** 「推し量れない PC の不連続性」の縮小

1.2 命名法

次のセクションでは、太字の項目は、パケット内のシグナルまたはフィールドです。

太字のイタリック体の項目は、RISC-V ISA で定義されている命令または CSR のニーモニックです。

名前が '*p*' の斜体の項目は、ハードウェアに組み込まれているパラメーターまたは構成可能なハードウェア値を指します。

第 2 章

2. 分岐トレース

命令デルタトレース（分岐トレースとも呼ばれます）は、プログラムによって取得されたデルタに関する情報を送信することによって、既知の開始アドレスからの実行を追跡することによって機能します。デルタは通常、ジャンプ、コール、リターン、ブランチタイプの命令によって導入されますが、割り込みと例外もデルタのタイプです。

命令デルタトレースは、プロセッサが実行しているプログラムに基づいてプロセッサが動作する決定的な方法を利用して、命令シーケンスの効率的なエンコーディングを提供します。

このアプローチは、デコーダが利用できるプログラムバイナリのオフラインコピーに依存するため、通常、動的（自己変更）プログラムやプログラムバイナリへのアクセスが禁止されているプログラムには適していません。

プログラムバイナリは十分ですが、アセンブリまたは上位レベルのソースコードにアクセスすると、トレースされた命令にソースコード行番号やラベル、変数名などを付け加えることで、デコーダがデバッグにデコードされたトレースを表示する能力が向上します。

このアプローチは、デコーダがターゲットから命令メモリを要求するように配置することで、決定的に動的なコードの小さなセクションに対処するために拡張することができます。メモリルックアップは一般的にパフォーマンスの低下につながりますが、ブートアップ時やサービスの登録時に調整される可能性のあるオペレーティングシステムの例外/割り込みベクトルポインタなど、控えめなジャンプテーブルを調べるのに適しています。静的プログラムと動的にリンクされたプログラムの両方を、この方法でトレースできます。静的にリンクされたプログラムは、一般に既知のアドレス空間で動作するため、通常は物理メモリに直接マッピングされるため、簡単です。動的にリンクされたプログラムでは、トレースまたはストップモードデバッグを使用してメモリ割り当て操作を追跡する必要があります。

2.1 命令デルタ・トレースの概念

2.1.1 順次命令

RISC-V のような命令セットアーキテクチャでは、すべての命令が無条件に実行されるか、または少なくともその実行はプログラムバイナリに基づいて決定することができる場合、デルタ間の命令は順次実行されるものと見なされます。したがって、トレースでレポートする必要はありません。トレースには、分岐が取得されたかどうか、取得した間接ジャンプのアドレス、またはその他のプログラム カウンターの不連続性のみが含まれている必要があります。

2.1.2 不連続性の PC

不变プログラムカウンタ不連続とは、プログラムのバイナリ単独から推測できないプログラムカウンタの変更です。このような場合、命令デルタ・トレースには宛先アドレス（次の有効な命令のアドレス）が含まれている必要があります。

間接ジャンプは、次の命令アドレスがプログラムバイナリに埋め込まれた定数ではなく、レジスタの内容によって決定されるこの例です。この場合、ジャンプ後の命令のアドレス（ジャンプターゲットとも呼ばれる）をトレースする必要があります。

割り込みと例外は、もう 1 つの不連続性の PC の不連続性の形式です。これらは以下で詳しく説明します。

2.1.3 分岐

分岐とは、ジャンプがレジスタまたはフラグの値に条件付きである命令です。デコーダがプログラム フローに従うようにするには、トレースに分岐が実行されたかどうかを含める必要があります。

直接分岐の場合、宛先アドレスがプログラムバイナリでエンコードされている場合（定数として、またはプログラム カウンタからの定数オフセットとして）、それ以上の情報は必要ありません。直接分岐は RISC-V ISA でサポートされている唯一のブランチ タイプです。

2.1.4 割り込みと例外

割り込みは、特定の命令やイベントの結果として意図的にではなく、プログラムの実行に対して非同期的に発生する、異なる種類のデルタです。例外は、通常は特定の命令アドレスにリンクしても、同じ方法で考えることができます。

デコーダーは、一般に命令シーケンス内で割り込みが発生した場所を認識しないため、トレースは通常のプログラム フローが停止したアドレスを報告するとともに、例外の種類を報告するのと同じくらい簡単な非同期の送り先を示す必要があります。割り込みまたは例外が発生した場合、最後の命令は事前にトレースする必要があります。これに続いて、次の有効な命令アドレス（トラップ・ハンドラーの最初のアドレス）をトレースする必要があります。

注: すべての例外および割り込みがトラップを引き起こすわけではありません (定義についてはセクション 1.1 を参照)。最も顕著なのは、浮動小数点例外と無効な割り込みはトラップされません。例外または割り込みがトラップされない場合、プログラム カウンターは変更されません。だから、すべての例外/割り込みをトレースする必要はなく、単にトラップします。このドキュメントでは、割り込みと例外は、トラップが発生した場合にのみトレースされます。

2.1.5 同期

トレースを堅牢にするには、トレース内に定期的な同期ポイントが必要です。同期は、値を指定した完全な命令アドレス (および場合によってはコンテキスト識別子) を送信することによって行われます。デコーダとデバッガは、同期の理由を送信してもメリットがあります。同期の頻度は、ロバストネスとトレース帯域幅のトレードオフです。

命令トレース エンコーダーは完全に同期する必要があります。

- リセット後または停止から再開した後にトレースされた最初の命令の場合。
- 命令がトレースされ、前の命令がトレースされなかった場合。
- 命令が割り込みサービス ルーチンまたは例外ハンドラの最初の場合。
- 長期間経過した後。

2.1.6 トレースの終了

何らかの理由でトレースが停止した場合は、最終的なトレース命令のアドレスを出力する必要があります。

トレースが停止する理由の例を次に示します。

- ハートは停止する可能性があります(デバッグモードに入ります)。
- ハートはリセットされる可能性があります。
- エンコーディングは停止する可能性があります(たとえば、トレースオフトリガー経由で - セクション 3.2.4 を参照してください)。
- エンコーダーによって実装されたフィルター処理機能の一致条件が満たされなくなる可能性があります。
- エンコーダが無効になっている可能性があります。

2.2 オプションおよびランタイム構成可能モード

命令トレース エンコーダーは、複数のトレース モードをサポートする場合があります。デコーダが着信パケットを正しく処理するには、現在のアクティブな構成を知らせる必要があります。

構成は、エンコーダー構成が変更されるたびにエンコーダーによって発行されるパケットによって報告されます。

このようなモードの一般的な例を次に示します。

- デルタ・アドレス・モード: プログラム・カウンターの不連続性は、絶対アドレス値ではなく差分としてエンコードされます。
- フルアドレスモード: プログラムカウンタの不連続性は、絶対アドレス値としてエンコードされます。
- 暗黙的な例外モード: 例外の宛先アドレス（例外トラップのアドレス）はデコーダによって認識され、トレースでエンコードされないと見なされます。
- 順次推定可能なジャンプモード: 2 つの命令の組み合わせ効果を考慮することで、間接ジャンプのターゲットを推測できます。
- 暗黙的な戻りモード: 関数呼び出しの宛先アドレスは呼び出し履歴から派生し、トレースでエンコードされません。
- ブランチ予測モード: エンコーダーブランチ予測器（およびデコーダ内の同一のコピー）によって正しく予測される分岐は、取られた/非取り出しとしてエンコードされず、より効率的なブランチ数としてエンコードされます。
- ジャンプターゲットキャッシュモード: 推論できないジャンプターゲットのアドレスを報告するのではなく、最近のジャンプターゲットをキャッシュし、代わりにキャッシュエントリインデックスを報告することで効率を向上させることができます。

モードには、関連付けられたパラメーターを指定できます。詳細については、表 [7.1](#) を参照してください。

すべてのモードは、サポートする必要があるデルタ・アドレス・モードを除いてオプションです。

2.2.1 デルタアドレスモード

関連パラメータ: なし

デルタ・アドレス・モードでは、アドレスは、現在の命令の実際のアドレスと、アドレスを含む前のパケットで報告された命令の実際のアドレスとの差としてエンコードされます。この差分エンコードは、完全なアドレスよりも少ないビットを必要とするため、トレース圧縮の効率が高くなります。

2.2.2 フルアドレスモード

関連パラメータ: なし

完全アドレス モードでは、トレース内のすべてのアドレスが差分形式ではなく絶対アドレスとしてエンコードされます。この種のエンコードは常に効率が悪くなりますが、ソフトウェア デコーダ開発者にとって便利なデバッグの手助けになります。

2.2.3 暗黙的な例外モード

関連パラメータ: なし

RISC-V 特権 ISA 仕様では、例外ハンドラーのベース アドレスを *utvec/stvec/mtvec* CSR レジスタに格納します。一部の RISC-V 実装では、下位アドレス ビットは、*CSR* レジスタの原因である *ucause/scause/m* に格納されます。

デフォルトでは、例外または割り込みが発生すると、**tvec* 値と **cause* 値の両方が報告されます。

暗黙の例外モードでは、トレースから **tvec* (トラップ・ハンドラー・アドレス) を省略するため、効率が向上します。

このモードは、デコーダーが例外の原因からトラップ ハンドラのアドレスを推測できる場合にのみ使用できます。

2.2.4 順次推論ジャンプモード

関連パラメータ: *sijump_p*。

デフォルトでは、間接ジャンプのターゲットは常に、不連続性の PC と見なされます。ただし、ジャンプターゲットを指定するレジスタが定数でロードされた場合、状況によっては、そのレジスタは推論可能であると考えることができます。hart は、連続して推論可能なターゲットを持つジャンプを識別し、この情報をエンコーダーに個別に提供する必要があります。ジャンプを推論不能として扱うかどうかの最終決定は、エンコーダーによって行う必要があります。デコーダがジャンプ先を推測できるようにするには、定数ロードとジャンプの両方をトレースする必要があります。順次推論可能なジャンプを構成する内容の詳細については、セクション 3.1.1 を参照してください。

2.2.5 暗黙的な戻りモード

関連するパラメーター: *call_counter_size_p*, *return_stack_size_p*, *itype_width_p*, *itype_width_p*。

関数の戻り値は通常は間接的なジャンプですが、動作の良いプログラムは、標準の呼び出し規約を使用して関数が呼び出されたプログラムのポイントに戻ります。これらのプログラムでは、リターンの宛先アドレスを明示的に通知することなく、実行パスを判別することができます。暗黙的な戻りモードでは、トレース エンコーダの効率が大幅に向上します。

戻り値は、関連付けられたコールが以前のパケットすでに報告されている場合に限り、推論可能として扱うことができます。エンコーダーは、この場合に備える必要があります。これは、カウンタを使用して、トレースされる入れ子になった呼び出しの数を追跡することによって実現できます。カウンタは呼び出しで増分し (テールコールは行いません)、リターンのデクリメント(定義については 3.1.1 を参照)。カウンタはオーバーまたはアンダーフローを行わないし、同期パケットが送信されるたびに 0 にリセットされます。リターンは推論可能として扱われ、カウントがゼロでない場合(つまり、関連付けられたコールが以前のパケットすでに報告されている)場合、トレース パケットは生成されません。

このようなスキームは低成本であり、プログラムが「行き届いた」限り機能します。エンコーダーは、戻りアドレスが実際に関連付けられた呼び出しに続く命令のアドレスであることをチェックしません。したがって、リターン アドレスを変更するプログラムは、この最小限の実装ではこのモードを使用してトレースすることはできません。

また、エンコーダーは、期待されるリターン アドレスのスタックを維持し、実際のリターン アドレスが予測に一致する場合のみ、リターンを推論可能として扱うことができます。これはすべてのプログラムに対して完全に堅牢ですが、実装にはよりコストがかかります。この場合、リターン アドレスが予測と一致しない場合は、現在スタック上にあるリターン アドレスの数と共に、パケットを通じて明示的に報告する必要があります。これにより、デコーダはどのリターンが報告されているかを判断できます。

2.2.6 分岐予測モード

関連パラメータ: *bpred_size_p*。

分岐予測なしで、各実行された分岐の結果は分岐マップに格納されます：各分岐の取得/非取得状態が時系列で格納されるビット ベクトル。

このエンコードはブランチあたり 1 ビットで効率的ですが、それでもトレース パケットの量が比較的大きくなる場合があります。例えば：

- 推論できないジャンプを含むコードのタイトなループを実行しています。ループの各反復はブランチ マップにビットを追加します。
- アイドル状態のループに座って割り込みを待っています。これは、何の興味も実際には起こっていないときに大量のトレースを生成します！
- 一部の実装では、アイドル ループでスピンするブレークポイント。

エンコーダ内に分岐予測変数を追加することで、大幅なコーディング効率を得ることができます。エンコーダとデコーダの同期を保つためには、同じ動作の予測変数をデコーダ ソフトウェアに実装する必要があります。

予測変数は、2 つの *bpred_size_p* エントリの参照テーブルを構成する必要があります。各エントリは、*bpred_size_p* 命令アドレスのビット *bpred_size_p :1* でインデックス付けされます（圧縮された命令が圧縮されていない場合は *bpred_size_p +1:2 bpred_size_p* サポートされている場合）、それぞれに 2 ビット予測状態が含まれています。

- 00: 予測が行われず、予測が失敗した場合は 01 に移行します。
- 01: 予測が取られていない、予測が成功した場合は 00 に移行、それ以外の 11;
- 11: 予測が実行され、予測が失敗した場合は 10 に移行します。
- 10: 予測を取り、予測が成功した場合は 11 に移行し、それ以外は 00 に移行する。

MSB は予測結果、LSB は最新の実際の結果を表します。予測値が変化するには、予測が 2 回失敗する必要があります。

同期パケットが送信されると、ルックアップ テーブルエントリは 01 に初期化されます。

gShare 予測変数(ヘンサーとパターソンを参照)などの他の予測変数を考慮する必要があります。さまざまなルックアップテーブルのサイズと予測アルゴリズムの利点を判断するには、さらにいくつかの実験が必要です。

2.2.7 ジャンプターゲットキャッシュモード

関連パラメーター: *cache_size_p*。

デフォルトでは、推論不能ジャンプのターゲットアドレスは、通常は差分形式でトレースに出力されます。同じ関数が繰り返し呼び出された場合(たとえばループ内)、同じアドレスが繰り返し出力されます。

エンコーダーにジャンプターゲットキャッシュを追加することで、効率向上を得ることができます。エンコーダとデコーダの同期を保つには、同じ動作のキャッシュをデコーダソフトウェアに実装する必要があります。小さなキャッシュでも大幅に改善されます。

キャッシュは、*cache_size_p* 命令アドレスを含むことができる 2 つの *cache_size_p* エントリで構成されます。この関数は直接マップされ、各エントリは *cache_size_p* 命令アドレスのビット *cache_size_p :1* でインデックス付けされます(圧縮命令がサポートされていない場合は +1:2 を *cache_size_p*)。

各推論不能ジャンプターゲットは、まずキャッシュ内のインデックスにあるエントリと比較されます。キャッシュ内で見つかった場合、インデックス番号はターゲットアドレスではなくトレースされます。キャッシュ内に見つからない場合は、そのインデックスのエントリが現在の命令アドレスに置き換えられます。

同期パケットが送信されると、キャッシュエントリはすべて無効になります。

第3章

3. エンコーダーインターフェイスへのハート

3.1 インターフェイスの要件

このセクションでは、RISC-V のハートからトレース エンコーダーに渡す必要がある情報について一般的に説明し、必須の内容とオプションの内容を区別します。

以下の情報は必須です。

- 廃止される命令の数。
- 例外または割り込みがあったかどうか、そしてそうであれば、原因(*原因/原因/mcause* CSR から)とトラップ値(*utval/stval/mtval* CSR から)
- RISC-V のハートの現在の特権レベル。
- 以下の場合の廃止された指示の *instruction_type*。
 - ソース コードから推測できないターゲットを使用してジャンプします。
 - 取られた枝と非取り出された枝;
 - 例外または割り込み(**ret*命令) から戻ります。
- *instruction_address* の対象:
 - *cannot* ソース コードから推測できないターゲットを使用してジャンプします。
 - *cannot* ソース コードから推測できないターゲット(ジャンプ先とも呼ばれる)を使用してジャンプした直後に、命令が終了しました。
 - 取られた枝と非取り出された枝;
 - 最後の命令は、例外または割り込みの前に終了しました。
 - 最初の命令は、例外または割り込みの後に廃止されました。
 - 最後の命令は、特権変更の前に廃止されました。
 - 最初の命令は、特権の変更後に終了しました。

次の情報はオプションです。

- コンテキスト情報:
 - コンテキストおよび/またはハート ID。
 - コンテキストが変更されたときに実行するアクションの種類。
- 以下 *instruction_type* の手順の *instruction_type*。
 - *cannot* ソースコードから推論できないターゲットを持つ呼び出し。
 - *can* ソースコードから推論できるターゲットを持つ呼び出し。
 - ソースコードから推論できないターゲット *cannot* を使用したテールコール。
 - ソースコードから推論できるターゲット *can* を使用したテールコール。
 - *cannot* ソースコードから推論できないターゲットを返します。
 - *can* ソースコードから推論できるターゲットを返します。
 - コルーチンスワップ;
 - ソースコードから推測できないターゲットを持つ上記の分類のいずれにも適合しないジャンプ。
 - ソースコードから推測できるターゲットを持つ上記の分類のいずれにも適合しないジャンプ。
- コンテキストがサポートされている場合、次の *instruction_address* では:
 - コンテキスト変更前に最後の命令が廃止されました。
 - コンテキストの変更後に最初の命令が廃止されました。
- ジャンプターゲットが連続して推論可能かどうか。

必須情報は、「[第 6 章](#)」で概説されているブランチトレースアルゴリズムを実装するために最低限必要な情報です。オプションの情報は、代替または改善されたトレースアルゴリズムを容易にします。

- 暗黙的な戻りモード（セクション [2.2.5](#) を参照）では、入れ子になった関数呼び出しの数を追跡するエンコーダーが必要です。
- 基本的なコードプロファイリングに役立つ単純なアルゴリズムは、ターゲットが推論可能かどうかに関係なく、関数呼び出しと戻り値のみを報告します。
- 分岐予測手法を使用して、特にループの場合はエンコーダの効率をさらに向上させることができます（セクション [2.2.6](#) を参照）。この場合、エンコーダはすべてのブランチのアドレスを認識する必要があります。
- ターゲットをレジスタに読み込む前の命令とジャンプの両方がトレースされている場合、推論不可能なジャンプは、トレース出力で報告する必要がないとして扱うことができます。

3.1.1 ジャンプ分類とターゲット推論

ジャンプは、**推論可能** または**推論不能** に分類されます。推定可能なジャンプは、バイナリ実行可能ファイルまたはその表現(例えば ELF)から推測することができるターゲットを有します。この仕様の目的のために、次の厳密な定義が適用されます。

ジャンプのターゲットがジャンプオペコード内に埋め込まれた定数を介して供給される場合、それは**推論可能** と分類されます。推測できないジャンプは *inferable*、定義上**推論不可能** です。

しかし、上記の定義では推論不可能として分類されているにもかかわらず、命令のペアを考慮することによってバイナリ実行可能ファイルから推測することができるいくつかのジャンプターゲットがあります。

具体的には、次の方法で指定されるジャンプターゲット

- *lui* または *c.lui* (定数を含むレジスター)
- *auipc* (PC からの定数オフセットを含むレジスタ)。

このようなジャンプターゲットは、一対の命令が連続して破棄された場合(すなわち、*auipc*, *lui*, *lui* または *c.lui* がジャンプの直前に)連続して**推論可能** に分類されます。注: 指示が連続して廃止されるという制限は、ハートとエンコーダ間の追加のシグナリングを最小限に抑えるために必要であり、連続実行が標準になることが予想されるため、トレース効率への影響は最小限に抑えられる必要があります。順次に推定可能なジャンプのサポートはオプションです。

ジャンプは、推奨される呼び出し規約に従って、オプションでさらに分類することができます。

- 呼び出し:
 - *jal* x1;
 - *jal* x5;
 - *jalr* x1, rs ここで rs != x5;
 - - *jalr* x5, rs ここで rs != x1;
 - - *c.jalr* rs1 ここで rs1 != x5;
 - - *c.jal*.
- テール呼び出し:
 - *jal* x0;
 - *c.j*;
 - *jalr* x0, rs ここで rs != x1 と rs != x5;
 - - *c.jr* rs1 ここで rs1 != x1 と rs1 != x5.
- リターン:
 - *jalr* rd, rs (rs == x1 または rs == x5) と rd != x1 と rd != x5;
 - - *c.jr* rs1 ここで rs1 == x1 または rs1 == x5.

- *Co(共同)ルーチンスワップ:*
 - *jalr x1,x5;*
 - *jalr x5, x1;*
 - - *c.jalr x5.*
- *その他:*
 - *jal rd* where *rd* != *x1* and *rd* != *x5*;
 - *jalr rd,rs* ここで *rs* != *x1* と *rs* != *x5* と *rd* != *x0* と *rd* != *x1* と *rd* != *x5* と *rd* != *x5*.

3.1.2 関係の between RISC-V コアとエンコーダー

エンコーダーは、単一のハートで実行される命令をエンコードすることを目的としています。

しかし、RISC-V コアには複数のハートが含まれているのは一般的です。これは、いくつかの異なる方法でコアによってサポートすることができます。

- 1 つの hart ごとにインターフェイスの別のインスタンスを実装します。各インスタンスを個別のエンコーダーインスタンスに接続して、すべてのハートを同時にトレースできます。あるいは、外部のマルチキングは、一度に 1 つの特定のハートをトレースするために、単一のエンコーダと組み合わせて使用することができます。
- コアの singe インターフェイスを実装し、コア内部でマルチプレクシングを行い、インターフェイスに接続するハートを選択します。

(細かいマルチスレッド構成で動作する複数のハートを持つ単一のエンコーダを使用することは技術的に可能ですが、スレッド切り替えの結果として頻繁に発生するコンテキストの変更はエンコード効率が非常に悪くなり、この構成はお勧めできません)。

3.2 命令インターフェース

ここでは、RISC-V hart と、前のセクションで説明した情報を伝えるトレース エンコーダーとの間のインターフェイスについて説明します。シグナルは、次のグループのいずれかに割り当てられます。

- M: 必須です。インターフェイスには、このシグナルのインスタンスを含める必要があります。
- O: オプションです。インターフェイスには、このシグナルのインスタンスが含まれる場合があります。
- MR: 必須、複製される場合があります。クロックサイクルごとに最大 N 個の分岐を取り出すことができるハートの場合、インターフェイスにはこの信号の N 個のインスタンスが含まれている必要があります。
- OR: オプションで、複製される場合があります。クロック サイクルごとに最大 N 個の分岐を取り出すことができるハートの場合、インターフェイスにはこの信号の 0 または N 個のインスタンスが含まれている必要があります。

- BR: ブロック、複製してもよい。ブロック内の複数の命令を破棄できるハートに必須。OR 単位のレプリケーション。省略した場合、インターフェイスは代わりに SR グループ信号を含める必要があります。
- SR: 単一、複製されてもよい。ブロック内の 1 つの命令のみを廃止できるハートに対して必須。OR 単位のレプリケーション（セクション 3.2.2 を参照）。省略した場合、インターフェイスには BR グループ信号を含める必要があります。

表 3.1 および 3.2 は、サイクルごとに複数の命令の廃棄を効率的にサポートするように設計されたインターフェースのシグナルをリストしています。次の説明では、複数のリタイアの動作について説明します。しかし、一度に 1 つの命令しか廃止できないハートについては、シグナリングを単純化することができ、これについてはセクション 3.2.1 で後述します。

ブロックに表示される情報は、**iaddr** から始まる連続した命令ブロックを表し、そのすべてが同じサイクルで終了しました。**itype** が **itype** 1 または 2（例外または割り込みを示す）の場合、破棄される命令の数はゼロになります。原因と **tval** は、**itype** が 1 または 2 の場合にのみ定義されます。**iretire=0** および **itype=0** の場合、他のすべてのシグナルの値は未定義です。

iretire には、このブロックで廃止された命令によって表される半単語の数が含まれ、**ilastsize** 最後の命令のサイズが大きくなります。命令カウントではなくハーフワードを使用すると、エンコーダーはブロック内のすべての命令のサイズにアクセスしなくとも、ブロック内の最後の命令のアドレスを簡単に計算できます。

itype は 3 ビットまたは 4 ビット幅にすることができます。**itype_width_p** が 3 の場合、单一のコード (6) を使用して、すべての推論不能ジャンプを示します。これは実装が簡単ですが、ジャンプ型を完全に分類する必要がある暗黙的な戻りモード（セクション 2.2.5 を参照）の使用を排除します。

iaddr は通常仮想アドレスですが、物理アドレスの場合はエンコーダーの動作には影響しません。

クロック サイクルごとに最大 N 分岐を廃棄できるハートの場合、信号グループ MR、OR、および BR または SR のいずれかを N 回複製する必要があります。シグナル・グループ 0 は最も古い命令ブロックに関する情報を表し、グループ N-1 は最新の命令ブロックを表します。このインターフェースは、1 サイクルあたりの特権、コンテキスト、例外、または割り込みのうち、複数の特権をサポートしており、グループ M および O のシグナルは複製されません。さらに、**itype** はシグナルグループの 1 つの値 1 または 2 のみを取ることができ、これは最新の有効なグループでなければなりません（つまり、番号が大きいグループの場合は **iretire** と **itype** はゼロでなければなりません）。サイクルで N 個未満の分岐が廃止された場合は、最初に番号の小さいグループを使用する必要があります。たとえば、1 つの分岐がある場合、グループ 0 のみを使用し、2 つの分岐がある場合は、1 番目の分岐までの命令をグループ 0 に報告し、2 番目の分岐までの命令をグループ 1 に報告する必要があります。

sijump はオプションであり、ハートが連続して推論可能なジャンプを検出するロジックを実装しない場合は省略できます。エンコーダーが **sijump** 入力を提供する場合、入力がこの機能を実装するハートに接続されているか、または結合されているかを示すパラメーターも指定する必要があります。これは、デコーダがハートの能力を認識できるようにするためにです。ハートがサポートしていない場合、エンコーダとデコーダで連続して推定可能なジャンプモードを有効にすると、デコーダによる正しい再構築が防止されます。

コンテキストフィールドを使用して、デコーダに追加情報を伝達できます。たとえば、次の例を参照してください。

- ソフトウェア スレッド ID。

表 3.1: 命令インターフェイス信号

信号	グループ	関数
itype [<i>itype_width_p</i> -1:0]	MIR	命令ブロックの終了タイプ (コード 6 - 15 の定義については 3.1.1 を参照): 0: ブロック内の最終命令は、他の名前付き itype コードのどれでもありません。 1: 例外。ブロック内の最後の終了した命令の後にトラップが発生した例外。 2: 割り込み。ブロック内の最後の破棄された命令の後にトラップが発生した割り込み。 3: 例外または割り込みリターン 4: 非取り出しブランチ; 5: 取られた分岐; 6: <i>itype_width_p</i> が 3 の場合は推論できないジャンプ、それ以外の場合には予約済み; 7: 予約済み; 8: 推論できない呼び出し。 9: 呼び出しを中止する。 10: 推論不可能なテールコール。 11: 推定可能なテールコール。 12: コルーチンスワップ; 13: リターン; 14: 他の推論不可能なジャンプ。 15: 他の推論可能なジャンプ。
cause [<i>cause_width_p</i> -1:0]	M	例外または割り込み原因 (原因/原因). itype =1 か 2 でない限り無視されます。
tval [<i>iaddress_width_p</i> -1:0]	M	関連するトラップ値、例えば、アドレス例外の障害のある仮想アドレスは、 utval/stval/mtval CSR に書き込まれます。将来のオプションの拡張では tval 、 tval が現在ゼロを提供している場合に補助的な情報を提供するように定義される可能性があります。 itype =1 または 2 を除き、無視されます。
priv [<i>privilege_width_p</i> -1:0]	M	このサイクルですべての命令の特権レベルが廃止されました。
iaddr [<i>address_width_p</i> -1:0]	MR	このブロックで 1 番目の命令のアドレスが廃止されました。 iretire =0 の場合は無効です
context [<i>context_width_p</i> -1:0]	O	すべての命令のコンテキストは、このサイクルで廃止されました。
ctype [<i>ctype_width_p</i> -1:0]	O	コンテキストのレポート動作: 0: 報告しない。 1: 不正確に報告する。 2: 正確に報告する。 3: 非同期不連続として報告する。
sijump	OR	itype がこのブロックが不連続で終了することを示している場合、このシグナルを 1 に設定すると、そのシグナルは順次に推定可能であり、先行する auipc 、 lui 、 lui または c.lui がトレースされている場合はエンコーダーによって推論可能として扱われる可能性があることを示します。8、10、12 もしくは 14 以外の itype コードでは無視されます。

表 3.2: 命令インターフェイス信号 - ブロックごとに複数のリタイア

信号	グループ	関数
<code>iretire[iretire_width_p-1:0]</code>	BR	命令によって表されるハーフワードの数は、このブロックで廃止されました。
<code>ilastsize[ilastsize_width_p-1:0]</code>	BR	最後に廃止された命令のサイズは、 $2^{ilastsize}$ ハーフワードです。

表 3.3: 命令インターフェイス信号 - ブロックごとの単一のリタイア

信号	グループ	関数
<code>iretire[0:0]</code>	SR	このブロックで破棄された命令の数 (0 または 1)。

- オペレーティング システムからのプロセス ID。
- CSR が書き込まれるときに、コンテキストを CSR の数値と値に設定することで、CSR の値をデコーダに伝えるために使用できます。
- 1 つのエンコーダが複数のハート(セクション 3.1.2 を参照)で共有されている場合は、ハート ID を動的に変更できる場合に、ハート ID を示すためにも使用できます。

表 3.4 に、さまざまな `ctype` 値のアクションを示します。一般的な動作は、コンテキストの変更後の最初のリタイアを除いて、このシグナルがゼロのままになることです。`ctype_width_p` は 1 または 2 であってもよい。幅の縮小オプションは、コンテキストの変更を不正確に報告する場合にのみサポートされます。

表 3.4: コンテキストタイプ `ctype` 値と対応するアクション

型	値	アクション
未報告	0	アクションなし (コンテキストを報告しない)
不正確な報告コンテキスト	1	たとえば、SW スレッドやオペレーティング システムのプロセスの変更があります。 できるだけ早い都合の良い機会に新しいコンテキスト値を報告します。 アドレス情報なしで報告され、コンテキストの正確な変更点をソースコードから推測できると仮定します(例えば CSR 書き込み)。
コンテキストを正確に報告する	2	このブロックで廃止された第 1 命令のアドレスと新しいコンテキストを報告します。 事前に報告されていない支店があった場合は、最初に報告する必要があります。 特権の変更と同じように扱われます。
コンテキストをレポートとして 非同期不連続性	3	例としては、ハートの変更があります。 直前のコンテキストで終了した最後の命令と、新しいコンテキストの 1 番目の命令を報告する必要があります。 例外として同様に扱われます。

3.2.1 単一退職の簡素化

一度に 1 つの命令しか廃止できないハートの場合、このインターフェースは表 3.1 および 3.3 に示すシグナルに単純化できます。単純化は、次のように要約できます。

- ブロック内で破棄された命令の数が 0 または 1 だけであるため、エンコーダーは最後に終了した命令のアドレスを推測するための情報を必要としません（これは、1 番目の命令と同じで、唯一の命令は廃止されました）。したがって、`ilastsize` は必要ありませんし、`iretire` は単に命令が引退したかどうかを示します。

1 サイクルあたりに廃棄できる命令の最大数をエンコーダーに示すパラメーター `retires_p iretire` の適切な解釈を選択するために、单一または複数のリタイアをサポートできるエンコーダーで使用できます `iretire`。`ilastsize` エンコーダー入力は、これらの出力を提供しない单一廃棄ハートに接続する場合は、低く結ばなければなりません。

3.2.2 代替の複数リタイアインターフェース構成

1 サイクルあたり複数の命令を廃止できるが、複数のブランチを除く場合、好ましい解決策はグループ BR、MR、OR からのシグナルの 1 つのインスタンスを使用することです。ただし、`hart` がサイクル内で N 分岐を廃止できる場合、グループ MR、OR からのシグナルの N インスタンス、および SR または BR のいずれかを使用する必要があります（各インスタンスは単一命令またはブロックのいずれかになります）。

`hart` がサイクルごとに N 命令を廃止できるが、ブランチが 1 つしかない場合、グループ SR、MR、OR からのシグナルの N インスタンスを使用して、すべての命令の明示的な詳細を提供することが許可されます（ただし、`recomm`）。

3.2.3 オプションのサイドバンド信号

表 3.5 および 3.6 で説明されているように、追加機能を提供するために、オプションのサイドバンド信号を含めてもよいです。

なお、エンコーダーで出力する必要があるユーザー定義の情報は、コンテキスト入力を介して適用する必要があります。

表 3.5: オプションのサイドバンドエンコーダ入力信号

信号	グループ	関数
impdef[<i>impdef_width_p</i>-1:0]	O	実装定義されたサイドバンド信号。これらの一般的な用途は、フィルタリング用です（第 4 章を参照してください）。
trigger[2:0]	OR	ビット 0 のパルスは、エンコーダーのトレースを開始し、他のフィルタリング条件も満たされる、さらに通知するまで続行します。ビット 1 のパルスが発生すると、エンコーダーは、さらに通知されるまでトレースを停止します。セクション 3.2.4 を参照してください。
halted	O	ハートは停止します。アサーション時に、エンコーダーは、停止前に破棄された最後の命令のアドレスを報告するパケットを出力し、続いてトレースが停止したことを示すサポート パケットを出力します。デアサーション時に、エンコーダーは再びトレースを開始し、同期パケットを開始します。
reset	O	ハートはリセット中です。エンコーダーが hart とは異なるリセットドメインにある場合、エンコーダーは、エントリの開始時にトレースが終了し、終了時に再起動したことを示すことができます。動作は停止のために上記の通りです。

表 3.6: オプションのサイドバンドエンコーダ出力信号

信号	グループ	関数
stall	O	ハートへの失速要求。一部のアプリケーションでは、この信号を使用して、トレース エンコーダーがトレース パケットを出力できない場合（パケットトランSPORT インフラストラクチャからのバックプレッシャーなど）、この信号を使用して、hart を停止させることによって実現できる無損失トレースが必要な場合があります。

3.2.4 デバッグモジュールからのトリガ出力の使用

RISC-V hart のデバッグモジュールにはトリガユニットが付いています。これは、一致制御レジスタを定義します。
(*mcontrol*) に 4 ビットを含むアクションフィールド、およびトレース用にこのフィールドのコード 2 から 5 を予約します。これらのアクションコードは、表 3.7 に示すように定義されます。実装された場合、各アクションは、トリガーを引き起こした命令と同じサイクルで、hart からの出力にパルスを生成する必要があります。

表 3.7: モジュールのデバッグ トリガのサポート (*mcontrol* アクション)

値	説明
2	トレースオン:これは、エンコーダーが提供する場合は、 trigger[0] に接続する必要があります。
3	トレースオフ:これは、エンコーダーが提供する場合は、 trigger[1] に接続する必要があります。
4	トレース通知:これは、エンコーダーが提供する場合は、 trigger[2] に接続する必要があります。これにより、エンコーダーは、ブロック内の最後の命令のアドレスを含むパケットを出力します(有効になっている場合)。

トレースオンアクションとトレースオフアクションは、いつトレースが発生するかをハートが制御する手段を提供します。Tracenotify は、指定された命令が明示的に報告されるようにするための手段を提供します。この機能は、ウォッチポイントと呼ばれることもあります。

3.2.5 リタイアメントシーケンスの例

表 3.8: 例 1 : 9 命令は 4 サイクルにわたって廃止され、2 つの分岐

引退	命令トレース ブロック
1000: <i>divuw</i> 1004: <i>add</i> 1008: <i>or</i> 100C: <i>c.jalr</i>	<i>iretire</i> =7, <i>iaddr</i> <i>iaddr</i> =0x1000, <i>itype</i> <i>itype</i> =8
0940: <i>addi</i> 0944: <i>c.beq</i>	<i>iretire</i> =3, <i>iaddr</i> <i>iaddr</i> =0x0940, <i>itype</i> <i>itype</i> =4
0946: <i>c.bnez</i>	<i>iretire</i> =1, <i>iaddr</i> <i>iaddr</i> =0x0946, <i>itype</i> <i>itype</i> =5
0988: <i>lbu</i> 098C: <i>csrrw</i>	<i>iretire</i> =4, <i>iaddr</i> <i>iaddr</i> =0x0988, <i>itype</i> <i>itype</i> =0

第 4 章

4. フィルタリング

この章の内容は参考にするだけです。

フィルター処理は、エンコーダーがトレースを生成するかどうかを制御するメカニズムを提供します。たとえば、トレースする必要があります。

- 命令アドレスが特定の範囲内にある場合。
- 1 つの命令アドレスから始まり、2 番目の命令アドレスまで続けます。
- 1 つ以上の指定された特権レベルの場合。
- 特定のコンテキストまたはコンテキストの範囲の場合。
- 指定された例外の原因または特定の **tval** 値の例外ハンドラーまたは割り込みハンドラー。
- **impdef** またはトリガー信号に適用される値に基づきます。
- 一定期間
- など。

これを実現する方法は、実装に固有のものです。

推奨される実装の 1 つは、次の機能を提供します。

- **iaddress**、**context** and および **tval** 入力に対して、さまざまな算術オプション (<,>, =, !=など) を提供するコンパレータ;
- **priv** と **cause** の入力のための複数の選択の選択;
- **interrupt** 入力と **impdef** 入力のマスクされたマッチング。
- **trigger[0]** がアサートされたときにトレースを有効にし、**trigger[0]** がアサートされるまでトレースを続行する機能 (セクション 3.2.4 を参照)。

第 5 章

5. エンコーダ出力パケットのトレース

このセクションの大部分では、トレース エンコーダから出力されるパケットのペイロードについて説明します。これらのパケットの転送に使用されるインフラストラクチャはこのドキュメントの範囲外であるため、パケットがトランスポート用にカプセル化される方法は指定されていません。ただし、カプセル化器に次の情報を提供する必要があります。

- パケットタイプ。
- パケット長（バイト単位）
- パケット ペイロード。

トランSPORT・スキームの例として、UltraSoC メッセージング・インフラストラクチャとアーム・トレス・バスの 2 つの方あります。図 5.1 は、UltraSoC インフラストラクチャに使用されるカプセル化を示しています。

- ヘッダー バイトには、ペイロード長をバイト単位で指定する 5 ビット フィールド、”フロー” を示す 2 ビット フィールド（宛先ルーティング インジケータ）、およびオプションの 16 ビット タイムスタンプが存在するかどうかを示すビットが含まれています。
- インデックス フィールドは、パケットの送信元を示します。ビット数はシステムに依存し、トレース エンコーダーによって生成される初期値は 0 です（インフラストラクチャを通じて伝達される場合に調整されます）。
- オプションの 2 バイトのタイムスタンプ。
- パケット ペイロード。

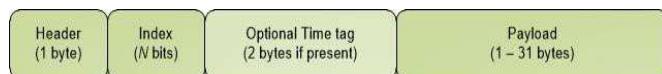


図 5.1: カプセル化されたパケット形式の例

または、ATB の場合、パケットの送信元は ATID バス フィールドによって示され、「フロー」と同等の値が存在しません。

- ペイロード長をバイト単位で指定する 5 ビット・フィールド
- オプションの 16 ビット タイムスタンプが存在するかどうかを示すビット。
- オプションの 2 バイト・タイム・スタンプ。
- パケット ペイロード。

パケットが ATB ワードにアラインメントされ始める場合、パケットの最後の拍動の ATBYTES バス フィールドを使用して有効なバイト数を示すことができます。

このセクションの残りの部分では、インフラストラクチャから独立する必要があるペイロード部分の内容について説明します。各テーブルでは、フィールドは送信順にリストされます: テーブルの最初のフィールドが最初に送信され、マルチビット フィールドが最初に LSB に送信されます。

このパケット ペイロード形式は、エンコードされた命令トレースを出力するために使用されます。3 つの異なる形式は、エンコード アルゴリズムのニーズに応じて使用されます。次の表は、ペイロードの形式を示しています - つまり、カプセル化を除く。

最高のパフォーマンスを得るために、実際のパケット長は「符号ベース圧縮」を使用して調整することができます。少なくとも、これはフォーマット 1 と 2 のパケットのアドレス フィールドに適用する必要がありますが、形式に関係なくパケット全体に適用するのが理想的です。この手法は、パケットの最上位の終端から同一のビットを除去し、それに応じてパケットの長さを調整します。この短縮パケットを受信したデコーダは、最も重要な受信ビットから符号拡張を行うことで、元の全長パケットを再構築できます。

次の表に示すペイロード長、または符号ベースの圧縮を適用した後のペイロード長がバイト全体の長さの倍数ではない場合、ペイロードは最も近いバイト境界まで符号拡張される必要があります。

最大のエンコード効率を提供する一方で、可変長パケットは、パケット間の境界がメモリに書き込まれたとき、またはパケットが通信チャネルを介してオフチップでストリーミングされる場合に発生する場所を特定するという点で、いくつかの課題を提示する可能性があります。これに対する 2 つの解決策は次のとおりです。

- 最大パケット ペイロード長が $2^N - 1$ (たとえば、N が 5 の場合、最大長は 31 バイト)、最小パケット ペイロード長が 1 の場合、パケット ペイロード内で少なくとも $2^N - 1$ バイトのシーケンスが発生しないため、少なくとも $2^N - 1$ バイトのシーケンスの後に見られる最初の非ゼロバイトはパケットの最初のバイトである必要があります。この方法は、メモリまたはデータ ストリームでのアライメントに使用できます。
- メモリに書き込まれたパケットに適した代替アプローチは、メモリを M バイトのブロック(例えば 1kbyte ブロック)に分割し、すべてのブロックの最初のバイトが常にパケットの最初のバイトになるようにメモリに書き込む方法です。つまり、パケットはブロック境界をまたがることができないため、ブロック内の最後のメッセージの終わりとブロック境界の間に 0 バイトを使用する必要があります。

5.1 フォーマット 3 パケット

フォーマット 3 パケットは、同期、コンテキストの報告、およびサポート情報に使用されます。4 つのサブフォーマットがあります。

このドキュメントでは、”同期パケット” という用語が使用されます。これは、特にフォーマット 3、サブフォーマット 0 およびサブフォーマット 1 パケットを参照します。

5.2 フォーマット 3 サブフォーマット 0 - 同期

このパケットには、デコーダが命令を完全に識別するために必要なすべての情報が含まれています。最初のトレースされた命令（その命令が例外ハンドラーの最初の命令でもある場合を除く）と再同期タイマーの満了によって再同期がスケジュールされている場合に送信されます。

表 5.1: パケット形式 3、サブフォーマット 0

フィールド名	ビット	説明
format	2	11 (同期): 同期
subformat	2	00 (開始): トレースの開始、または再同期
branch	1	アドレスが分岐命令を指し、分岐が取得された場合は 0 に設定します。命令が分岐でない場合、または分岐が取られていない場合は、1 に設定します。
privilege	<i>privilege_width_p</i>	報告された命令の特権レベル
context	<i>context_width_p</i> 、または <i>nocontext_p</i> が 1 の場合は 0	命令コンテキスト
address	<i>iaddress_width_p</i> <i>iaddress_lsb_p</i> <i>b_p</i>	完全な命令アドレス。アドレスの配置は、元のバイトアドレス <i>iaddress_lsb_p</i> 再作成するために、アドレスをシフトしたままにする必要があるによって決定されます。

5.2.1 フォーマット 3 分岐フィールド

このビットは、報告されたアドレスが分岐命令を指している場合の取得済み/取られていない状態を示します。このビットが削除され、ブランチステータスが代わりに「引き継がれ」、次の *te_inst* パケットで報告された場合、全体的な効率は若干向上します。これは考慮されたが、このアプローチが失敗するいくつかの病理学的なケースがある。たとえば、最初のトレースされた命令が分岐であり、その後に例外が続く状況を考えてみましょう。この結果、2 つの連続した命令でフォーマット 3 のパケットが生成されます。2 番目のパケットにはブランチマップが含まれていないため、1 番目のブランチのブランチステータスを報告する方法はありません。これには 2 つの問題があります。

- 同じサイクルで 2 つのパケットを生成する必要があり、エンコーダーが複雑になります。
- 図 6.1 に示すアルゴリズムが複雑になります。

5.3 フォーマット 3 サブフォーマット 1 - 例外

このパケットには、デコーダが命令を完全に識別するために必要なすべての情報も含まれています。例外の後に送信され、例外ハンドラのアドレスを報告するだけでなく、例外の原因とエラー命令のアドレスも含まれます。

暗黙的な例外モードが有効になっている場合（セクション [2.2.3](#) を参照）、アドレスは省略されます。

表 5.2: パケット形式 3、サブフォーマット 1

フィールド名	ビット	説明
形式	2	11 (同期): 同期
subformat	2	01 (例外): 例外原因とトラップ ハンドラ アドレス。
branch	1	アドレスが分岐命令を指し、分岐が取得された場合は 0 に設定します。命令が分岐でない場合、または分岐が取られていない場合は、1 に設定します。
privilege	<i>privilege_width_p</i>	報告された命令の特権レベル。
context	<i>context_width_p</i> 、ま たは <i>nocontext_p</i> が 1 の場 合は 0	命令コンテキスト。
ecause	<i>ecause_width_p</i>	例外の原因。
interrupt	1	割り込み。
address	<i>iaddress_width_p</i> <i>iaddress_lsb_p</i>	完全な命令アドレス。アドレスの配置は、元のバイトアドレス <i>iaddress_lsb_p</i> 再作成するために、アドレスをシフトしたままにする必要があるによって決定されます。
tvalepc	<i>iaddress_width_p</i>	eCAUSE が 2 で interrupt が 0 (無効な命令例外)、またはト ラップ値以外の場合は例外アドレス。

5.3.1 フォーマット 3 の tvalepc フィールド

このフィールドは、不正な命令のアドレス、またはトラップ値を報告します。これにより、すべての必要なケースに対して、障害のある命令のアドレスが報告されます。トラップ値は、ハードウェア ブレークポイント、アクセスまたはページフォルト、命令、不適切な位置合わせであるが、不正な命令（オペコードに設定されている）ではないロードまたはストアの違反命令のアドレスに設定されます。

5.4 フォーマット 3 サブフォーマット 2 - コンテキスト

このパケットにはコンテキストのみが含まれ、コンテキストが変更されたときに出力され、不正確に報告される可能性があります（表 [3.4](#) を参照）。

5.5 フォーマット 3 サブフォーマット 3 - サポート

このパケットは、デコーダを支援するためのサポート情報を提供します。次の場合に発行されます。

表 5.3: パケット形式 3、サブフォーマット 2

フィールド名	ビット	説明
format	2	11 (同期): 同期
subformat	2	10 (コンテキスト): コンテキストの変更
privilege	<i>privilege_width_p</i>	新しいコンテキストの特権レベル。
context	<i>context_width_p</i>	命令コンテキスト。

- トレースが有効または無効になっています。
- 動作モードが変わります。
- 1つ以上のトレースパケットを送信できません（たとえば、パケットトランSPORTインフラストラクチャからのバックプレッシャーが原因で送信されます）。

options フィールドは、実装固有の個々のビットのセット（エンコーダーでサポートされるオプションモードごとに1つずつ）に置き換える必要があるプレースホルダーです。

5.5.1 フォーマット 3 サブフォーマット 3 qual_status フィールド

トレースが終了すると、エンコーダーは最後にトレースされた命令のアドレスを報告し、フォーマット 3、サブフォーマット 3（サポート情報）パケットを使用してこれに従います。トレースが終了したことを示す 2つのコードが用意されています：**ended_rep** と **ended_upd**。これは、セクション 5.6.2 で詳細に説明されているのとまったく同じあいまいなケースに関連 5.6.2 しており、原則として、そのセクションで説明されているメカニズムを使用して、最後にトレースされた命令がループラベルにあるときにあいまいさを解消することができます。しかし、このメカニズムは、フォーマット 1/2 パケットを作成する際に、フォーマット 3 パケットが次の命令から生成されることを知ることに依存します。これは、エンコードアルゴリズムが 3 段階のパイプを使用して、前の命令、現在の命令、および次の命令にアクセスするためです。ただし、次の命令が特権の変更または例外であるというデコードは簡単ですが、次の命令がフィルタリング基準を満たしているかどうかを判断する方がはるかに複雑であり、この情報は通常、少なくともパイプラインステージを追加せずに利用できません。これは、異なるメカニズムが必要であることを意味し、トレースが終了したことを示す 2つのコードを持つことによって提供されます。

- ended_rep** は、トレースが終了していない場合、前のパケットが発行されなかつことを示します。
- ended_upd** は、2番目のループ反復でループラベルを実行した後にトレースが停止したことを意味する、非発生性の PC の不連続性のために、上記のパケットがとにかく発行されたことを示します。

エンコーダー実装がフィルタ結果に早期アクセスでき、最後の修飾命令が、異常な PC の不連続性に続く命令でもあるときに、デザイナーが **updiscon** ビットを使用することを選択した場合は、常に **ended_rep** を使用して修飾の喪失を示す必要があります。

表 5.4: パケット形式 3、サブフォーマット 3

フィールド名	ビット	説明
format	2	11 (同期): 同期
subformat	2	11 (サポート): デコーダのサポート情報
enable	1	エンコーダが有効かどうかを示します
encoder_mode	N	トレース アルゴリズムを識別します。 詳細とビット実装の数依存。現在分岐トレースは定義された唯一のモードで、値 0 で示されます。
qual_status	2	資格の状態を示します 00 (no_change): フィルター修飾 01 (ended_rep): 最後の資格取得命令を示すために明示的に送られた <i>te_inst</i> の前の <i>te_inst</i> が資格取得を終了しました。 10: (trace_lost): 1 つ以上のパケットが失われました。 11 : (ended_upd): 資格が終了し、最後の修飾命令でなくても、アップディスクのために <i>te_inst</i> が送信された場合)
options	N	すべてのランタイム構成ビットの値 ビット数および実装依存の定義。例は次のようになります。 <ul style="list-style-type: none"> - 「順次推定ジャンプ」連続的に推論可能なジャンプのターゲットを報告しない - '暗黙的な戻り' 関数の戻り値を報告しない - '暗黙的な例外' トラップ ベクトルが e から <i>te_inst</i> 決定できる場合、フォーマット 3、サブフォーマット 1 <i>te_inst</i> パケットからアドレスを除外します。 - '分岐予測' 分岐予測変数が有効 - 'ジャンプターゲットキヤッシュ' ジャンプターゲットキヤッシュが有効 - 「完全なアドレス」常に完全なアドレスを出力(SW デバッグオプション)

5.6 フォーマット 2 パケット

このパケットには命令アドレスのみが含まれ、命令のアドレスを報告する必要があり、報告されていないブランチ情報がない場合に使用されます。完全アドレス モードが有効になっていない限り、アドレスは差分形式です（セクション 2.2.2 を参照）。

表 5.5: パケット形式 2

フィールド名	ビット	説明
format	2	10 (addr のみ): 差分アドレスとブランチ情報なし
address	<i>iaddress_width_p</i> <i>iaddress_lsb_p</i>	差動命令アドレス。
notify	1	このビットの値が address の MSB と異なる場合、このパケットは、trigger[2] を介して通知が要求されたため、不連続のターゲットではない命令を報告していることを示します（セクション 3.2.4 を参照）。
updiscon	1	このビットの値が notify と異なる場合、このパケットは、不变の不連続性に続く命令を報告しており、例外、特権変更、または再同期の前の命令でもあることを示します（つまり、フォーマット 3 te_inst が直ちに続きます）。
irreport	1	このビットの値が updiscon と異なる場合、このパケットが次のいずれかの命令を報告していることを示します。リターンの後に続くアドレスは、implicit_return リターンアドレススタックの一番上にある予測リターンアドレスと異なるため、または現在のアドレススタックの深さまたはネストされたコールカウントを報告する必要があるため、例外、割り込み、特権の変更または再同期の前に最後に廃止された。
irdepth	<i>return_stack_size_p</i> + <i>(return_stack_size_p > 0 ? 1 : 0) + call_counter_size_p</i>	irreport の値が updiscon と異なる場合、このフィールドは、リターンアドレス スタックのエントリ数（失敗したりターンのエントリ番号）またはネストされた呼び出し回数を示します。 irreport が updiscon と同じ値の場合、このフィールドのすべてのビットも updiscon と同じ値になります。

5.6.1 フォーマット 2 通知フィールド

このビットは、ほとんどの場合、アドレスフィールドの MSB と同じ値を取るため、エンコード効率に影響を与えないように圧縮するようにエンコードされます。通知要求の結果としてアドレスが報告される場合をカバーするために必要です。input to 1.

5.6.2 フォーマット 2 通知フィールドとアップディスコンフィールド

これらのビットは、ほとんどの場合、パケット内の前のビットと同じ値を取ることによって、効率に影響を与えることなく、圧縮するようにエンコードされます(通知は通常、アドレスフィールドの MSB と同じ値 **address** であり、**updiscon** は通常通知と同じ値です **notify**)。そうでなければ、デコードソフトウェアが明確にプログラムの実行を再構築することができない病理学的なケースをカバーするために必要とされます。次のコードフラグメントを考えてみます。

```

looplabel - 4: opcode A
looplabel : opcode B
looplabel + 4: opcode C
:
looplabel + N: JALR # looplabel に飛ぶ

```

これは、次の反復に戻る間接的なジャンプを持つループです。これは、不連続性の不連続性であり、形式 1 または 2 パケットを介して報告されます。ただし、ループへの最初のエントリは、looplabel - 4 の命令からのフォールスルーであり、明示的に報告されません。これは、プログラムの実行パスを再構築するときにループラベルアドレスが 2 回発生することを意味します。先の命令は、不連続性を引き起こす可能性のある命令ではないため、デコーダは、最初にループラベルに達した時点で、これが実行の終了ではないかを判断できるようになります。したがって、実行パスは、それが *JALR* に到達するまで再構築を続けることができますが、ループラベルの *opcodeB* が最終的に廃止された命令であると推測することができます。ただし、この方法が機能しない状況があります。たとえば、ループラベル + 4 で例外が発生した場合を考えます。この場合、デコーダは、エンコーダからの追加情報がない場合、1 番目または 2 番目のループの反復中に発生したかどうかを判断できません。これが **updiscon** の目的です。詳細は次のとおりです。

考慮すべきシナリオは 4 つあります。

1. コードは最初のループ反復の最後まで実行され、エンコーダーは、1/2 形式を使用して、フォーマット 1/2 を使用してループラベルを報告し、ループの 2 番目のパスを実行します。この場合、**updiscon == notify** です。次のパケットは 1/2 形式になります。
2. コードは、最初のループ反復の最後まで実行され、ループラベル + 4 で 2 回目の反復で例外、特権の変更、または再同期が発生します。この場合、エンコーダは *JALR* に続いてフォーマット 1/2 で **looplabel** を **updateiscon == !notify** で報告し、次のパケットはフォーマット 3 になります。
3. ループラベルの最初の実行の直後に例外が発生します。この場合、エンコーダーは、フォーマット 0/1/2 を使用してループラベルを報告し、**updiscon == notify** を指定し、次のパケットはフォーマット 3 です。
4. **hart** は、ループラベルでの命令の廃止をエンコーダーに通知するように要求します。この場合、エンコーダーは、ループラベルの最初の実行を **notify ==** で報告します。**address [MSB]** とそれ以降の実行は、**notify == address[MSB]** (これらは、とにかく *JALR* の結果として報告されたであろうため) を伴います。

デコーダの観点からこれを見ると、デコーダはループ(looplabel)内の第 1 命令のアドレスを報告するフォーマット 1/2 を受け取ります。最後に報告されたアドレスからの実行パスの後にループラベルに到達します。ループラベルの前に、不連続性が発生しないため、**notify** と **updiscon** の値を考慮に入れる必要があります、最終的に廃止された命令に達したかどうかを判断するために次のパケットを待つ必要があります。

- **updiscon==!notify** の場合が、ケース 2 を示します。デコーダは、ループラベルが 2 回目に検出されるまで続行する必要があります。
- **updiscon == notify** の場合、デコーダはケース 1 と 3 をまだ区別できず、次のパケットを待つ必要があります。
 - 次のパケットがフォーマット 3 の場合、これはケース 3 です。デコーダは既に正しい命令に達しています。
 - 次のパケットがフォーマット 1/2 の場合、これはケース 1 です。デコーダは、ループラベルが 2 回目に検出されるまで続行する必要があります。
- **通知 == !address [MSB]** は、ケース 4、1 番目の反復を示します。デコーダが正しい命令に達しました。

この例ではループラベル + 4 で例外を使用していますが、ループラベル + 4 の形式 3 を引き起こす可能性のあるものは、特権の変更または再同期タイマーの満了という同じ動作になります。また、ループラベルが最後にトレースされた命令である場合にも発生する可能性があります（トレースが何らかの理由で無効になっていたため）。この点の詳細については、セクション 5.5.1 を参照してください。

注: アドレスが報告され、それが不連続性の次の命令でない場合は常に、**notify** ビットのみを実装し、**address[MSB]** の逆に設定することで、正しいデコーダの動作を実現することができます。しかし、例外、割り込み、または再同期の前にフォーマットを出力する場合、これは **address[MSB]** とは異なる時間を **notify** する必要があったため、これははるかに効率が悪かったでしょう（この命令が推論できないジャンプのターゲットである可能性が低いため）。2 つのビットを別々に使用すると、優れた圧縮が行われます。

5.6.3 形式 2 のレポートと詳細

これらのビットは、ほとんどの場合、**updiscon** フィールドと同じ値を取るため、エンコード効率に影響を与えないように圧縮するようにエンコードされます。**implicit_return** モードが有効になっている場合、エンコーダーは、単純なカウント **call_counter_size_p** (**call_counter_size_p** 非ゼロ) または予測リターン アドレスのスタック（ゼロ以外）として、トレースされた入れ子になった呼び出しの数を追跡します(**return_stack_size_pnon-0**)。

予測された戻りアドレスのスタックが実装されている場合、予測された戻りアドレスは実際の戻りアドレスと比較され、予測ミスが発生した場合は、**irreport** が逆の値に設定された **te_inst** パケットが **updiscon** に生成されます。

場合によっては、パケットが例外、割り込み、特権変更、または再同期の前に最後の命令を報告している場合は、現在のスタックの深さまたはコール数を報告する必要があります。懸念される 2 つのケースがあります。

- 報告されたアドレスが戻り値の後の命令であり、誤って予測されていない場合、エンコーダーは現在のスタックの深さまたは呼び出し回数を報告する必要があります（ゼロ以外）。これを行わない場合、デコーダは、最も外側の入れ子になった呼び出しから報告されたアドレスに遭遇するまで、実行パスの後を追って実行しようとします。

- 報告されたアドレスが戻り値の後の命令でない場合、エンコーダーは、次の場合を除き、現在のスタックの深さまたは呼び出し回数を報告する必要があります。
 - 最後の呼び出し以降、戻りはありません（この場合、デコーダーは、最も内側の呼び出しで正しく停止します）
 - 最後のリターン以降、報告されていないブランチが少なくとも 1 つありました（この場合、デコーダは未処理のブランチがないコードで正しく停止します）。

これを行わない場合、デコーダは報告されたアドレスに遭遇するまで実行パスに従い、ほとんどの場合、これが正しいポイントになります。ただし、報告されたアドレスは実行パスで複数回発生するため、再帰関数ではこれは保証されません。

5.7 フォーマット 1 パケット

このパケットにはブランチ情報が含まれ、ブランチ情報を報告する必要がある場合（たとえば、ブランチ マップがいっぱいであるため）、または命令のアドレスを報告する必要があり、以前のパケット以降に少なくとも 1 つのブランチが存在する場合に使用されます。含まれている場合、アドレスは、完全アドレスモードが有効になっていない限り、差分形式です（セクション [2.2.2](#) を参照）。

5.7.1 フォーマット 1 アップディスクフィールド

セクション [5.6.2](#) を参照してください。

5.7.2 フォーマット 1 branch_map フィールド

ブランチ マップがいっぱいになると、そのマップを報告する必要がありますが、ほとんどの場合、アドレスを報告する必要はありません。これは、**branches** を 0 に設定することで示されます。例外は、最終分岐の直前の命令によって不連続性が発生し、その場合は **branches** が 31 に設定される場合です。

サイズ(1, 3, 7, 15, 31)の選択は効率損失を最小にするように設計されている。レポートする分岐の数が **branch_map** フィールドの選択されたサイズより少ないため、平均していくつかの「無駄な」ビットが存在します。テーパーされたサイズのセットを使用すると、短いパケットの場合、無駄なビットの数は平均して少なくなります。アップディスク間の分岐の数がランダムに分散されている場合、ブランチカウントが大きいパケットを生成する確率は低くなり、パケットが長い場合の無駄が増えると全体的な影響が少なくなります。さらに、パケットが生成される速度は低いブランチ 数の場合は高くなる可能性があるため、このケースの無駄を減らすことで、最も重要な時に全体的な帯域幅が向上します。

表 5.6: パケット形式 1 - アドレス、ブランチ マップ

フィールド名	ビット	説明
format	2	01 (差分): 分岐情報を含み、差分アドレスを含めることができます。
branches	5	有効なビット branch_map 。 branch_map のビット branch_map は、次のようにして決定されます。 0: (この形式では発生しません) 1:1 ビット 2-3:3 ビット 4-7:7 ビット 8-15:15 ビット 16-31: 31 ビット たとえば、ブランチ = 12、 branch_map は 15 ビット長、12 LSB は有効です。
branch_map	branches フィールドによって決定されます。	分岐が取られるかどうかを示すビットの配列。 ビット 0 は、実行された最も古い分岐命令を表します。 各ビットについて: 0: 分岐が取られる 1: 分岐が取られていない
address	<i>iaddress_width.p</i> <i>iaddress_lsb.p</i>	差動命令アドレス。
notify	1	このビットの値がアドレスの MSB と異なる場合、このパケットは、 trigger[2] を介して通知が要求されたため、不連続のターゲットではない命令を報告していることを示します(セクション 3.2.4 を参照)。
updiscon	1	このビットの値が notify の MSB と異なる場合、このパケットは、不連続性の不連続性に続く命令を報告しており、例外、特権変更、または再同期の前の命令でもあることを示します(つまり、フォーマット 3 te_inst が直ちに続きます te_inst)。
irreport	1	このビットの値が updiscon と異なる場合、このパケットが次のいずれかの命令を報告していることを示します。リターンの後に続くアドレスは、 implicit_return リターンアドレススタックの一番上にある予測リターンアドレスと異なるため、または現在のアドレススタックの深さまたはネストされたコールカウントを報告する必要があるため、例外、割り込み、特権の変更または再同期の前に最後に廃止されました。
アイルデプス	<i>return_stack_size.p</i> + <i>(return_stack_size.p > 0 ? 1 : 0)</i> + <i>call_counter_size.p</i>	irreport の値が updiscon と異なる場合、このフィールドは、リターンアドレス スタックのエントリ数(失敗したリターンのエントリ番号)またはネストされた呼び出し回数を示します。 irreport が updiscon と同じ値の場合、このフィールドのすべてのビットも updiscon と同じ値になります。

表 5.7: パケット形式 1 - アドレスなし、ブランチ マップ

フィールド名	ビット	説明
format	2	01 (差分): 分岐情報を含み、差分アドレスを含めることができます。
branches	5	branch_map 内の有効なビット数。 branch_map の長さは branch_map 、次のように決定されます。 0: 31 ビット、パケット内のアドレスなし 1-31: (この形式では発生しません)
branch_map	31	分岐が取られるかどうかを示すビットの配列。 ビット 0 は、実行された最も古い分岐命令を表します。 各ビットについて: 0: 分岐が取られる 1: 分岐が取られていない

5.7.3 フォーマット 1 のイステータスおよびイデプスフィールド

セクション [5.6.3](#) を参照してください。

5.8 フォーマット 0 パケット

この形式は、オプションの効率拡張を目的としています。現在、正しく予測された分岐の数を報告するために、およびジャンプターゲットキャッシュインデックスを報告するための 2 つの拡張機能が定義されています。

分岐予測がサポートされ、有効になっている場合は、完全分岐マップ（形式 1 を介して）を出力するか、正しく予測された分岐の数を出力するかの選択があります。正しく予測された分岐の数が 31 以上の場合は、カウント形式が使用されます。報告されていないブランチが 31 個ある場合（つまり、ブランチマップがいっぱいである）、すべての分岐が正しく予測されていない場合、ブランチマップが出力されます。次の条件の下で、ブランチカウントが出力されます。

- 分岐は誤って予測されます。カウント値は、正しく予測された分岐の数から 31 を引いた値になります。アドレス情報は提供されません - 予測に失敗したブランチの暗黙的な情報です。
- アップディスク、割り込み、または例外では、エンコーダーがアドレスを出力する必要があります。この場合、エンコーダーはブランチ数（正しく予測されたブランチの数、マイナス 31）を出力します。
- 分岐数が最大値に達します。厳密に言えば、アドレスは、この場合のために必要とされませんが、パケット形式を上記のケースと区別する必要がないように含まれています。帯域幅への影響を無視できるほど、まれに発生します。

ジャンプターゲットキャッシュがサポートされ、有効になっており、アップディスクに続くアドレスがキャッシュ内にある場合、エンコーダーはフォーマット 0、サブフォーマット 1 を使用してキャッシュインデックスを出力できます。ただし、結果のパケットが短い場合は、エンコーダーは形式 1 または 2 を使用して差分アドレスを出力することを選択できます。これは、差動アドレスがゼロか非常に小さい場合に発生することがあります。

表 5.8: パケット形式 0、サブフォーマット 0 - アドレスなし、ブランチカウント

フィールド名	ビット	説明
format	2	00 (オプト・エクスト): オプションの効率拡張のためのフォーマット
subformat	セクション 5.8.1 を参照してください。	0 (正しく予測された分岐)
branch_count	32	正しく予測された分岐の数から 31 を引いた数。
branch_fmt	2	00 (no-addr): パケットにアドレスが含まれておらず、最後の正しい予測に続くブランチが失敗しました。 01-11: (この形式では発生しません)

5.8.1 フォーマット 0 サブフォーマットフィールド

このフィールドの幅は、サポートされるオプションの形式の数によって異なります。現在、2 つのオプション形式が定義されています（正しく予測された分岐とジャンプターゲットキャッシュ）。幅は、*f0s_width* 検出フィールドで指定されます（セクション 7.1 を参照）。複数のオプション形式がサポートされている場合、フィールド幅は 0 以外でなければなりません。ただし、オプションフォーマットが 1 つしかサポートされていない場合は、そのフィールドを省略することができ、サポートパケットのオプションフィールドからフィールドの値を推測することができます。

表 5.9: パケットフォーマット 0、サブフォーマット 0 - アドレス、ブランチカウント

フィールド名	ビット	説明
format	2	00 (オプト・エクスト): オプションの効率拡張のためのフォーマット
subformat	セクション 5.8.1 を参照してください。	0 (正しく予測された分岐)
branch_count	32	正しく予測された分岐の数から 31 を引いた数。
branch_fmt	2	10 (addr): パケットにアドレスが含まれています。 分岐命令を指している場合、ブランチは正しく予測されました。 11 (addr-fail): パケットに address 、予測に失敗したブランチを指すアドレスが含まれています。
address	<i>iaddress_width_p</i> <i>iaddress_lsb_p</i>	差動命令アドレス。
notify	1	このビットの値が address の MSB と異なる場合、このパケットは、 trigger[2] を介して通知が要求されたため、不連続のターゲットではない命令を報告していることを示します(セクション 3.2.4 を参照)。
updiscon	1	このビットの値が notify と異なる場合、このパケットは、不变の不連続性に続く命令を報告しており、例外、特権変更、または再同期の前の命令でもあることを示します(つまり、フォーマット 3 te_inst が直ちに続きます te_inst)。
irreport	1	このビットの値が updiscon と異なる場合、このパケットが次のいずれかの命令を報告していることを示します。そのアドレスが implicit_return のリターン アドレス スタックの先頭にある予測されたリターン アドレスと異なるため、リターンの後に続きます。現在のアドレス スタックの深さまたはネストされた呼び出し数を報告する必要があるため、例外、割り込み、特権の変更、または再同期の前に最後に廃止されました。
irdepth	<i>return_stack_size_p</i> + (<i>return_stack_size_p</i> > 0 ? 1 : 0) + <i>call_counter_size_p</i>	irreport の値が updiscon と異なる場合、このフィールドは、リターン アドレス スタックのエントリ数(失敗したリターンのエントリ番号)またはネストされた呼び出し回数を示します。 irreport が updiscon と同じ値の場合、このフィールドのすべてのビットも updiscon と同じ値になります。

表 5.10: パケット形式 0, サブフォーマット 1 - ジャンプターゲットインデックス, ブランチマップ

フィールド名	ビット	説明
format	2	00 (オプト・エクスト): オプションの効率拡張のためのフォーマット
subformat	セクション 5.8.1 を参照してください。	1 (ジャンプターゲットキヤツシュ)
index	<i>cache_size_p</i>	ターゲット アドレスを含むエントリのジャンプターゲットキヤツシュ インデックス。
branches	5	branch_map 内の有効なビット数。 branch_map の長さは branch_map 、次のように決定されます。 0: (この形式では発生しません) 1:1 ビット 2-3:3 ビット 4-7:7 ビット 8-15:15 ビット 16-31: 31 ビット たとえば、ブランチ = 12、 branch_map は 15 ビット長、12 LSB は有効です。
branch_map	branches フィールドによって決定されます。	分岐が取られるかどうかを示すビットの配列。 ビット 0 は、実行された最も古い分岐命令を表します。 各ビットについて: 0: 分岐が取られる 1: 分岐が取られていない
irreport	1	このビットの値が branch_map[MSB] と異なる場合、このパケットが次のいずれかの命令を報告していることを示します。 そのアドレスが、上部の予測されたリターンアドレスと異なるため、リターンの後に続きます。 返送アドレス スタック implicit_return 、 現在のアドレス スタックの深さまたはネストされた呼び出し数を報告する必要があるため、例外、割り込み、特権の変更、または再同期の前に最後に廃止されました。
irdepth	<i>return_stack_size_p</i> + <i>(return_stack_size_p > 0 ? 1 : 0) + call_counter_size_p</i>	irreport の値が branch_map[MSB] と異なる場合、このフィールドは、リターンアドレス スタックのエントリ数（失敗したリターンのエントリ番号）またはネストされた呼び出し数を示します。 irreport が branch_map[MSB] と同じ値の場合、このフィールドのすべてのビットも branch_map[MSB] と同じ値になります。

表 5.11: パケット形式 0、サブフォーマット 1 - ジャンプターゲットインデックス、ブランチマップなし

フィールド名	ビット	説明
format	2	00 (オプト・エクスト): オプションの効率拡張のためのフォーマット
subformat	セクション 5.8.1 を参照してください。	1 (ジャンプ ターゲット キャッシュ)
index	<i>cache_size_p</i>	ターゲット アドレスを含むエントリのジャンプ ターゲット キャッシュ インデックス。
branches	5	branch_map 内の有効なビット数。 branch_map の長さは branch_map 、次のようにして決定されます。 0: branch_map パケットに branch_map なし 1-31: (この形式では発生しません)
irreport	1	このビットの値が branch[MSB] と異なる場合、このパケットが次のいずれかの命令を報告していることを示します。 リターンの後に続くアドレスは、 implicit_return リターンアドレススタックの一番上にある予測リターンアドレスと異なるため、または現在のアドレススタックの深さまたはネストされたコールカウントを報告する必要があるため、例外、割り込み、特権の変更または再同期の前に最後に廃止されました。
irdepth	<i>return_stack_size_p</i> + (<i>return_stack_size_p</i> > 0 ? 1 : 0) + <i>call_counter_size_p</i>	irreport の値が branches[MSB] と異なる場合、このフィールドは、リターン アドレス スタックのエントリ数（失敗したリターンのエントリ番号）またはネストされた呼び出し回数を示します。 irreport が branches[MSB] と同じ値の場合、このフィールドのすべてのビットも branches[MSB] と同じ値になります。

(セクション 5.5 を参照してください。この規定により、既存のフォーマットの効率を低下させることなく、将来追加のフォーマットを追加することができます。

5.8.2 フォーマット 0 branch fmt フィールド

これは、アドレスが不要な場合はゼロになり、**branch_count** フィールドの上位ビットを圧縮できるようにエンコードされます。

ブランチカウントがアドレスなしで報告されるとき、それは分岐が予測に失敗したためです。ただし、アドレスがブランチカウントと共に報告される場合、パケットが不連続性、例外、または **branch_count** が 0xffff_ffff されたときに分岐が発生したためです。後者の場合、報告されたアドレスは常にブランチ用であり、前者の場合は、その場合である可能性があります。分岐の場合は、予測が満たされているかどうかについて明確にする必要があります。これが満たされた場合、報告されたアドレスは、最後に正しく予測されたブランチのアドレスになります。

5.8.3 フォーマット 0 のイステータスフィールドとイデプスフィールド

これらのビットは、ほとんどの場合、直前のビット (**updiscon**、**branch_map[MSB]** または **branches[MSB]** は特定の、パケット形式に応じて) と同じ値を取るようにエンコードされます。目的と動作は、セクション 5.6.3 で説明されているとおりです。

ジャンプターゲット キャッシュ（サブフォーマット 1）の場合、暗黙的なリターン予測に失敗したが、ジャンプターゲットキャッシュに存在するリターン アドレスがこの形式で報告されるように、これらのキャッシュが含まれます。すべての暗黙的な戻りエラーが形式 1 を使用して報告される場合、実装ではこれらを省略できます。

第 6 章

6. リファレンスアルゴリズム

本章の内容は参考程度のものです。

図 6.1 に、圧縮ブランチトレースの参照アルゴリズムを示します。図では、次の用語が使用されます。

- *te_inst*. エンコーダーによって出力されるパケットタイプの名前(第 5 章を参照)。
- *inst*. インストラクション」の略。
- *updiscon*. 不耐えがよる PC の不連続性。これは、プログラム カウンタがソース コードのみから予測できない量だけ変更される命令を識別します(**itype** 値 8、10、12、または 14)。
- *Qualified?*。フィルター条件を満たす命令が修飾され、トレースされます。
- *Branch?*。命令が分岐かどうか(**itype** 値 4 または 5) です。
- *branch map*。各ビットが分岐の結果を表すベクトル。0 は、分岐が取得されたことを示し、1 は、それが行われなかつたことを示します。
- *e_ccd*。例外がシグナル状態になっているか、コンテキストが変更され、非発生性の PC 不連続として扱う必要があります (表 3.4 を参照)。
- *ppch*。特権が変更されたか、コンテキストが変更され、正確に報告する必要があります (表 3.4 を参照)。
- *ppch_br*。上記のように、ブランチ マップは空ではありません。
- *er_ccdn*。命令の廃止と例外が同じサイクルで通知されたか、またはコンテキストが変更され、非発生性 PC の不連続性、またはコンテキスト通知として扱われるべきです (表 3.4 を参照)。
- *exc_only*。例外は同時退職なしで合図された;
- *cci*。不正確に報告される可能性がある コンテキストの変更 (表 3.4 を参照);
- *rep_br*。完全なブランチ マップまたは誤予測によるレポート分岐。

- *branches*。検出されたが、まだデコーダに報告されていない分岐の数。
- *pbc*。正しく予測された分岐数（分岐予測変数が無効か存在しない場合は常にゼロ）。
- *resync count*。同期パケットの送信が必要なタイミングを追跡するために使用されるカウンタ（セクション 6.2 を参照）。
- *max_resync*。同期パケットをスケジュールする再同期カウンタ値（セクション 6.2 を参照）。
- *resync br*。再同期カウンタが最大値に達し、ブランチ マップにまだ出力されていないエントリがあります（セクション 6.2 を参照）。

図 6.1 は、単一退職システムでのみ示されるように、命令動作による命令を示しています。コアからエンコーダへのインターフェイスでは、RISC-V のハートが複数の廃棄命令に関する情報を同時に提供することができますが、エンコーダによって生成されるパケット シーケンスは、一度に 1 つの命令を廃棄する場合と同じでなければなりません。

エンコーダ内の 3 段階のパイプラインが想定され、エンコーダーは現在、前、および次の命令を可視化します。すべてのパケットは、現在の命令に関連する情報を使用して生成されます。オレンジダイヤモンドは前の命令に基づく決定を示し、緑のダイヤモンドは次の命令に基づく決定を示し、他のすべてのダイヤモンドは現在の命令に基づいています。

さらに、エンコーダーは、わかりやすくするために図に示されていない、さらに 1 つのパケットの種類を生成できます。サポートパケット（フォーマット 3、サブフォーマット 3 - セクション 5.5 を参照）は、次の場合に送信されます。

- エンコーダーが有効または無効になっているか、その構成が変更されてエンコーダーの動作モードがデコーダに通知されます。
- 最終的な修飾命令がトレースされた後、トレースが停止したことをデコーダに通知します。
- トレース パケットが失われた場合（たとえば、パケットが書き込まれるバッファがいっぱいになる場合など）。この場合、次にスペースが使用可能になったときにバッファにロードされる最初のパケットは、サポートパケットでなければなりません。この後、トレースは同期パケットで再開されます。

注: **halted** または **reset** されたサイドバンド信号がアサートされている場合（表 3.5 を参照）、エンコーダーは非修飾命令を受け取ったかのように動作します（出力 *te_inst* 前の命令のアドレスを報告し、その後に *te_support*。）；

6.1 フォーマットの選択

2つを除くすべての場合において、パケット形式は関連付けられた決定からの「はい」結果によってのみ決定されます。

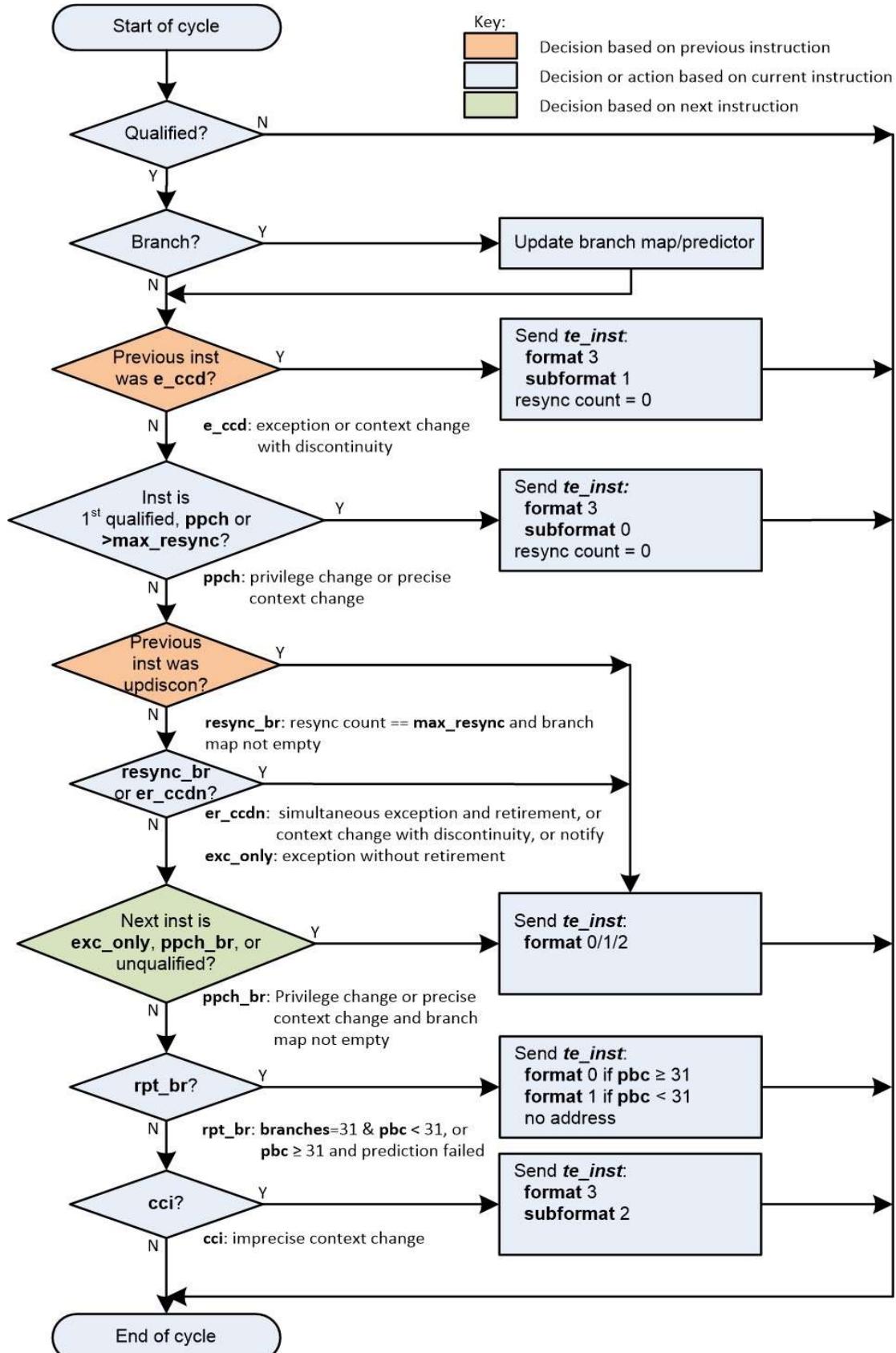


図 6.1: 命令デルタトレースアルゴリズム

46

RISC-V プロセッサトレースバージョン 1.0

(アドレスなし) の分岐情報を単独で報告する場合、形式 1 と形式 0 の間の選択は、正しく予測される分岐の数によって異なります（予測変数がサポートされていない場合、または無効になっている場合は 0 になります）。報告するブランチが少なくとも 31 個になるまで、パケットは生成されません。これらの 31 個の分岐のうち少なくとも 1 つの結果が正しく予測されなかった場合、形式 1 が使用されます。すべてが正しく予測された場合、この時点では何も出力されず、エンコーダーは正しく予測されたブランチの結果をカウントし続けます。分岐の結果のいずれかが正しく予測されないと、エンコーダーはフォーマット 0、サブフォーマット 0 パケットを出力します。セクション 5.8 も参照してください。

図の途中にある「フォーマット 0/1/2」ケースの形式を選択するには、さらに説明する必要があります。

- 正しく予測される分岐の数が 31 以上の場合、形式 0、サブフォーマット 0 が常に使用されます。
- それ以外の場合、ジャンプ ターゲット キャッシュがサポートされ、有効になっており、報告されるアドレスがキャッシュ内にある場合、通常は形式 0、サブフォーマット 1 が使用され、アドレスに関連付けられたキャッシュ インデックスが報告されます。レポートするブランチがある場合は、ブランチ情報が含まれます。ただし、エンコーダーは、パケットが短くなる場合は、(ブランチ情報の有無にかかわらず差分アドレスを含む) 同等の形式の 1 または 2 パケットを出力することを選択できます（セクション 5.8 を参照）。
- それ以外の場合は、レポートする分岐がある場合は、形式 1 が使用され、それ以外の形式は 2 です。

パケット形式 0、1、2 は、通常はアドレスが最後のフィールドになるように構成されています。アドレスを表すために必要なビット数を最小限に抑えることで、パケットの合計サイズが減少し、効率が大幅に向上します。章 5 を参照してください。

6.2 再同期

セクション 2.1.5 では、フォーマット 3 同期パケットは、「長期間」の後に出力する必要があります。これを決定する正確なメカニズムは指定されていませんが、以前の `te_inst` 同期メッセージが送信されてから、送信された `te_inst` パケットの数またはクロックサイクルの経過数をカウントするオプションがあります。

再同期が必要な場合、デコーダが履歴を必要とせずにその時点からトレースを開始できるように、形式 3 パケットを出力することが主な目的です。ただし、デコーダが既に同期されている場合は、フォーマット 3 パケットまでの実行パスをシムレスに継続して実行パスに従う必要があります。そのため、フォーマット 3 パケットを出力する前に、報告されていないブランチがある場合（フォーマット 3 にはブランチ マップが含まれていないため）、前の命令に対してフォーマット 1 パケットを出力する必要があります。再同期タイマーを超過した場合、フォーマット 3 が送信されます。この前のサイクル（再同期タイマー値に正確に達した場合）では、ブランチ マップが空でない場合、形式 1 が生成されます。

6.3 複数のリタイアに関する考慮事項

このセクションで前述したように、単一廃止システムでは、参照アルゴリズムが各廃止命令に適用されます。命令がブロック単位で破棄される場合、ブロック内の最初と最後の命令のみを考慮する必要があり、その間にある命令はすべて「面白くない」ため、エンコーダの状態には影響しません（図 6.1 を通るルートは、長方形のボックスを通過しません）。

ほとんどの場合、ブロックの最初または最後の命令（両方ではない）が興味深いので、エンコーダーはブロックから複数のパケットを生成する必要はありません。ただし、これが当てはまらない場合も少なく、エンコーダーが同じブロックから 2 つのパケットを生成する必要がある場合もあります。

たとえば、ブロック内の最初の命令は、最初のトレースされた命令である場合、パケットを生成する必要があります。ただし、ブロックが例外または割り込み(**itype**= 1 または 2) を示す場合は、ブロック内の最後の命令もパケットを生成する必要があります。

サイクルごとに複数のパケットを生成するとエンコーダが複雑になり、このような状況はまれにしか発生しなくて済むため、エンコーダ内の弹性バッファリングが推奨されます。これにより、エンコーダーがブロックから 2 つの連続したパケットを生成している間に、後続のブロックをキューに入れられます。エンコーダーは、ハートが何も報告しない場合、または **itype** = 0(エンコーダにとって面白くない)のブロックがある場合、サイクルがあるときはいつでも弹性バッファを排出できます。

連続するブロックが最初と最後の命令の両方からパケットを生成する必要がある病理学的なケースがありますが、エラスティック バッファリングは、ブロックが連続したサイクルでも入力される場合にのみ必要です。実際には、このような状況が発生するケースはほとんどありません。最悪のケースは、例外が `ecall` であり、トラップ ハンドラの最初の数個の命令で他の形式の例外または割り込みが発生する上記の例のバリエーションです。

- ブロック 1: **itype** = 1 (`ecall`)、**iretires** > 1。最初の命令（最初にトレース）からパケットを生成し、最後の命令（最後の `ecall` の前に）を生成します。
- ブロック 2: **itype** = 1 または 2 (他の例外または割り込み)、**iretires** > 0。最初の命令 (`ecall` トラップ ハンドラ) からパケットを生成し、最後の命令 (他の例外または割り込みの前に最後に) を生成します。
- ブロック 3: 最初の命令からパケットを生成する (他の例外または割り込みトラップ ハンドラー)

`ecall` はハートのフェッチユニットに知られており、予測できるので、ブロックが可能な場合があります。ブロック 1 の後にサイクルが発生する 2。ただし、他の例外または割り込みは予測できない場合、ブロック 2 と 3 の間に複数のサイクルが存在し、エンコーダーが “追いつく” ことが可能になると考えるのが妥当です。エンコーダーはこのケースを処理するのに十分な弹性バッファリングを実装し、何らかの理由でエラスティック バッファオーバーフローが発生した場合は、トレースが失われたことを示すサポート パケットを発行することをお勧めします。

第 7 章

7. パラメータと検出

このドキュメントでは、表 7.1 に示すように、バスの幅、オプション機能の有無、リソースのサイズなど、エンコーダの側面を記述するためのパラメータを定義します。

実装によっては、いくつかのパラメータが本質的に固定される場合もあれば、何らかの方法で設計に渡されるパラメータもあります。

表 7.1: エンコーダーへのパラメータ

パラメータ名	範囲	説明
<i>arch_p</i>		エンコーダが準拠しているアーキテクチャ仕様のバージョン（初期バージョンの場合は 0）。
<i>bpred_size_p</i>		分岐予測変数のエントリ数は $2^{bpred\ size\ p}$ です。エントリの最小数は 2 なので、値 0 は、分岐予測変数が実装されないです。
<i>cache_size_p</i>		ジャンプターゲットキャッシュのエントリ数は $2^{cache\ size\ p}$ です。エントリの最小数は 2 なので、値 0 は、ジャンプターゲットキャッシュが実装されないです。
<i>call_counter_size_p</i>		ネストしたコールカウンタのビット数は $2^{call\ counter\ size\ p}$ です。エントリの最小数は 2 なので、値 0 は暗黙的な戻り呼び出しカウンターが実装されないです。
<i>ctype_width_p</i>		ctype バスの幅
<i>context_width_p</i>		コンテキストバスの幅
<i>ecause_width_p</i>		例外原因バスの幅
<i>ecause_choice_p</i>		複数の選択肢を使用して一致する例外のビット数
<i>f0s_width_p</i>		フォーマット 0 の subformat フィールドの幅 <i>te_inst</i> パケット（セクション 5.8.1 を参照）。
<i>filter_context_p</i>	0 または 1	1 の場合にサポートされるコンテキストでのフィルタリング
<i>filter_excint_p</i>		例外の原因または中断に対するフィルター処理は、non_zero 時にサポートされます。サポートされるネストされた例外の数は次の値です。 2 ^{<i>filter_excint_p</i>}
<i>filter_privilege_p</i>	0 または 1	1 の場合にサポートされる特権に対するフィルター処理
<i>filter_tval_p</i>	0 または 1	1 の場合にサポートされるトラップ値に対するフィルター処理（指定された <i>filter_excint_p</i> がゼロ以外の場合）
<i>iaddress_lsb_p</i>		トレースする命令アドレスバスの LSB。1 は圧縮命令がサポートされ、2 はサポートされています。
<i>iaddress_width_p</i>		命令アドレスバスの幅。これは DXLEN と同じです。
<i>iretire_width_p</i>		iretire バスの幅
<i>ilastsize_width_p</i>		ilastsize バスの幅
<i>itype_width_p</i>		iType バスの幅
<i>nocontext_p</i>	0 または 1	1 の場合、 <i>te_inst</i> パケットからコンテキストを除外します。
<i>privilege_width_p</i>		特権バスの幅
<i>retires_p</i>		ロックごとに破棄できる命令の最大数
<i>return_stack_size_p</i>		リターンアドレススタックの of エントリ 数 は、 $2^{return\ stack\ size\ p}$ 。エントリの最小数は 2 なので、値 0 は暗黙的な戻りリストが実装されないです。
<i>sijump_p</i>	0 または 1	sijump は、連続して推論可能なジャンプを識別するために使用されます
<i>taken_branches_p</i>		iリタイア、i型等が複製される回数
<i>impdef_width_p</i>		実装定義入力バスの幅

7.1 エンコーダパラメータの検出

正しく動作するには、デコーダは、検出可能な属性の形式で、実行時にエンコーダのパラメータの一部を決定できる必要があります。これらのパラメータは、デコーダによって検出可能であるか、または既定値で固定されている必要があります（つまり、エンコーダーが特定のパラメータを検出可能にしない場合は、そのパラメータの既定値のみを実装する必要があります。表 7.2 に、必要な検出可能な属性を示します。

検出可能な属性にアクセスするには、デバッガーや監視用のハートなどの外部エンティティがエンコーダーに要求する必要があります。エンコーダーは、1つまたは複数の異なる形式で検出情報を提供します。優先形式は、トレース インフラストラクチャを介して送信されるパケットです。別の形式は、外部エンティティがエンコーダーによって維持されるレジスタまたはメモリマップスペースから値を読み取ることができるようになります。セクション 7.2 は、これを達成する方法の例を示しています。

表 7.2: 必須属性

名前	既定	パラメータマッピング
<i>arch</i>	0	<i>arch_p</i>
<i>bpred_size</i>	0	<i>bpred_size_p</i>
<i>cache_size</i>	0	<i>cache_size_p</i>
<i>call_counter_size</i>	0	<i>call_counter_size_p</i>
<i>context_width</i>	0	<i>context_width_p - 1</i>
<i>ecause_width</i>	3	<i>ecause_width_p - 1</i>
<i>f0s_width</i>	0	<i>f0s_width_p</i>
<i>iaddress_lsb</i>	0	<i>iaddress_lsb_p - 1</i>
<i>iaddress_width</i>	31	<i>iaddress_width_p - 1</i>
<i>nocontext</i>	0	<i>nocontext</i>
<i>privilege_width</i>	2	<i>privilege_width_p - 1</i>
<i>return_stack_size</i>	0	<i>return_stack_size_p</i>
<i>sijump</i>	0	<i>sijump_p</i>

使いやすさを高めるため、デコーダで直接必要としない場合でも、すべてのエンコーダのパラメータを検出可能な属性にマッピングすることをお勧めします。特に、フィルタリング機能に関連する属性。表 7.3 に、第 4 章で説明されているフィルタリングの推奨事項に関連する属性と、このドキュメントで説明されている他のパラメータに関連する属性を表 7.4 に示します。

表 7.3: オプションのフィルタ属性

名前	既定	パラメータマッピング
<i>comparators</i>	0	<i>comparators_p - 1</i>
<i>filters</i>	0	<i>filters_p - 1</i>
<i>ecause_choice</i>	5	<i>ecause_choice_p</i>
<i>filter_context</i>	1	<i>filter_context_p</i>
<i>filter_excint</i>	1	<i>filter_excint_p</i>
<i>filter_privilege</i>	1	<i>filter_privilegep</i>
<i>filter_tval</i>	1	<i>filter_tval_p</i>

表 7.4: その他の推奨属性

名前	既定	説明
<i>ctype_width</i>	1	<i>ctype_width_p - 1</i>
<i>ilastsize_width</i>	0	<i>ilastsize_width_p - 1</i>
<i>itype_width</i>	4	<i>itype_width_p - 1</i>
<i>iretire_width</i>	2	<i>iretire_width_p - 1</i>
<i>retires</i>	0	<i>retires_p - 1</i>
<i>taken_branches</i>	0	<i>taken_branches_p - 1</i>
<i>impdef_width</i>	0	<i>impdef_width_p - 1</i>

7.2 ipxact の説明例

このセクションでは、ipxact フォームで表される探索情報の例を示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<ipxact:component
    xmlns:ipxact="http://www.accellera.org/XMLSchema/IPXACT/1685-2014"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.accellera.org/XMLSchema/IPXACT/1685-2014
                        http://www.accellera.org/XMLSchema/IPXACT/1685-2014/index.xsd">
    <ipxact:vendor>UltraSoC</ipxact:vendor>
    <ipxact:library>TraceEncoder</ipxact:library>
    <ipxact:name>TraceEncoder</ipxact:name>
    <ipxact:version>0.8</ipxact:version>
    <ipxact:memoryMaps>
        <ipxact:memoryMap>
            <ipxact:name>Trace Encoder Register Map</ipxact:name>
            <ipxact:addressBlock>
                <ipxact:name>>Trace Encoder Register Address Block</ipxact:name>
                <ipxact:baseAddress>0</ipxact:baseAddress>
                <ipxact:range>128</ipxact:range>
                <ipxact:width>64</ipxact:width>

            <ipxact:register>
                <ipxact:name>discovery_info_0</ipxact:name>
                <ipxact:addressOffset>'h0</ipxact:addressOffset>
                <ipxact:size>64</ipxact:size>
                <ipxact:access>read-only</ipxact:access>
                <ipxact:field>
                    <ipxact:name>version</ipxact:name>
                    <ipxact:description>text</ipxact:description>
                    <ipxact:bitOffset>0</ipxact:bitOffset>
                    <ipxact:bitWidth>4</ipxact:bitWidth>
                </ipxact:field>
                <ipxact:field>
                    <ipxact:name>minor_revision</ipxact:name>
```

```

<ipxact:description>text</ipxact:description>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>4</ipxact:bitOffset>
<ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>arch</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>8</ipxact:bitOffset>
<ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>bpred_size</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>12</ipxact:bitOffset>
<ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>cache_size</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>16</ipxact:bitOffset>
<ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>call counter.size</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>20</ipxact:bitOffset>
<ipxact:bitWidth>3</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>comparators</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>23</ipxact:bitOffset>
<ipxact:bitWidth>3</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>context_type_width</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>26</ipxact:bitOffset>
<ipxact:bitWidth>5</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>context_width</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>31</ipxact:bitOffset>
<ipxact:bitWidth>5</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>ecause_choice</ipxact:name>
```

```

<ipxact:description>text</ipxact:description>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>36</ipxact:bitOffset>
<ipxact:bitWidth>3</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
    <ipxact:name>ecause_width</ipxact:name>
    <ipxact:description>text</ipxact:description>
    <ipxact:bitOffset>39</ipxact:bitOffset>
    <ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
    <ipxact:name>filters</ipxact:name>
    <ipxact:description>text</ipxact:description>
    <ipxact:bitOffset>43</ipxact:bitOffset>
    <ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
    <ipxact:name>filter_context</ipxact:name>
    <ipxact:description>text</ipxact:description>
    <ipxact:bitOffset>47</ipxact:bitOffset>
    <ipxact:bitWidth>1</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
    <ipxact:name>filter_excint</ipxact:name>
    <ipxact:description>text</ipxact:description>
    <ipxact:bitOffset>48</ipxact:bitOffset>
    <ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
    <ipxact:name>filter_privilege</ipxact:name>
    <ipxact:description>text</ipxact:description>
    <ipxact:bitOffset>52</ipxact:bitOffset>
    <ipxact:bitWidth>1</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
    <ipxact:name>filter_tval</ipxact:name>
    <ipxact:description>text</ipxact:description>
    <ipxact:bitOffset>53</ipxact:bitOffset>
    <ipxact:description>text</ipxact:description>
    <ipxact:bitWidth>1</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
    <ipxact:name>filter_impdef</ipxact:name>
    <ipxact:description>text</ipxact:description>
    <ipxact:bitOffset>54</ipxact:bitOffset>
    <ipxact:bitWidth>1</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
    <ipxact:name>f0s_width</ipxact:name>

```

```

        <ipxact:description>text</ipxact:description>
        <ipxact:description>text</ipxact:description>
        <ipxact:bitOffset>55</ipxact:bitOffset>
        <ipxact:bitWidth>2</ipxact:bitWidth>
    </ipxact:field>
    <ipxact:field>
        <ipxact:name>iaddress_lsb</ipxact:name>
        <ipxact:description>text</ipxact:description>
        <ipxact:bitOffset>57</ipxact:bitOffset>
        <ipxact:bitWidth>2</ipxact:bitWidth>
    </ipxact:field>
</ipxact:register>

<ipxact:register>
    <ipxact:name>discovery_info_1</ipxact:name>
    <ipxact:addressOffset>'h4</ipxact:addressOffset>
    <ipxact:size>64</ipxact:size>
    <ipxact:access>read-only</ipxact:access>
    <ipxact:field>
        <ipxact:name>iaddress_width</ipxact:name>
        <ipxact:description>text</ipxact:description>
        <ipxact:bitOffset>0</ipxact:bitOffset>
        <ipxact:bitWidth>7</ipxact:bitWidth>
    </ipxact:field>
    <ipxact:field>
        <ipxact:name>ilastsize_width</ipxact:name>
        <ipxact:description>text</ipxact:description>
        <ipxact:bitOffset>7</ipxact:bitOffset>
        <ipxact:bitWidth>7</ipxact:bitWidth>
    </ipxact:field>
    <ipxact:field>
        <ipxact:name>itype_width</ipxact:name>
        <ipxact:description>text</ipxact:description>
        <ipxact:description>text</ipxact:description>
        <ipxact:bitOffset>14</ipxact:bitOffset>
        <ipxact:bitWidth>7</ipxact:bitWidth>
    </ipxact:field>
    <ipxact:field>
        <ipxact:name>iretire_width</ipxact:name>
        <ipxact:description>text</ipxact:description>
        <ipxact:bitOffset>21</ipxact:bitOffset>
        <ipxact:bitWidth>7</ipxact:bitWidth>
    </ipxact:field>
    <ipxact:field>
        <ipxact:name>nocontext</ipxact:name>
        <ipxact:description>text</ipxact:description>
        <ipxact:bitOffset>28</ipxact:bitOffset>
        <ipxact:bitWidth>1</ipxact:bitWidth>
    </ipxact:field>

```

```

<ipxact:field>
  <ipxact:name>privilege_width</ipxact:name>
  <ipxact:description>text</ipxact:description>
  <ipxact:bitOffset>29</ipxact:bitOffset>
  <ipxact:bitWidth>2</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
  <ipxact:name>retires</ipxact:name>
  <ipxact:description>text</ipxact:description>
  <ipxact:bitOffset>31</ipxact:bitOffset>
  <ipxact:bitWidth>3</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
  <ipxact:name>return_stack_size</ipxact:name>
  <ipxact:description>text</ipxact:description>
  <ipxact:bitOffset>34</ipxact:bitOffset>
  <ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
  <ipxact:name>sijump</ipxact:name>
  <ipxact:description>text</ipxact:description>
  <ipxact:bitOffset>38</ipxact:bitOffset>
  <ipxact:bitWidth>1</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
  <ipxact:name>taken_branches</ipxact:name>
  <ipxact:description>text</ipxact:description>
  <ipxact:bitOffset>39</ipxact:bitOffset>
  <ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
  <ipxact:name>impdef_width</ipxact:name>
  <ipxact:description>text</ipxact:description>
  <ipxact:bitOffset>43</ipxact:bitOffset>
  <ipxact:bitWidth>5</ipxact:bitWidth>
</ipxact:field>
</ipxact:register>

</ipxact:addressBlock>
<ipxact:addressUnitBits>8</ipxact:addressUnitBits>
</ipxact:memoryMap>
</ipxact:memoryMaps>
</ipxact:component>

```

第8章

8. 今後の方向性

現在のフォーカスは圧縮ブランチ トレースですが、他にも便利なプロセッサ トレースが多数あります（以下の詳細は順ではありません）。これらは、現在のスコープが完了した後、将来追加される可能性のある機能として考慮する必要があります。

8.1 データトレース

トレース エンコーダーは、オフロードとストアに関する情報をオフチップ デコーダに通信するパケットを出力します。必要な帯域幅を減らすために、レポート データの値はオプションであり、アドレスとデータの両方を、有益な場合に差分にエンコードできます。これは、転送方向に関係なく、同じ転送サイズの新しい値と以前の値との差を出力する必要があります。

エンコードされていない値は、同期や他の場合に使用されます。

8.2 高速プロファイリング

このモードでは、エンコーダーは、プログラム カウンタをサンプリングできるように、プロセッサを定期的に停止する必要がある従来のプロファイリング方法に代わる非侵入的な代替手段を提供します。エンコーダーは、例外、コール、またはリターンが検出されたときにパケットを発行し、次に実行された命令(宛先命令)を報告します。オプションで、エンコーダーは現在の命令(つまりソース命令)を報告することもできます。

8.3 命令間サイクルカウント

このモードでは、エンコーダーは、連続する命令の終了間のサイクル数を報告することによって、ハートが失速している場所をトレースします。

8.4 トランスポート

現在のチャーターが満たされた後、輸送メカニズムを定義し、標準化する必要があります。これには、オーロラベースのセルデス、PCIe、イーサネットが含まれます。

第9章

9. デコーダー

このデコーダの実装では、分岐予測またはリターン アドレス スタック (*return_stack_size_p* と両方のゼロ *bpred_size_p*) が存在しないものとします。

エンコーダとデコーダの両方の C コード実装のリファレンスは、https://github.com/riscv/riscv-trace-spec/tree/master/te_codec/src にあります。

9.1 デコーダ擬似コード

```
# global variables
global      pc           # Reconstructed program counter
global      last_pc       # PC of previous instruction
global      branches = 0   # Number of branches to process
global      branch_map = 0 # Bit vector of not taken/taken (1/0) status
                         # for branches
global bool  stop_at_last_branch = FALSE # Flag to indicate reconstruction is to end at
                                         # the final branch
global bool  inferred_address = FALSE  # Flag to indicate that reported address from
                                         # format 0/1/2 was not following an uninferable
                                         # jump (and is therefore inferred)
global bool  start_of_trace = TRUE    # Flag indicating 1st trace packet still
                                         # to be processed
global      address        # Reconstructed address from te_inst messages
global      options         # Operating mode flags
global array return_stack   # Array holding return address stack
global      irstack_depth = 0  # Depth of the return address stack
```

```

# Process te_inst packet. Call each time a te_inst packet is received #
function process_te_inst (te_inst)
    if (te_inst.format == 3)
        if (te_inst.subformat == 3) # Support packet
            process_support(te_inst)
            return
        if (te_inst.subformat == 2) # Context packet
            return

        inferred_address = FALSE
        address      = (te_inst.address << discovery_response.iaddress_lsb)
        if (te_inst.subformat == 1 or start_of_trace)
            branches = 0
            branch_map = 0
        if (is_branch(get_instr(address))) # 1 unprocessed branch if this instruction is a branch
            branch_map = branch_map | (te_inst.branch << branches)
            branches++
        if (te_inst.subformat == 0 and !start_of_trace)
            follow_execution_path(address, te_inst)
        else
            pc          = address
            last_pc     = pc # previous pc not known but ensures correct
                           # operation for is_sequential_jump()
        start_of_trace = FALSE
        irstack_depth = 0

    else
        if (start_of_trace) # This should not be possible!
            ERROR: Expecting trace to start with format 3
            return
        if (te_inst.format == 2 or te_inst.branches != 0)
            stop_at_last_branch = FALSE
            if (options.full_address)
                address = (te_inst.address << discovery_response.iaddress_lsb)
            else
                address += (te_inst.address << discovery_response.iaddress_lsb)
        if (te_inst.format == 1)
            stop_at_last_branch = (te_inst.branches == 0)
            # Branch map will contain <= 1 branch (1 if last reported instruction was a branch)
            branch_map = branch_map | (te_inst.branch_map << branches)
            if (te_inst.branches == 0)
                branches += 31
            else
                branches += te_inst.branches

        follow_execution_path(address, te_inst)

```

```

# Follow execution path to reported address #
function follow_execution_path(address, te_inst)

local previous_address = pc
local stop_here      = FALSE
while (TRUE)
    if (inferred_address) # iterate again from previously reported address to
        #find second occurrence
        stop_here = next_pc(previous_address)
    if (stop_here)
        inferred_address = FALSE
    else
        stop_here = next_pc(address)
    if (branches == 1 and is_branch(get_instr(pc)) and stop_at_last_branch)
        # Reached final branch - stop here (do not follow to next instruction as
        # we do not yet know whether it retires)
        stop_at_last_branch = FALSE
        return
    if (stop_here)
        # Reached reported address following an uninferable discontinuity - stop here
        if (branches > (is_branch(get_instr(pc)) ? 1 : 0))
            # Check all branches processed (except 1 if this instruction is a branch)
            ERROR: unprocessed branches
            return
    if (te_inst.format != 3 and pc == address and !stop_at_last_branch and
        (te_inst.notify != get_previous_bit(te_inst, "notify")) and
        (branches == (is_branch(get_instr(pc)) ? 1 : 0)))
        # All branches processed, and reached reported address due to notification,
        # not as an uninferable jump target
        return
    if (te_inst.format != 3 and pc == address and !stop_at_last_branch and
        !is_uninferable_discon(get_instr(last_pc)) and
        (te_inst.updiscon == get_previous_bit(te_inst, "updiscon")) and
        (branches == (is_branch(get_instr(pc)) ? 1 : 0)) and
        ((te_inst.irreport == get_previous_bit(te_inst, "irreport")) or
        te_inst.irdepth == irstack_depth))
        # All branches processed, and reached reported address, but not as an
        # uninferable jump target
        # Stop here for now, though flag indicates this may not be
        # final retired instruction
        inferred_address = TRUE
        return
    if (te_inst.format == 3 and pc == address and
        (branches == (is_branch(get_instr(pc)) ? 1 : 0)))
        # All branches processed, and reached reported address
        return

```

```

# Compute next PC #
function next_pc (address)

local instr      = get_instr(pc)
local this_pc   = pc
local stop_here = FALSE

if (is_inferable_jump(instr))
    pc += instr.imm else if (is_sequential_jump(instr, last_pc)) # lui/auipc
followed by
                                # jump using same register
    pc = sequential_jump_target(pc, last_pc)
else if (is_implicit_return(instr))
    pc = pop_return_stack()
else if (is_uninferable_discon(instr))
    if (stop_at_last_branch)
        ERROR: unexpected uninferable discontinuity else
        pc      = address
        stop_here = TRUE
    else if (is_taken_branch(instr))
        pc += instr.imm
    else
        pc += instruction_size(instr)

if (is_call(instr))
    push_return_stack(this_pc)

last_pc = this_pc

return stop_here

# Process support packet #
function process_support (te_inst)

local stop_here = FALSE

options = te_inst.options
if (te_inst.qual_status != no_change)
    start_of_trace = TRUE # Trace ended, so get ready to start again
if (te_inst.qual_status == ended_upd and inferred_address)
    local previous_address = pc
    inferred_address  = FALSE
    while (TRUE)
        stop_here = next_pc(previous_address)
        if (stop_here)
            return
    return

```

```

# Determine if instruction is a branch, adjust branch count/map,
# and return taken status #
function is_taken_branch (instr)
    local bool taken = FALSE

    if (!is_branch(instr))
        return FALSE

    if (branches == 0)
        ERROR: cannot resolve branch
    else
        taken = !branch_map[0]
        branches-
        branch_map >> 1

    return taken

# Determine if instruction is a branch #
function is_branch (instr)

    if ((instr.opcode == BEQ)      or
        (instr.opcode == BNE)      or
        (instr.opcode == BLT)      or
        (instr.opcode == BGE)      or
        (instr.opcode == BLTU)     or
        (instr.opcode == BGEU)     or
        (instr.opcode == C.BEQZ) or
        (instr.opcode == C.BNEZ))
        return TRUE

    return FALSE

# Determine if instruction is an inferable jump #
function is_inferable_jump (instr)

    if ((instr.opcode == JAL)      or
        (instr.opcode == C.JAL) or
        (instr.opcode == C.J) or
        (instr.opcode == JALR and instr.rs1 == 0))
        return TRUE

    return FALSE

```

```

# Determine if instruction is an uninferable jump #
function is_uninferable_jump (instr)

    if ((instr.opcode == JALR and instr.rs1 != 0) or
        (instr.opcode == C.JALR) or
        (instr.opcode == C.JR))

        return TRUE

    return FALSE

# Determine if instruction is an uninferable discontinuity #
function is_uninferable_discon (instr)

    if (is_uninferable_jump(instr) or
        (instr.opcode == URET)      or
        (instr.opcode == SRET)      or
        (instr.opcode == MRET)      or
        (instr.opcode == DRET)      or
        (instr.opcode == ECALL)     or
        (instr.opcode == EBREAK)    or
        (instr.opcode ==
         C.EBREAK))
        return TRUE

    return FALSE

# Determine if instruction is a sequentially inferable jump #
function is_sequential_jump (instr, prev_addr)

    if (not (is_uninferable_jump(instr) and options.sijump))
        return FALSE

    local prev_instr = get_instr(prev_addr)

    if((prev_instr.opcode == AUIPC) or
       (prev_instr.opcode == LUI)  or
       (prev_instr.opcode == C.LUI))
        return (instr.rs1 == prev_instr.rd)

    return FALSE

```

```

# Find the target of a sequentially inferable jump #
function sequential_jump_target (addr, prev_addr)

local instr      = get_instr(addr)
local prev_instr = get_instr(prev_addr)
local target    = 0

if (prev_instr.opcode == AUIPC)
    target = prev_addr
target += prev_instr.imm
if (instr.opcode == JALR)
    target += instr.imm

return target

# Determine if instruction is a call #
# - excludes tail calls as they do not push an address onto the return stack
function is_call (instr)

if ((instr.opcode == JALR and instr.rd == 1) or
    (instr.opcode == C.JALR)           or
    (instr.opcode == JAL and instr.rd == 1) or
    (instr.opcode == C.JAL))
    return TRUE

return FALSE

# Determine if instruction return address can be implicitly inferred #
function is_implicit_return (instr)

if (options.implicit_return == 0) # Implicit return mode disabled
    return FALSE

if ((instr.opcode == JALR and instr.rs1 == 1 and instr.rd == 0) or
    (instr.opcode == C.JR and instr.rs1 == 1))
    if ((te_inst.irreport != get_previous_bit(te_inst, "irreport")) and
        te_inst.irdepth == irstack_depth)
        return FALSE
    return (irstack_depth > 0) return

FALSE

```

```

# Push address onto return stack #
function push_return_stack (address)

if (options.implicit_return == 0) # Implicit return mode disabled
    return

local irstack_depth_max      = discovery_response.return_stack_size ?
                                2**discovery_response.return_stack_size :
                                2**discovery_response.call_counter_size
local instr                   = get_instr(address)
local link                   = address

if (irstack_depth == irstack_depth_max)
    # Delete oldest entry from stack to make room for new entry added below
    irstack_depth--
    for (i = 0; i < irstack_depth; i++)
        return_stack[i] = return_stack[i+1]

    link += instruction_size(instr)

return_stack[irstack_depth] = link
irstack_depth++

return

# Pop address from return stack #
function pop_return_stack ()

irstack_depth-- # function not called if irstack_depth is 0, so no need
                # to check for underflow
local link = return_stack[irstack_depth]

return link

```

第 10 章

10. コードとパケットの例

以下の例では、ret は非推論可能と呼ばれていますが、これは implicit-return モードがオフの場合にのみ当てはまります。

1. 80001a84 から、メイン()で debug_printf() をコールします。

00000000800019e8 <main>:

.....: ...

80001a80: f6d42423	sw a3,-152(s0)
80001a84: ef4ff0ef	jal x1,80001178 <debug_printf>

PC: 80001a84 ->80001178

jal のターゲットは推定可能であるため、NOte_inst パケットが送信されます。

0000000080001178 <debug_printf>:

80001178: 7139	addi sp,sp,-64
8000117a: ...	

2. debug_printf():からの戻り値

80001186: ...
80001188: 6121 addi sp,sp,64
8000118a: 8082 ret

PC: 8000118a ->80001a88

ret のターゲットは推論不能であるため、te_inst te_inst パケット IS が送信されます:
te_inst[format=2 (ADDR_ONLY): アドレス =0x80001a88、アップディスク=0]

80001a88: 00000597	auipc a1,0x0
80001a8c: 65058593	addi a1,a1,1616 # 800020d8 <main+0x6f0>

3. Func_2() を終了し、最終的に取られたブランチの後に ret が続きます。

0000000800010b6 <Func_2>:

.....:	
800010da: 4781	li a5,0
800010dc: 00a05863	blez a0,800010ec <Func_2+0x36>

PC: 800010dc ->800010ec, branch_map にブランチを追加しますが、まだパケットは送信されません。
 branches = 0;branch_map = 0;
 branch_map |= 0 «branches++ ;

800010ec: 60e2	ld	ra,24(sp)
800010ee: 6442	ld	s0,16(sp)
800010f0: 64a2	ld	s1,8(sp)
800010f2: 853e	mv	a0,a5
800010f4: 6105	addi	sp,sp,32
800010f6: 8082	ret	

PC: 800010f6 ->80001b8a

ret のターゲットが不確定であるため、branch_map に ONE ブランチを入れた te_inst パケットが送信されます。
te_inst[format = 1 (DIFF_DELTA): branches =1, branch_map =0x0, address =0x80001b8a ($\Delta=0xab0$) updiscon =0]

0000000800019e8 <main>:

.....:		
80001b8a: f4442603	lw	a2,-188(s0)
80001b8e: ..		

4.3 つのブランチ、次に関数が Proc_1() に戻ります。

000000080001100 <Proc_6>:

.....:		
80001112: c080	sw	s0,0(s1)
80001114: 4785	li	a5,1
80001116: 02f40463	beq	s0,a5,8000113e <Proc_6+0x3e>

PC: 80001116 ->8000111a、branch_map に取られていないブランチを追加しましたが、パケットはまだ送信されていません。
 branches = 0;branch_map = 0;branch_map |= 1 «branches++ ;

8000111a: c81d *beqz s0,80001150 <Proc_6 +0x50>*

PC: 8000111a ->8000111c、ブランチを追加 branch_map に取られていないが、まだパケットは送信されていない。

branch_map |= 1 «branches++ ;

8000111c: 4709 *li a4,2*
8000111e: 04e40063 *beq s0,a4,8000115e <Proc_6+0x5e>*

PC: 8000111e ->8000115e、branch_map にブランチを追加取るが、まだパケットは送信されていない。
branch_map |= 0 «branches++ ;

8000115e: 60e2 *ld ra,24(sp)*
80001160: 6442 *ld s0,16(sp)*
80001162: c09c *sw a5,0(s1)*
80001164: 64a2 *ld s1,8(sp)*
80001166: 6105 *addi sp,sp,32*
80001168: 8082 *ret*
00000000800011d6 <Proc_1>:
.....:
80001258: 00093783 *ld a5,0(s2)*
8000125c:

PC: 80001168 ->80001258

ret のターゲットは推論不可能であるため、*te_inst* パケットが送信され、3 つの分岐が branch_map
te_inst[形式 = 1 (DIFF_DELTA): branches =3、branch_map =0x3、address=0x80001258
(Δ =0x148),updiscon=0]

5.2 つの枝、2 つのジャル、および ret を持つ複雑な例

00000000800011d6 <Proc_1>:

.....:
8000121c: 441c *lw a5,8(s0)*
8000121e: c795 *beqz a5,8000124a <Proc_1+0x74>*

PC: 8000121e ->8000124a、branch_map に分岐を追加しますが、まだパケットは送信されません。
branches = 0;branch_map = 0;
branch_map |= 0 «branches++ ;

8000124a: 44c8 *lw a0,12(s1)*
8000124c: 4799 *li a5,6*
8000124e: 00c40593 *addi a1,s0,12*
80001252: c81c *sw a5,16(s0)*
80001254: eadff0ef *jal x1,80001100 <Proc_6>*

PC: 80001254 ->80001100

*jal*のターゲットは推定可能であるため、パケット *te_inst*送信する必要はありません。

```
0000000080001100 <Proc_6>:
80001100: 1101      addi sp,sp,-32
80001102: e822      sd s0,16(sp)
80001104: e426      sd s1,8(sp)
80001106: ec06      sd ra,24(sp)
80001108: 842a      mv s0,a0
8000110a: 84ae      mv s1,a1
8000110c: fedff0ef  jal x1,800010f8 <Func_3>
```

PC: 8000110c ->800010f8

*jal*のターゲットは推定可能であるため、*te_inst*パケットを送信する必要はありません。

```
00000000800010f8 <Func_3>:
800010f8: 1579      addi a0,a0,-2
800010fa: 00153513  seqz a0,a0
800010fe: 8082      ret
```

PC: 800010fe ->80001110

*ret*のターゲットは推論不能であるため、*te_inst* パケットはまもなく送信されます。

000000080001100 <Proc_6>:

```
.....:
80001110: c115      beqz a0,80001134 <Proc_6+0x34>
80001112: ....
```

PC: 80001110 ->80001112, branch_map に取られていないブランチを追加します。

branch_map |= 1 «branches++ ;

te_inst[format = 1 (DIFF_DELTA): branches =2, branch_map =0x2, address=0x80001110
(δ=0xffffffffffff4), updiscon=1]

11. 奥付

この文章は RISC-V のプロセッサトレース仕様を @shibatchii が RISC-V アーキテクチャ勉強のため訳しているものです。

原文は <https://github.com/riscv/riscv-trace-spec/blob/master/riscv-trace-spec.pdf> です。

原文のライセンス表示

ですが、

"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.

や

"The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10", Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, May 2017.

のように

Creative Commons Attribution 4.0 International License

表示がありませんが、まずは「RISC-V トレース仕様 バージョン 1.0」日本語訳 @shibatchii
ということで進めます。まずいよーとなつたら速攻削除します。

この文章は <https://github.com/shibatchii/RISC-V>
に置いてあります。

翻訳方法としては、原文の LaTex にちょっと手を入れて Ligature(合字)をしないようにして PDF 生成、それを MS office word で読み込み doc 形式へ、そこで全文自動翻訳をかけ、できてきた翻訳を原文を見つつ体裁整えを行っています。

フォントは今までメイリオを使ってましたが、見やすさを考えて UD フォントにしてあります。BIZ UDP 明朝の方が原文に近い感じがしますが、読みにくく感じたので BIZ UDP ゴシックにしてあります。

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。

Twitter: @shibatchii

Mail:shibatchii@gmail.com

2020/06/07 @shibatchii