RISC-V Zicond 拡張について

塩谷 亮太 (shioya@ci.i.u-tokyo.ac.jp)
東京大学大学院情報理工学系研究科 創造情報学専攻

塩谷 亮太

- 所属：
  - 東京大学 大学院情報理工学系研究科
    創造情報学専攻 准教授
  - https://www.rsg.ci.i.u-tokyo.ac.jp/lab/
- 専門：
  - コンピュータ・アーキテクチャやシステム・ソフトウェア,
    セキュリティなど
  - 特にプロセッサのマイクロアーキテクチャ
- 宣伝：
  - 塩谷研では修士や博士の学生さんを募集しています
  - 創造情報学専攻では来年４月入学が可能な冬入試もあります

今日の話題：RISC-V Zicond 拡張

- 以下の２命令から成る：

1. **czero.eqz** rd, rs1, rs2
   ```
   // rd = (rs2 == 0) ? 0 : rs1;
   ```

2. **czero.nez** rd, rs1, rs2
   ```
   // rd = (rs2 != 0) ? 0 : rs1;
   ```

- 一種の条件付き move
  - 条件ごとにゼロ or レジスタ値を代入

RISC-V Zicond 拡張

- 要は条件ごとにゼロ or レジスタ値を代入する命令

  ･･･以上

- ･･･だけで話が終わってしまうとアレなんですが，
  今日はこの２命令の背景等を説明

  - これが一体なんの役にたつのか？
  - なんでゼロ代入みたいな，謎の形してるのか？

  - （全部推測なんだけど，あってるんだろうか？

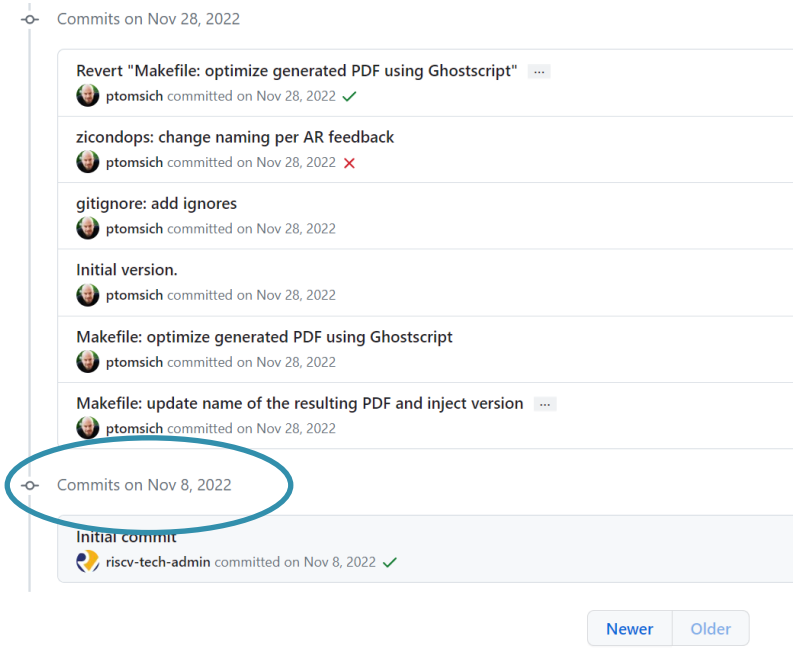
今日の話題

1. Zicond 拡張そのもの
2. 背景となる（マイクロ）アーキテクチャ技術
3. Zicond と従来の条件付き move の違い
4. 効果やサポート状況

Zicond 拡張の仕様

- 仕様：https://github.com/riscv/riscv-zicond

- （たぶん）割と最近策定が開始された拡張
  - コミットログを見ると，最初は2022年11月
  - 現在の最新版は 2023/04/09 リリースの v1.0-rc2



仕様書のイントロ

Chapter 1. Introduction

The Zicond extension defines provides a simple solution that provides most of the benefit and all of the flexibility one would desire to support conditional arithmetic and conditional-select/move operations, while remaining true to the RISC-V design philosophy. The instructions follow the format for R-type instructions with 3 operands (i.e., 2 source operands and 1 destination operand). Using these instructions, branchless sequences can be implemented (typically in two-instruction sequences) without the need for instruction fusion, special provisions during the decoding of architectural instructions, or other microarchitectural provisions.

（機械翻訳）Zicond拡張定義は、RISC-Vの設計思想に忠実でありながら、条件付き算術演算と条件付き選択/移動演算をサポートするために必要なほとんどの利点とすべての柔軟性を実現する、シンプルなソリューションを提供します。この命令は、3つのオペランド（すなわち、2つのソース・オペランドと1つのデスティネーション・オペランド）を持つR型命令の形式に従っている。これらの命令を使用すると、命令フュージョンやアーキテクチャ命令のデコード時の特別な規定、その他のマイクロアーキテクチャ規定を必要とせずに、分岐のないシーケンスを（通常は2命令シーケンスで）実装することができる。

Zicond の命令

- 以下の２命令から成る：

1. **czero.eqz** rd, rs1, rs2

   ```
   // rd = (rs2 == 0) ? 0 : rs1;
   ```

2. **czero.nez** rd, rs1, rs2

   ```
   // rd = (rs2 != 0) ? 0 : rs1;
   ```

ユースケース

- 以下の様に，１つ～２つの他の命令と組み合わせて使う想定
  - https://github.com/riscv/riscv-zicond/blob/main/zicondops.adoc より

Conditional add, if non-zero

```
rd = (rc != 0) ? (rs1 + rs2) : rs1
```

```
czero.eqz  rd, rs2, rc
add        rd, rs1, rd
```

```
// 以下のようなプログラムからのコード生成で使う
if (...) {
    i += 4;
}
```

Conditional select, if zero

```
rd = (rc == 0) ? rs1 : rs2
```

```
czero.nez rd,   rs1, rc
czero.eqz rtmp, rs2, rc
or rd, rd, rtmp
```

なぜ？

- これが一体なんの役にたつのか？
- なんでゼロを代入みたいな，謎の形なのか？

もくじ

1. 背景となる技術：
    1. 分岐予測
    2. 条件付き move
    3. レジスタ・リネーム
2. Zicond と従来の条件付き move の違い
3. 効果やサポート状況

工場のラインを考える

- ベルトコンベアのラインの上を製品が流れていく
  - ４人の人が，それぞれの工程の作業をおこなって完成
- 上のように1つしか製品をながさないと，
  - 各人は他の人が作業している間はヒマ

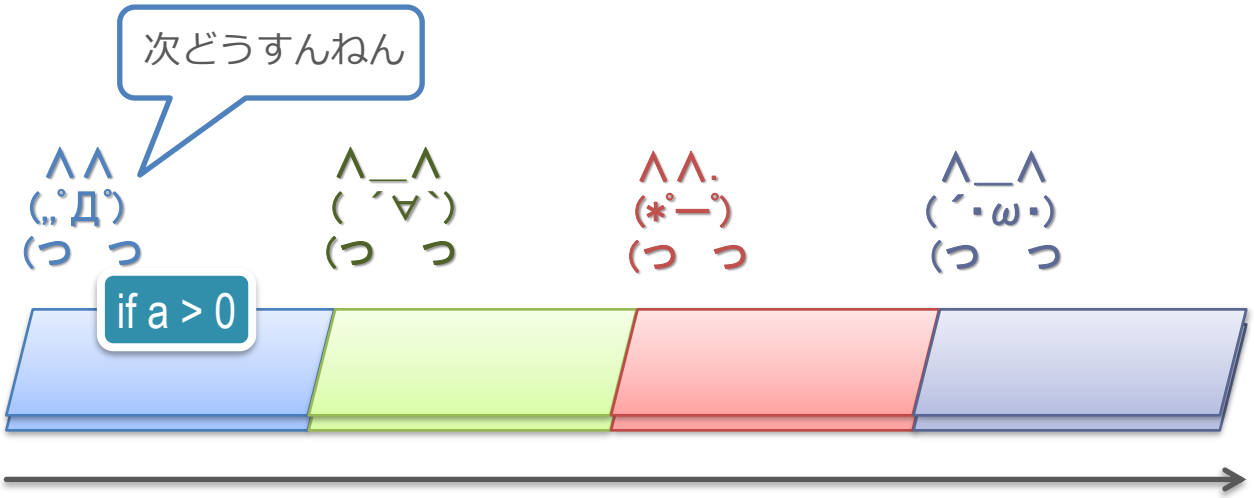
工場のラインを考える

- 実際の工場：複数の製品を同時に流す
  - 各工程を並列して処理することによりスループットを向上
- これが 命令パイプライン

分岐命令の処理と制御ハザード

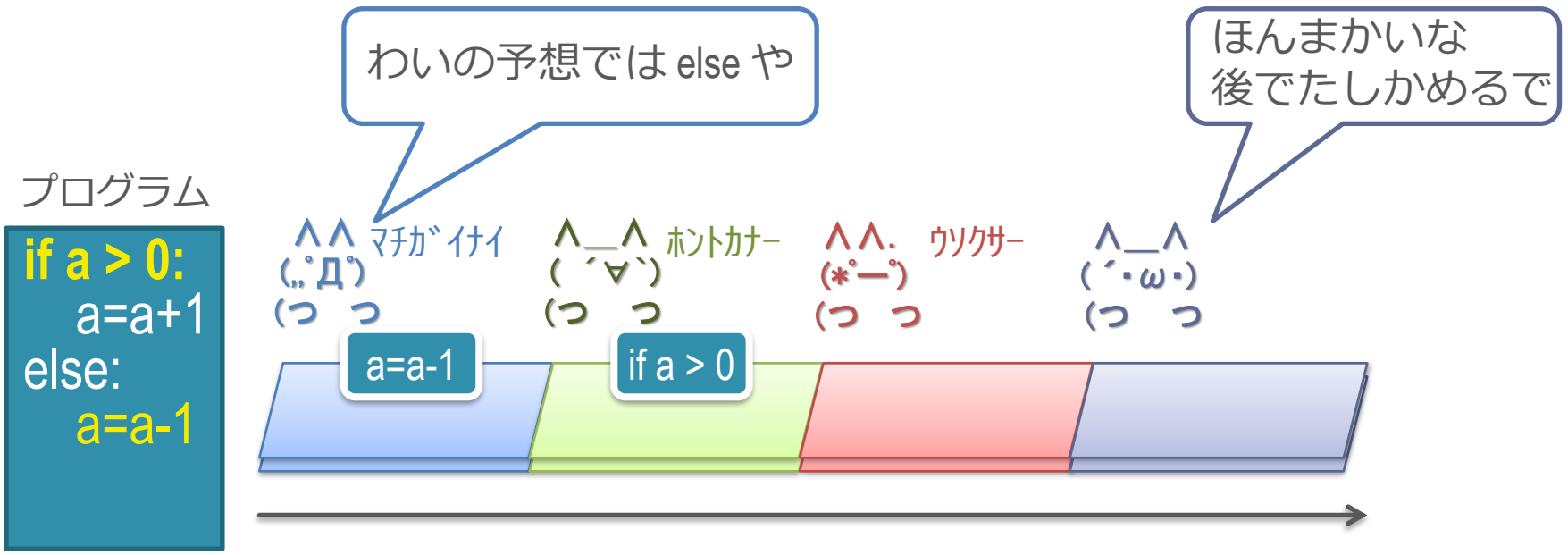
命令メモリ

```
if a > 0:
    a=a+1
else:
    a=a-1
```



- 「if a > 0」の結果は最終段の (´･ω･) の人まで反映出来ない
  - 先頭は次に a=a+1 と a=a-1 のどちらを取り込めばいいのかわからない

分岐予測

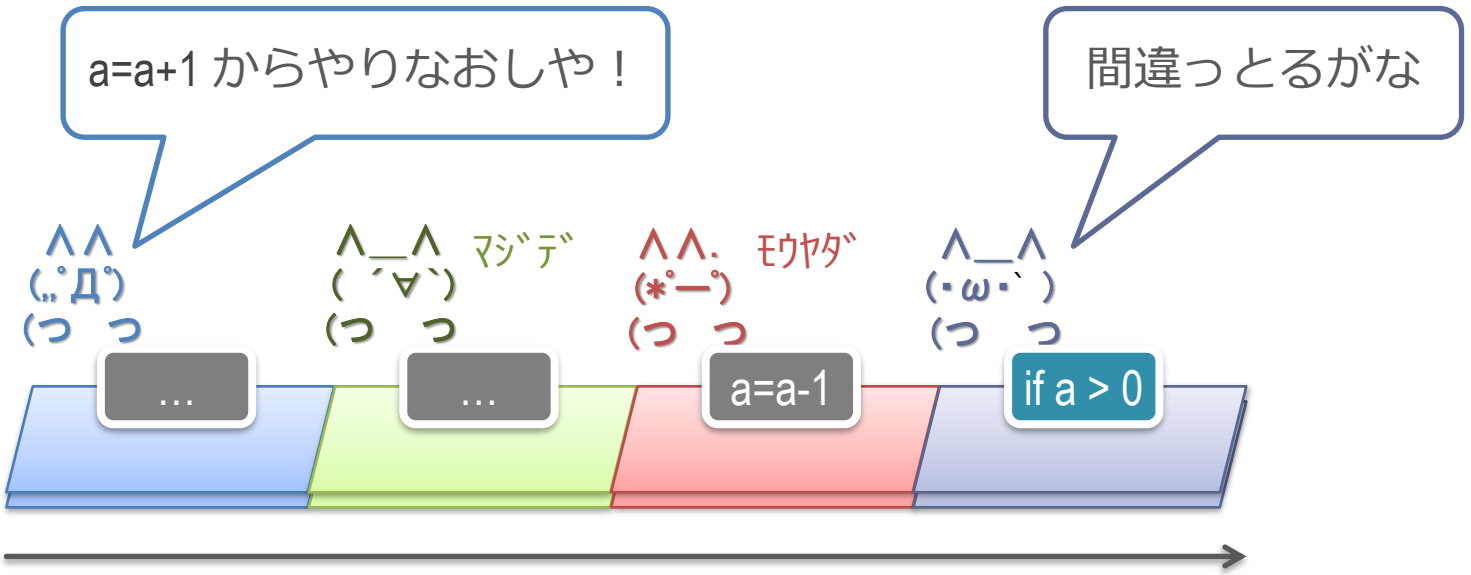


- 動作
  - 「if a > 0」の結果を予測して，命令を取り込む
    - 前回はこっちに行ったので，次もこっちに違いないとかで予測
  - あとから予測が正しいか確認する

分岐予測ペナルティ

プログラム

```
if a > 0:
    a=a+1
else:
    a=a-1
```



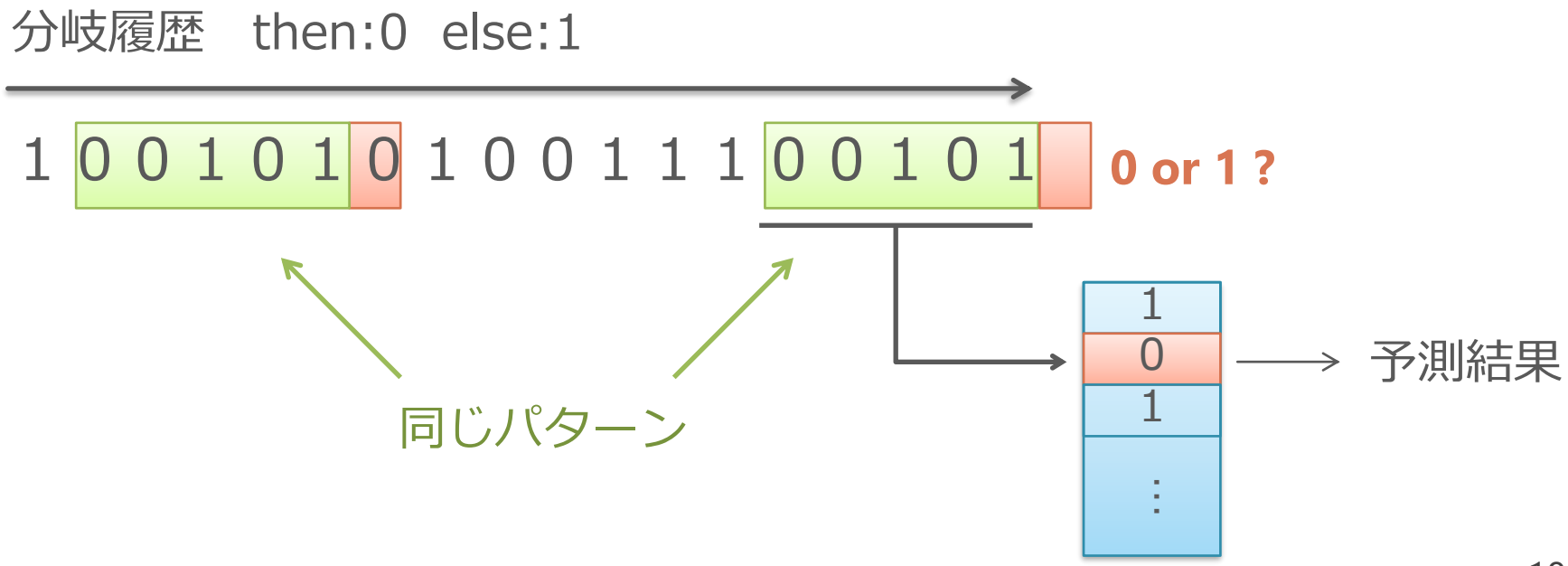
- 予測が間違っていた場合，以降の処理を取り消してやり直す
- この図では，無駄になるのは3命令分

大規模な高性能プロセッサの場合

- 取り消しは最悪１００命令以上に
  - 最近の CPU だと 8命令同時 × 10数段 とか

分岐予測の予測方法：過去のパターンを利用

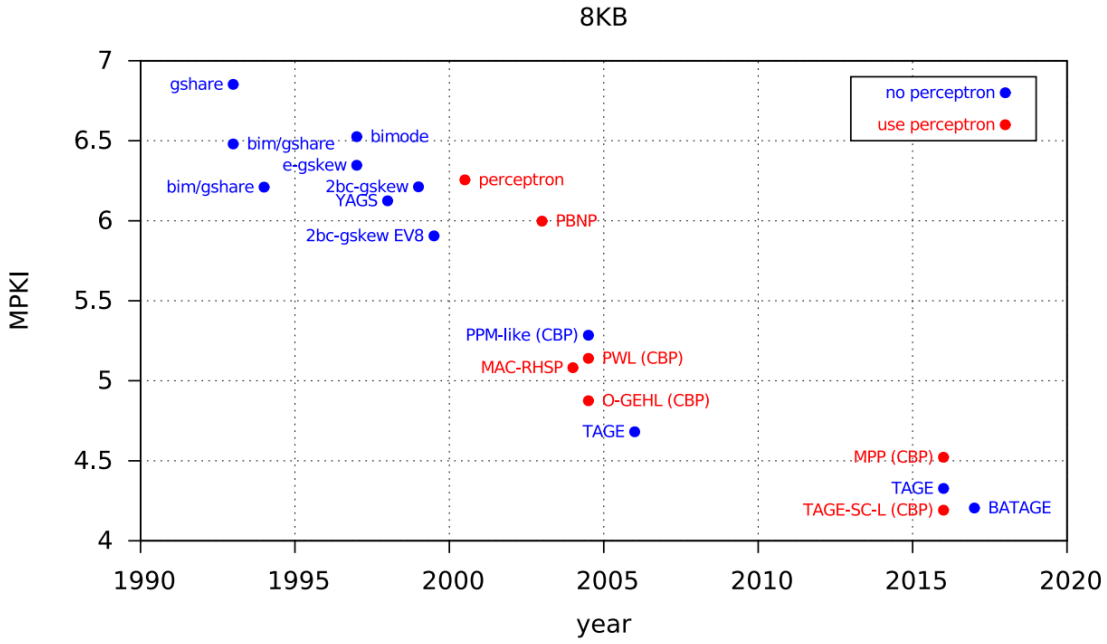
- アルゴリズムの例：
  - パターンをインデクスとしてテーブルにアクセス
  - 直前のパターンでテーブルをひく



現代の分岐予測器の性能

PIERRE MICHAUD, An Alternative TAGE-like Conditional Branch Predictor, TACO 2018 より

- Championship Branch Prediction (CBP) 2016 の環境を使って評価
  - https://www.jilp.org/cbp2016/
- 基本的には TAGE と呼ばれる予測器の一派が一番強い
  - MPKI=misses per kilo instructions
  - 1000 命令実行すると4回ぐらい予測ミスが起きるぐらい



分岐予測の予測方法と問題

- 分岐予測の基本的な予測方法：
  - 過去に起きたパターンが繰り返す性質を利用
  - 大概の分岐は，そもそも毎回同じ方向に行っている
- 過去の分岐結果を繰り返さないものは予測できない
  - `// ダメな例：ランダム値との比較の繰り返し`
    ```
    for (...) {
        if (random() > N) {...}
    }
    ```
  - then か else の２択なので，最低 50% は当たる

予測出来ない分岐の例：CoreMark 内の CRC 計算部分

- CoreMark：
  コンパクトな合成ベンチマーク
  - リンクドリストの処理，行列乗算，状態機械などを含む
  - https://github.com/eembc/coremark/tree/main
- 要所で右の CRC 計算を行う
  - 本処理の計算結果の値を使って分岐

→過去のパターンをくり返さないので予測できない

```
ee_u16
crcu8(ee_u8 data, ee_u16 crc)
{
    ee_u8 i = 0, x16 = 0, carry = 0;

    for (i = 0; i < 8; i++)
    {
        x16 = (ee_u8)((data & 1) ^ ((ee_u8)crc & 1));
        data >>= 1;

        if (x16 == 1)
        {
            crc ^= 0x4002;
            carry = 1;
        }
        else
            carry = 0;
        crc >>= 1;
        if (carry)
            crc |= 0x8000;
        else
            crc &= 0x7fff;
    }
    return crc;
}
```

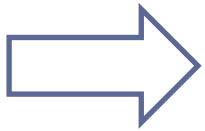
もくじ

条件付き move による分岐の回避

- 基本的な考え方：
  - if と else の双方の処理をやって，結果を選択する
  - この選択部分を「条件付き `move` 命令」として実装
    - 選択を分岐命令で実現してしまうと意味が無い

- 「`out = cond ? a : b`」
  - `cond` によって `a` と `b` のどっちかを選んで出力

```
if (cond) {
    out = in + 1;
}
else {
    out = in - 1;
}
```



```
a = in + 1;
b = in - 1;
out = cond ? a : b;
```

条件付き move

- 上から順にパイプラインに取り込んで実行するだけで良い
  - 予測して投機的に実行して取り消すとかがない
- ただし，then と else の双方の処理をやる必要がある

条件付き move による性能向上

- then/else パートがそれぞれ 4 命令から成る場合・・・
  - 通常の分岐で実装：
    - 50% の確率：予測ヒット時は 4 命令の実行で済むが，
      50% の確率：予測ミス時は追加で 100 命令無駄に実行
      → 期待値：4×0.5+104×0.5=54 命令
  - 条件付き move を使用：
    - 毎回 4×2=8 命令を実行
- 注意：
  - この計算はとても雑です
  - そもそも本当は命令数じゃなくて時間で考える必要がありますが，ここでは簡単のため命令数で考えています

分岐処理の中身が十分に小さくないと意味がない

- then/else パートがそれぞれ 200 命令から成る場合・・・
  - 通常の分岐で実装：
    - 50% の確率：予測ヒット時は 200 命令の実行で済むが，
    - 50% の確率：予測ミス時は追加で 100 命令無駄に実行
      → 期待値：200×0.5+300×0.5=250 命令
  - 条件付き move を使用：
    - 毎回 200×2=400 命令を実行

分岐予測が十分当たる場合も意味が無い
（プログラム中の大概の分岐は予測できる）

- MPKI=4 の場合・・・
  - 1000命令実行して予測ミスが４回
  - 分岐命令が５命令に１回出るとすると，ミス率２%程度

- then/else パートがそれぞれ 4 命令の場合・・・

  - 通常の分岐で実装：
    - 98 % の確率： 予測ヒット時は 4 命令の実行で済むが，
      2% の確率： 予測ミスし時は 100 命令無駄に実行
      → 期待値 4×0.98+104×0.02= 6 命令

  - 条件付き move を使用：
    - 毎回 8 命令を実行

CoreMark 内の CRC 計算部分は，うってつけ

- ここは条件付き move にすごく向いている
  - 過去のパターンを繰り返さないので予測できない
  - 分岐の中身が小さいので両パス実行しても影響が小さい

```
ee_u16
crcu8(ee_u8 data, ee_u16 crc)
{
    ee_u8 i = 0, x16 = 0, carry = 0;

    for (i = 0; i < 8; i++)
    {
        x16 = (ee_u8)((data & 1) ^ ((ee_u8)crc & 1));
        data >>= 1;

        if (x16 == 1)
        {
            crc ^= 0x4002;
            carry = 1;
        }
        else
            carry = 0;
        crc >>= 1;
        if (carry)
            crc |= 0x8000;
        else
            crc &= 0x7fff;
    }
    return crc;
}
```

既存の命令セットにおける条件付き move（１）

- ARM 64bit：Conditional select 命令
  - `CSEL Xd, Xn, Xm, cond`

    `// ４オペランドで３項演算子と同様の動作`
    `Xd = cond ? Xn : Xm`

既存の命令セットにおける条件付き move（２）

- Alpha：Conditional move 命令
  - CMOVxx Ra, Rb, Rc

    // ３オペランドで
    // xx の条件がなり立ったら代入（以下は非ゼロの場合）
    if (Ra)
        Rc = Rb

  - 組み合わせて使ったりする

```
// R4 = R1 ? R2 : R3
CMOVNE R1,R2→R4 // if (R1)  R4=R2
CMOVEQ R1,R3→R4 // if (!R1) R4=R3
```

既存の命令セットにおける条件付き move（３）

- x86：Conditional move 命令
  - CMOVcc r, r/m　　// cc の部分は条件コード

    ```
    // フラグが指定した条件を満たしたときだけ代入
    // Alpha と同じ
    if (cond)
       dst = src;
    ```

- 余談：APX 拡張で最近さらにいろいろ追加されたらしい･･･
  - https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-performance-extensions-apx.html

RISC-V では？

- 基本命令セットからは意図的に外されている
- Andrew Waterman, Design of the RISC-V Instruction Set Architecture より

  > We consciously omitted support for conditional moves and predication. Both enable some form of if-conversion, a transformation by which some control hazards can be traded for data hazards. ...

  （機械翻訳）条件付き move と述語のサポートは意図的に省いた。どちらもある種のif変換を可能にするもので、それによって制御ハザードをデータハザードと交換することができる。...

一応 B (Bitmanip) 拡張には提案があった

- ４オペランド命令として提案
  - `cmov rd, rs2, rs1, rs3`

    ```
    // ARM のものと同様
    //  ３項演算子のように働く
    rd = rs2 ? rs1 : rs3
    ```

- ドラフトにはあったが，廃案になったらしい？
  - https://muxup.com/2023q1/whats-new-for-risc-v-in-llvm-16
    … the previously proposed but now abandoned Zbt extension (part of the earlier bitmanip spec)
    （機械翻訳）以前提案されていたが、現在は放棄されているZbt拡張（以前のbitmanip仕様の一部）

なぜ条件付き move を入れたくないか？（１）

- ４オペランドタイプ（ARM 64, RISC-V B 拡張）の場合
  - `cmov rd, rs2, rs1, rs3   // rd = rs2 ? rs1 : rs3`
- イヤな点
  1. 定義できる命令数の減少
     - `32bit` 固定長の空間内で `5bit×4=20bit` が食われてしまう
  2. ハードの複雑化
     - レジスタ・ファイルなどのポートが追加で必要
     - この手の回路はポート数の２乗のオーダーで消費電力や面積が増加
- 余談：浮動小数点 の `Fused multiply add (FMA)` は４オペランドだが これは効果が非常に大きいため例外的に認められてる

なぜ条件付き move を入れたくないか？（２）

- ３オペランドタイプ（x86, Alpha）の場合
  - `cmov rd, rs2, rs1   // if (rs2) rd = rs1;`

- 問題なさそう？
  - 定義できる命令数が減少する？
    - ３オペランドなので，フィールドの消費は増えていない
  - ハードは複雑化する？
    - ３オペランドなので，これも大丈夫？

RISC-V から条件付き move が省かれた理由

- Andrew Waterman, Design of the RISC-V Instruction Set Architecture より

- > We consciously omitted support for conditional moves and predication.

  ...

  Both techniques complicate implementations with register renaming, since the old value of the destination register must be copied to the new physical register when the predicate is false.

  （機械翻訳）条件付き move と述語のサポートは意図的に省いた。

  ...

  どちらのテクニックも、述語が偽の場合、コピー先レジスタの古い値を新しい物理レジスタにコピーしなければならないため、レジスタリネーミングを伴う実装を複雑にする。

もくじ

レジスタ・リネーム

- 偽の依存を取り除くために行われる
- Out-of-order 実行を効率よく行うため
  - 偽の依存があると，スケジューリングがかなり制約される
- （･･･すいません，以下しばらくかなりややこしい話が続きます

真の依存と偽の依存（逆依存と出力依存）

1. 真の依存（RAW: Read after Write）：

```
I1: add x1←x2+1   // 普通に「依存」と言われて思うもの
I2: add x3←x1+1
```

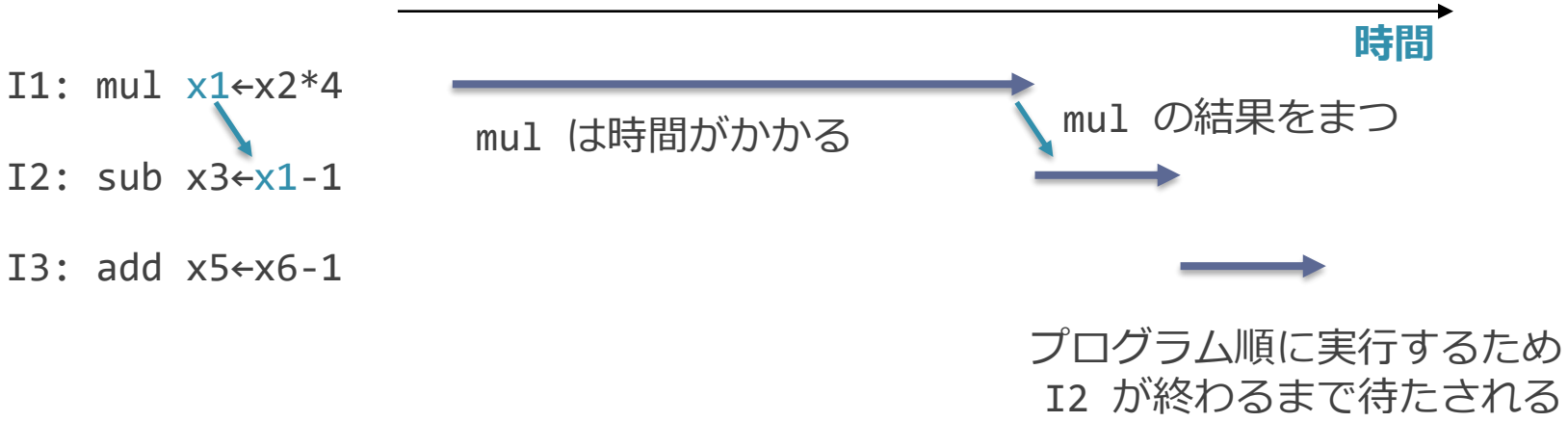
2. 逆依存（WAR: Write after Read）：

```
I1: add x2←x1+1
I2: add x1←x3+1
```

3. 出力依存（WAW: Write after Write）：

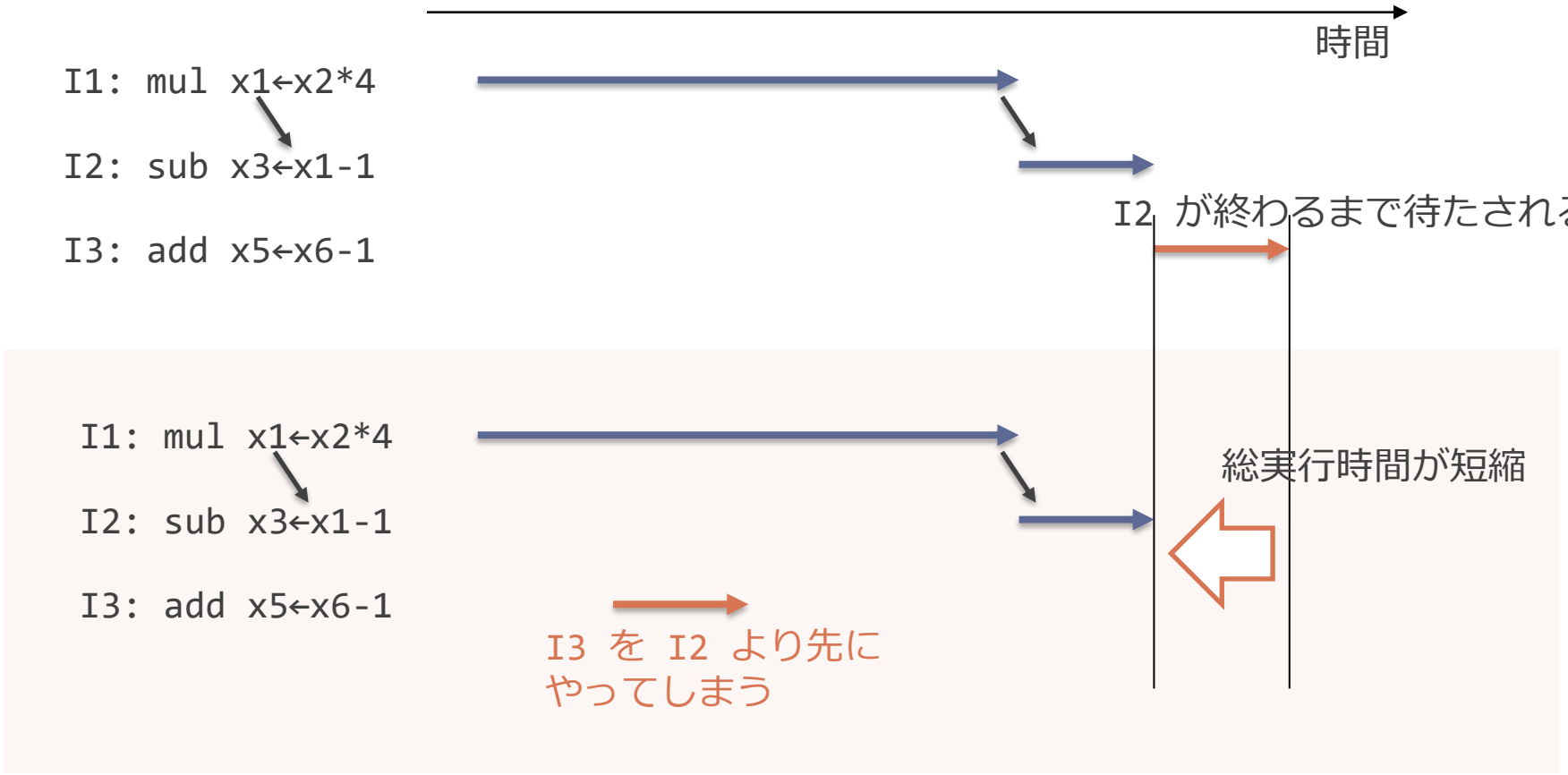
```
I1: add x1←x2+1
I2: add x1←x3+1
```

Out-of-order 実行のモチベーション



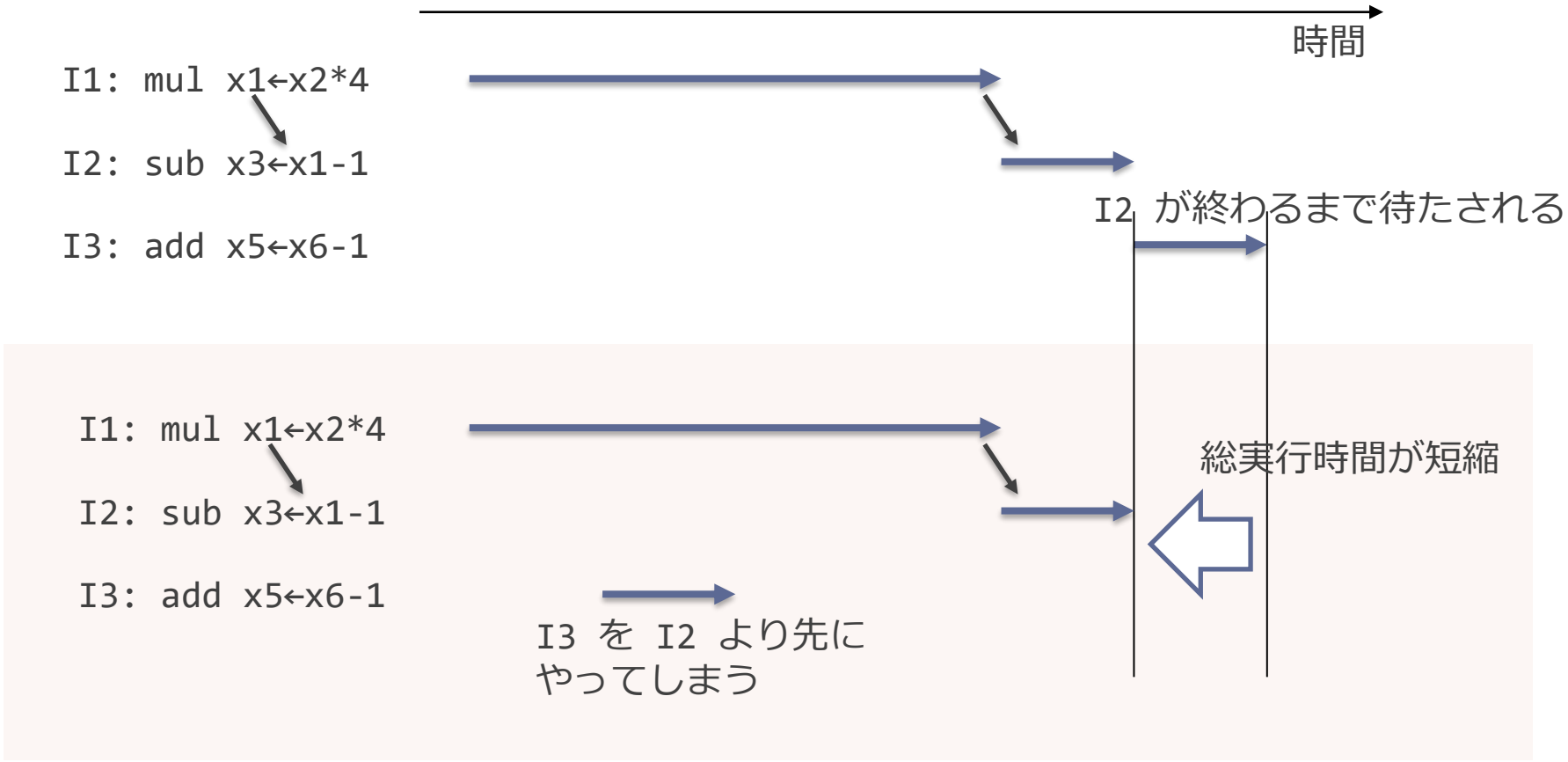
- 各命令の実行の様子を矢印で表示
  - mul は乗算なので時間がかかる=矢印が長い
- I1->I2->I3 のプログラムに書かれた順序通りに実行した場合・・・
  - I2 は I1 に依存しているため，I1 が終わるのをまつ
  - それに合わせて I3 の実行も遅れる
    - I3 は I1/I2 に関係ないのだから先にできるのでは？

Out-of-order 実行により高速化



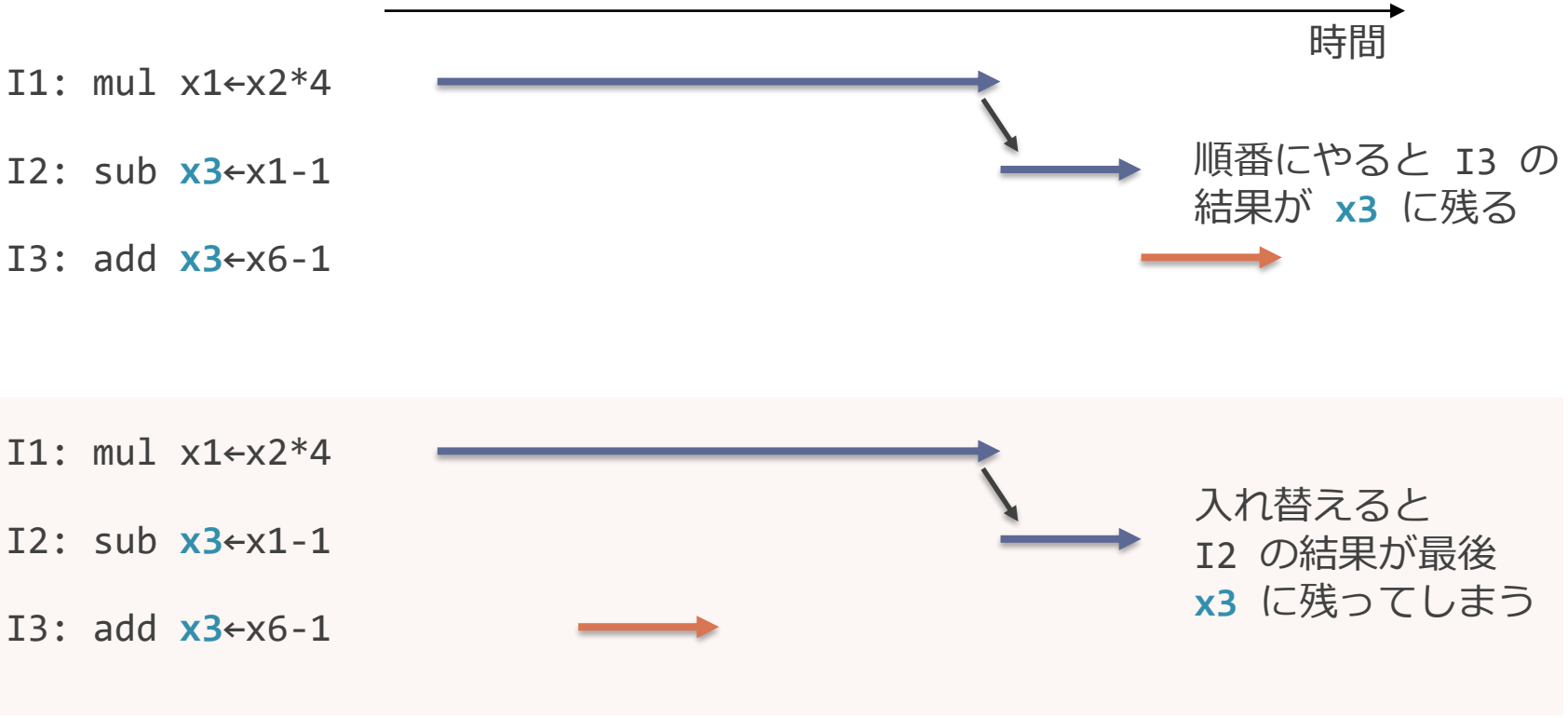
- I3 は I1/I2 に対して依存がないので，順序を入れ替えて実行開始する

実行結果は変わらない



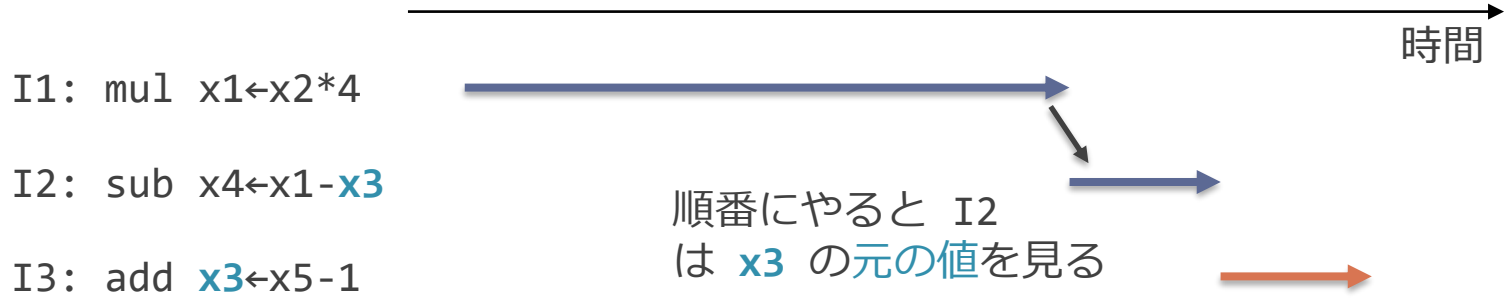
- この場合は I3 を I2 より先に実行しても，
  最後まで実行した場合の実行結果は変わらない
  - なぜなら I3 は I1/I2 と同じレジスタを触っていないから

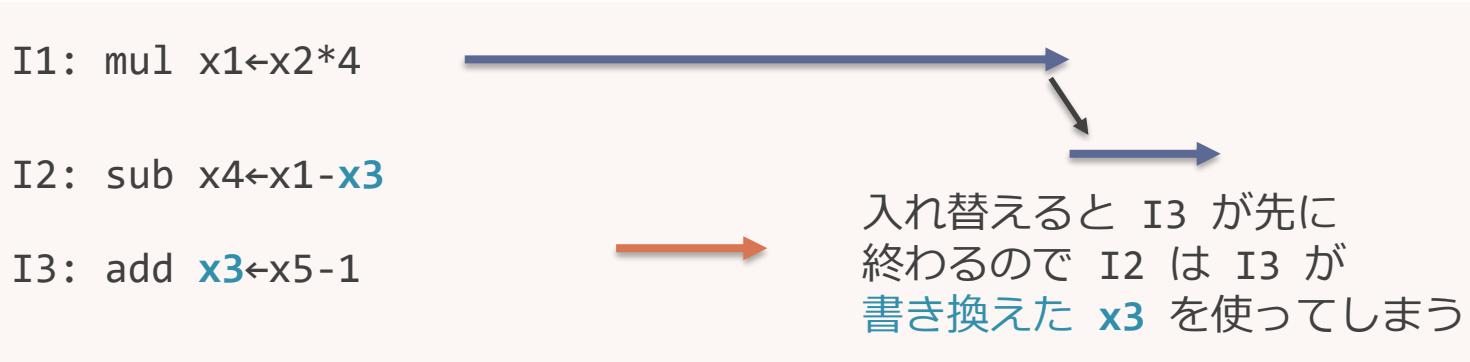
出力依存（WAW）があると破綻する



- 出力依存（WAW: Write after Write）
  - 同じレジスタに書き込むことで発生する依存
- 真の依存がないので I3 の演算自体は I1/I2 と無関係に開始できる
  - しかし I2 と I3 は同じ **x3** に書き込むので書き込み順を変えるとまずい

逆依存（WAR）があっても破綻する





- 逆依存（WAR: Write after Read）
  - I2 と I3 では x3 を読んだ後に書いている
- 真の依存がないので I3 の演算自体は I1/I2 と無関係に開始できる
  - しかし実行順を入れ替えると，I3 の結果を I2 が読んでしまう

レジスタ・リネーム

- 状況：
  - 偽の依存があると，実行の順序を入れ替えると破綻する
  - 真の依存が無いのであれば，原理的には演算自体はできる
- 目的：偽の依存を動的に取り除く
  - 真の依存だけ考えて順序を入れ替えられるようになる
- 方針：レジスタの名前を付け替える
  - 偽の依存の原因 = 同じレジスタ番号の使い回し
  - 命令間で同じレジスタを使うから偽の依存が生じる

レジスタ・リネーム

- ディスティネーションに専用のレジスタを動的に毎回新しく割り当てる
- レジスタ番号がかぶらないので，他の命令との間で出力依存や逆依存は生じなくなる
  - 出力依存：
    - 毎回新しい番号に書き込むため，自明に被らない
  - 逆依存：
    - 毎回新しい番号に書き込むため，自分より前に読んでるレジスタ番号と被ることがなくなる

```
I1: mul x3←x2*4
I2: add x3←x1+1
I3: sub x1←x5-1
I4: and x6←x7&1
```

```
I1: mul p20←p12*4
I2: add p21←x11+1
I3: sub p22←p15-1
I4: and p23←p17&1
```

レジスタ・リネーム

- 論理レジスタ：
  - 命令セットで定義されているレジスタ
  - プログラマから見える
- 物理レジスタ：
  - レジスタ・リネームによって割り当てられる内部のレジスタ
  - 変換表を使って論理レジスタ番号から変換して得る

```
I1: mul x3←x2*4
I2: add x3←x1+1
I3: sub x1←x5-1
I4: and x6←x7&1
```

```
I1: mul p20←p12*4
I2: add p21←x11+1
I3: sub p22←p15-1
I4: and p23←p17&1
```

レジスタ・リネームのまとめ

- 条件付き move に関係する重要なこと
  - 動的な命令ごとに新しい物理レジスタが割り当てられる
  - （この結論だけ分かっていればある意味 OK です

RISC-V から条件付き move が省かれた理由（再）

- Both techniques complicate implementations with register renaming, since the old value of the destination register must be copied to the new physical register when the predicate is false.

  どちらのテクニックも、述語が偽の場合、コピー先レジスタの古い値を新しい物理レジスタにコピーしなければならないため、レジスタリネーミングを伴う実装を複雑にする。

３オペランド条件付き move とリネーム

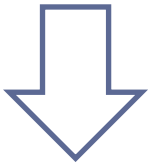
- `cmov rd, rs2, rs1   // if (rs2) rd = rs1;`
  - 条件が成り立ったときだけ `rd` に書き込む
  - 一見３つのオペランドだけがあるように見える

リネームの結果，３オペランドが４オペランドになってしまう

- リネームでは命令ごとに新しい物理レジスタが必ず割り当てられて書き込まれる
  - これにより偽の依存でレジスタ番号が被ることが原理的になくなる
  - しかし「条件付きで書かない」と言う動作が実現できなくなる

- 条件付き `move` が４オペランド命令になってしまう
  - `rd` の古い値を読んで，`rd` に新しく割り当てられるところにコピーする動作になる

```
// if (rs2==0) rd = rs1;
// 条件に応じて rd を書き換えない
cmov rd, rs2, rs1
```



```
cmov p20(rd) ← p1(rs2) == 0 ? p17(rs1) : p15(rd)
```

レジスタリネームと条件付き move のまとめ

- Both techniques complicate implementations with register renaming, since the old value of the destination register must be copied to the new physical register when the predicate is false.

  …

  どちらのテクニックも、述語が偽の場合、コピー先レジスタの古い値を新しい物理レジスタにコピーしなければならないため、レジスタリネーミングを伴う実装を複雑にする。

- ３オペランド条件付き move が実質４オペランドになってしまう
  - レジスタ・ファイル等の回路が複雑化する

Design of the RISC-V Instruction Set Architecture より，Alpha への批判

- Indeed, DEC’s first implementation with out-of-order execution employed some chicanery to avoid the extra datapath for this instruction. The Alpha 21264 executed the conditional move instruction by splitting it into two micro-operations, the first of which evaluated the move condition and the second of which performed the move [57]. This approach also required that the physical register file be widened by one bit to hold the intermediate result.

  （機械翻訳）実際、DECのアウトオブオーダー実行による最初の実装では、この命令のための余分なデータパスを回避するために、いくつかの奇策が用いられた。Alpha21264は、条件付き move 命令を2つのマイクロ演算に分割して実行し、1つ目のマイクロ演算で移動条件を評価し、2つ目のマイクロ演算で移動を実行した[57]。このアプローチでは、中間結果を保持するために物理レジスタ・ファイルを1 ビット拡張する必要もあった。

もくじ

Zicond の命令

- 以下の２命令から成る：

1. **czero.eqz** rd, rs1, rs2

   ```
   // rd = (rs2 == 0) ? 0 : rs1;
   ```

2. **czero.nez** rd, rs1, rs2

   ```
   // rd = (rs2 != 0) ? 0 : rs1;
   // rd = (rs2 == 0) ? rs1 : 0; と等価
   ```

Zicond の命令のリネーム

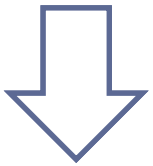
```
// Zicond のリネーム
// rd = (rs2 == 0) ? rs1 : 0;
czero.nez rd, rs1, rs2
```

```
czero.nez p20(rd) ← p1(rs2) == 0 ? p17(rs1) : 0
```

リネームしても３オペランドのまま！

- 即値のゼロを書き込むので，rd の古い値を読まなくてよくなってる

```
// Zicond のリネーム
// rd = (rs2 == 0) ? rs1 : 0;
czero.nez rd, rs1, rs2
```



```
czero.nez p20(rd) ← p1(rs2) == 0 ? p17(rs1) : 0
```

```
// 旧来の３オペランド条件付き move をリネームした場合
// if (rs2 == 0) rd = rs1;
cmov rd, rs2, rs1
```

このオペランドが不要に

```
cmov p20(rd)      ← p1(rs2) == 0 ? p17(rs1) : p15(rd)
```

塩谷の感想

- 思いつかんかったが，なるほどこれは賢いと思った
  - コロンブスの卵
  - 仕組み自体は極めて単純
    - でも Alpha や RISC-V の当初の設計者も思いつかなかった
- 一部 の RISC-V CPU では，ハードによる分岐命令の条件付き move への変換も実装していた
  - BOOM v3，SiFive の一部 CPU など
  - 要は難しいハードを考えてなんとかしてた
    - しかし ISA レベルで全く単純に解決された

仕様書のイントロ（再）

Chapter 1. Introduction

The Zicond extension defines provides a simple solution that provides most of the benefit and all of the flexibility one would desire to support conditional arithmetic and conditional-select/move operations, while remaining true to the RISC-V design philosophy. The instructions follow the format for R-type instructions with 3 operands (i.e., 2 source operands and 1 destination operand). Using these instructions, branchless sequences can be implemented (typically in two-instruction sequences) without the need for instruction fusion, special provisions during the decoding of architectural instructions, or other microarchitectural provisions.

（機械翻訳）Zicond拡張定義は、RISC-Vの設計思想に忠実でありながら、条件付き算術演算と条件付き選択/移動演算をサポートするために必要なほとんどの利点とすべての柔軟性を提供するシンプルなソリューションを提供します。この命令は、3つのオペランド（すなわち、2つのソース・オペランドと1つのデスティネーション・オペランド）を持つR型命令の形式に従っている。これらの命令を使用すると、命令フュージョンやアーキテクチャ命令のデコード時の特別な規定、その他のマイクロアーキテクチャ規定を必要とせずに、分岐のないシーケンスを（通常は2命令シーケンスで）実装することができる。

もくじ

状況

- まだ一応ドラフトだが，近日中に正式に承認されると思われる
  - 既に Freeze にはなっているが，Ratified にはまだなってない
  - 筋が良いので Fast track で審議されている？
    - 以下は https://github.com/riscv/riscv-zicond の仕様書より

1.1. Suitability for Fast Track Extension Process

This proposed extension meets the Fast Track criteria: it consists of two, simple R-form instructions, it addresses a wide range of use-cases for branchless sequences, it composes with the existing RISC-V instruction set, and is not expected to be contentious.

（機械翻訳）この拡張案は、2つの単純なRフォーム命令で構成され、分岐なしシーケンスの幅広いユースケースに対応し、既存のRISC-V命令セットと互換性があり、論争になる可能性がないと予想される点で、Fast Trackの基準を満たしている。

プロファイルへの導入

- アプリケーション・プロセッサの最新のプロファイル（RVA23U64）にも必須の拡張として Zicond は入っている
  - プロファイル：標準となる拡張のセット
  - https://github.com/riscv/riscv-profiles/blob/main/rva23-profile.adoc

The following mandatory extensions are new in RVA23U64:

- **V** Vector Extension.

Note	V was optional in RVA22U64.

- **Zvfhmin** Vector FP16 conversion instructions.
- **Zihintntl** Non-temporal locality hints.
- **Zicond** Conditional Zeroing instructions.
- **Zcb** Additional 16b compressed instructions.
- **Zfa** Additional scalar FP instructions.
- **Zawrs** Wait on reservation set.
- **Zjpm** Pointer masking (ignore high bits of addresses)

ツールのサポート（2023/08/09 時点）

- サポートを確認したツール
  - GNU binutils（Ver. 2.41 以降）
  - QEMU（Ver 8.0.0 以降）
  - Spike
- 開発中？
  - GCC：パッチはできて投げられているぽい？
    - https://patchwork.ozlabs.org/project/gcc/list/?series=341354
  - LLVM：実装は終わっている？
    - https://reviews.llvm.org/rGa755e80ed1d2fe439d7c8c0fa38c399e398aa4f0

RISC-V LLVM current status: vs draft RVA23U64 profile

	Assembler	Codegen
zicond (experimental)	✔	✔(*)
zcb	✔	✔
zfa (experimental)	✔	✔
zbc	✔	✔
zvfh (experimental)	✔	✔(*)
zfbfmin (experimental)	✔	WIP
zvfbfmin, zvfbfwma (experimental)	✔	WIP
zvkng, zvksg, zvbb, zvbc (experimental)	✔	✘

CoreMark における Zicond の導入と効果

- 手書きによりアセンブリコードを変換
  - コンパイラによるサポートはまだ使えないため
  - CRC 演算部分の分岐への適用
- 分岐予測ミス回数を測定
  - Spike で分岐トレースを生成
  - CBP シミュレータによる
    分岐予測ミス回数を測定
    - https://jilp.org/cbp2016/
- データ測定協力：松井くん

```
ee_u16
crcu8(ee_u8 data, ee_u16 crc)
{
    ee_u8 i = 0, x16 = 0, carry = 0;

    for (i = 0; i < 8; i++)
    {
        x16 = (ee_u8)((data & 1) ^ ((ee_u8)crc & 1));
        data >>= 1;

        if (x16 == 1)
        {
            crc ^= 0x4002;
            carry = 1;
        }
        else
            carry = 0;
        crc >>= 1;
        if (carry)
            crc |= 0x8000;
        else
            crc &= 0x7fff;
    }
    return crc;
}
```

Zicond により CoreMark では予測ミスが大幅に削減

- MPKI（1000命令あたりの分岐予測ミス回数）
  - Zicond なし
    - gshare_8KB: 16.4329
    - gshare_64KB: 12.4124
    - TAGE-SC-L_8KB: 5.0102
    - TAGE-SC-L_64KB: 3.7521
  - Zicond あり
    - gshare_8KB: 8.2413
    - gshare_64KB: 6.8847
    - TAGE-SC-L_8KB: 0.2897
    - TAGE-SC-L_64KB: 0.0768

- TAGE-SC-L だと予測ミスがほぼ全部なくなる
  - 高性能プロセッサの場合，性能は10%～20%程度向上しつつ消費電力は相当に減りそう
  - gshare は伝統的な予測器，TAGE-SC-L は現在最強の予測器の１つ

まとめ

- Zicond 拡張
  - 条件ごとにゼロ or レジスタ値を代入する命令
  - 最近 RISC-V に導入された
- 今日の話題
  - これが一体なんの役にたつのか？
  - なんでゼロ代入みたいな，謎の形してるのか？
- 非常に単純ながら大きな効果がある
  - モダンな分岐予測器と組み合わせれば CoreMark から分岐予測ミスを一掃できる
  - RISC-V の弱点の１つを解決