先進計算機構成論 10

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
shioya@ci.i.u-tokyo.ac.jp

前回の感想や質問とか

- 時々インテルのことを皮肉っぽく言うのが面白いです。
- Intel の命令を見ていると，_mm256_add_epi32 などで Throughput が 1/3 と 1 サイクルを下回る命令がいくつかあるのですが，これはどういう原理で動いているのでしょうか…？

前回の感想や質問とか

- 発行キューが早く減るように実行する順番を選んでいく方がいいと思うのですが（物理レジスタを節約できるので）、どのようにその順番を決めているのでしょうか。

- レジスタリネーミングで、フリーリストへの返却のタイミングがよく分かりませんでした。レジスタをどこかのタイミングで解放しないと、フリーリストにレジスタが無くなりうまくいかなくなると思うのですが、どのタイミングで解放するのでしょうか。

前回の感想や質問とか

- 最終課題ではISCAとかMICROの論文を読むとのことでしたが、私は深層学習を専攻しており、汎用のプロセッサよりもTPUやMN-COREのような機械学習向けのプロセッサに興味があります。ISCAやMICROではこういった特定の種類に特化したプロセッサの発表とかもありますか？

- 以前他の方の質問であったかもしれませんが，この分野のtop conferenceを確認したいです.

前回の感想や質問とか

- マトリクススケジューラのそれまでとの違いは，物理レジスタベースではなく命令ベースでwakeupしている点ですか.
- マトリクス・スケジューラは近年できた技術ということでしたが、最近のCPUはマトリクス・スケジューラを用いるのが一般的なのでしょうか？(開発後広く普及したのか？)

前回の感想や質問とか

- out-of-orderでの実行がなされる場合に、実際のソフトウェア開発で注意すべきことはあるのでしょうか。
- 現在主力となってるのはOoO発行/OoO完了とのことですが、もうIn-order発行のやつは完全に使われてないのでしょうか？
- 理情だがアウトオブオーダー実行の意味を先生が言ったとおりに捉えていた。

前回の感想や質問とか

- トマスロ方式より物理レジスタ方式の方が単純で設計しやすく性能も出しやすいとのことでしたが，二つの手法の性能比較を定量的に表した文献等ありますでしょうか.
- スパコンの富嶽について、CPUの観点から何か面白い話はあるでしょうか

前回の感想や質問とか

- 特にout-of-order発行ではEXやMEMなどの処理が複数の命令から同時に要求されそうで，並列実行可能な数以上に処理が要求されて詰まるようなことはないのか気になった
- コンパイラでの最適化ではなく、CPU単位でやる理由が知りたいです。(命令セットに依存する最適化を行うからって認識なのですが。)

前回の感想や質問とか

- 質問なのですが、論理レジスタが物理レジスタよりも小さく設定されていましたが、そのモチベーションは「レジスタリネームを用いるために、余剰なレジスタを残しておきたい」ということなのでしょうか。それ以外には論理レジスタと物理レジスタのサイズを変化させる理由が思いつきませんでした。

前回の内容

- 動的スケジューリングの詳細

今回の内容

1. 動的スケジューリングの詳細続き
    1. 例外への対応
    2. ロード・ストアへの対応
2. GPU のアーキテクチャ概要

前回までの動的スケジューリングの説明

- モチベーション：
  in-order 発行/ out-of-order 完了による，下記をなんとかしたい
  1. 出力依存（WAW）があると止まってしまう
  2. 依存がある命令があるとそこで
     パイプラインが完全に止まってしまう

- 動的スケジューリング
  1. レジスタ・リネームにより偽の依存を取り除く
  2. プログラム順ではなく，真に依存が満たされた順に命令を発行

レジスタ・リネーム

- 目的：出力依存と逆依存を取り除く
  - ◇ 真の依存にのみ従って発行を行う
- 方針：レジスタの名前を付け替える
  - ◇ 偽の依存の原因 = 同じレジスタの使い回し
- 各命令のディスティネーションに専用のレジスタを与える
  - ◇ 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

```
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←p11+1
I3: sub p22←p15-1
I4: and p23←p17&1
```

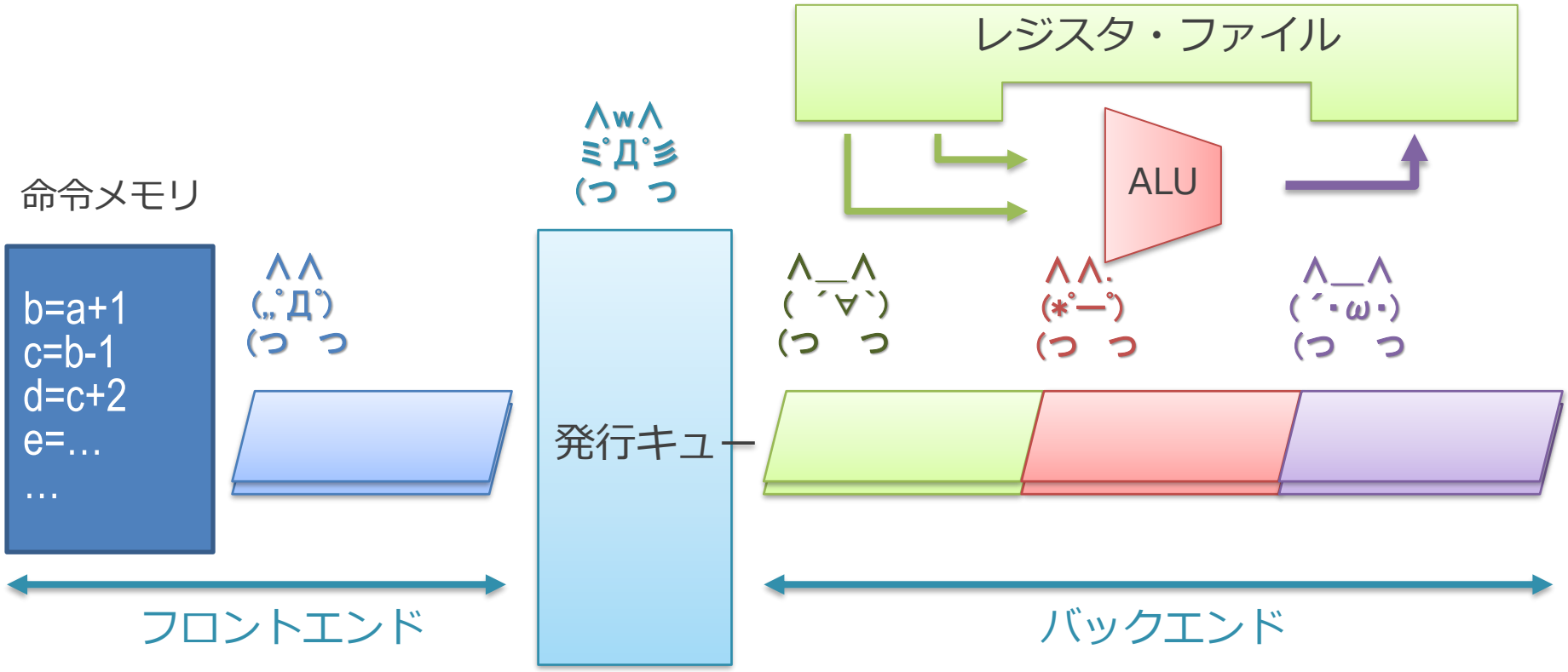
レジスタ・リネーム

- 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
  - ◇ 通常論理レジスタの数倍程度の数を用意する
  - ◇ プログラマからは見えない

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

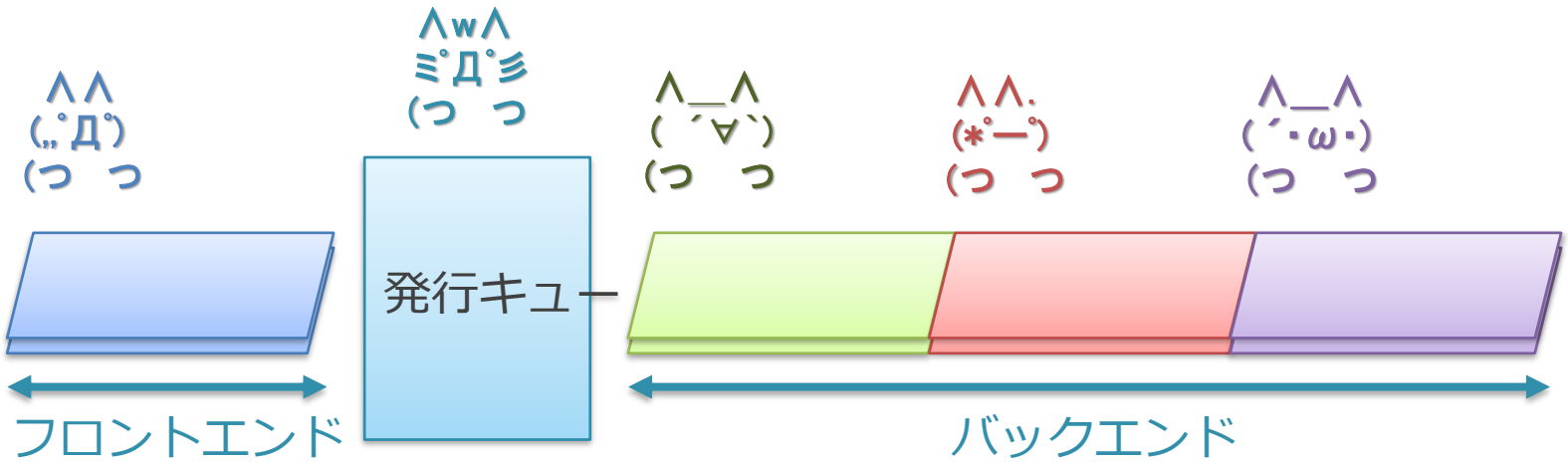
```
I1: mul p20←p12*4
I2: add p21←x11+1
I3: sub p22←p15-1
I4: and p23←p17&1
```

out-of-order 発行を行う CPU の構造



- 発行キューによって前後に分離された構造を持つ
  1. フロントエンド：　命令をフェッチ，リネーム
  2. 発行キュー：　発行待ち命令の待ち合わせのバッファ
  3. バックエンド：　命令を実行

大ざっぱな動作



1. フロントエンドで命令を順にフェッチしてリネーム
2. 発行キューにディスパッチ
3. 発行可能なものから順にバックエンドに命令を発行
4. レジスタを読んで演算器で実行し書き戻す

今回の内容

1. 動的スケジューリングの詳細
    1. **例外への対応**
    2. ロード・ストアへの対応
2. GPU のアーキテクチャ概要

例外

- 制御フロー：
  - ◇ 命令がどのように実行されていくか
    = PC がどのように変化するか の流れ
  - ◇ 通常は +4 されていき，分岐によってたまに飛ぶ
- 例外とは，例外的な制御フローのこと
  - ◇ 例外イベントが起きると，
    あらかじめ設定された場所に PC が飛ぶ

ソフトウェア例外とハードウェア例外

- 「例外（exception）」がさすもの：
  1. ソフトウェアの例外
     - `try {...}`
       `catch {...}`
  2. ハードウェアの例外
     - ゼロ除算
     - メモリ・アクセス違反
- 今回の講義では例外と言ったら，ハード例外をさすことに

例外ハンドラ：

- 例外ハンドラ：
  - 特別なレジスタに，例外発生時の飛び先アドレスを登録しておく
    - 例外が起きると強制的にそのアドレスに分岐してくる
  - そこに例外への対応コード（例外ハンドラ）を用意しておく

例外ハンドラ：

- 例外ハンドラの中身
  1. 回復不能な場合
     - ゼロ除算，NULL ポインタ・アクセスなど
     - プログラムの実行を終了させて，エラー・メッセージを表示
       - 「一般保護違反」「Segmentation fault」
  2. 回復可能な場合
     - スワップアウトされたメモリへのアクセスなど
     - スワップからのデータの読み出しなどを行ってから，元の命令を再実行

例外の例 1
（注：実際の RISC-V ではゼロ除算例外は存在しない）

- ゼロ除算例外の場合：

```
        csrw mtvec, HANDLER    // 例外ハンドラを設定
        ...
        div x1 ← x2 / 0        // ゼロ除算実行
        ...

        // ゼロ除算が起きると，ここに飛ばされる
        // OS が用意した例外ハンドラよって必要な処置が行われる
        //  （この場合はエラーを表示してプログラムを落とす）
HANDLER:
        call PRINT_ERROR_MSG
        call EXIT
```

例外の例 2

- スワップからの回復

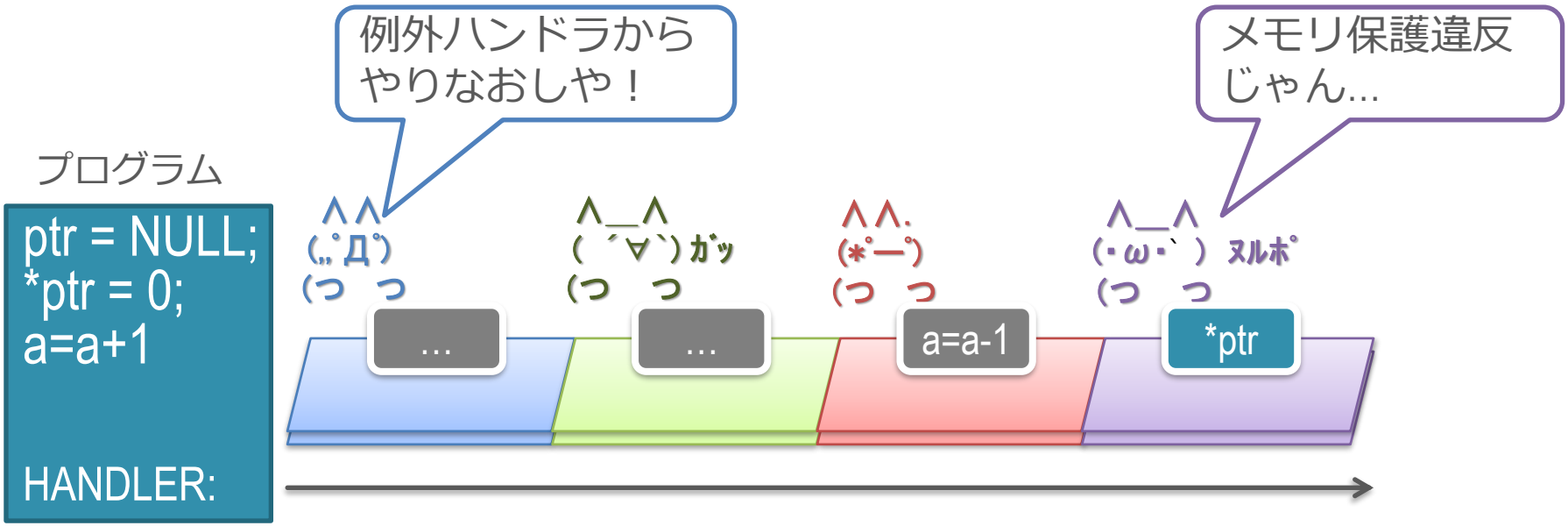
```
        ...
        ld x1 ← [x2]          // x2 のアドレスのデータは
        ...                   // メモリ不足により
                              // 現在スワップされて HDD にある

        // スワップ領域へのアクセスが起きると，ここに飛ばされる
        // OS が用意した例外ハンドラよって必要な処置が行われる
        //  （この場合は HDD から必要なデータを呼んでくる）
HANDLER:
        call RECOVER_SWAP
        mret // ld に戻ってやり直す
```

例外の例３

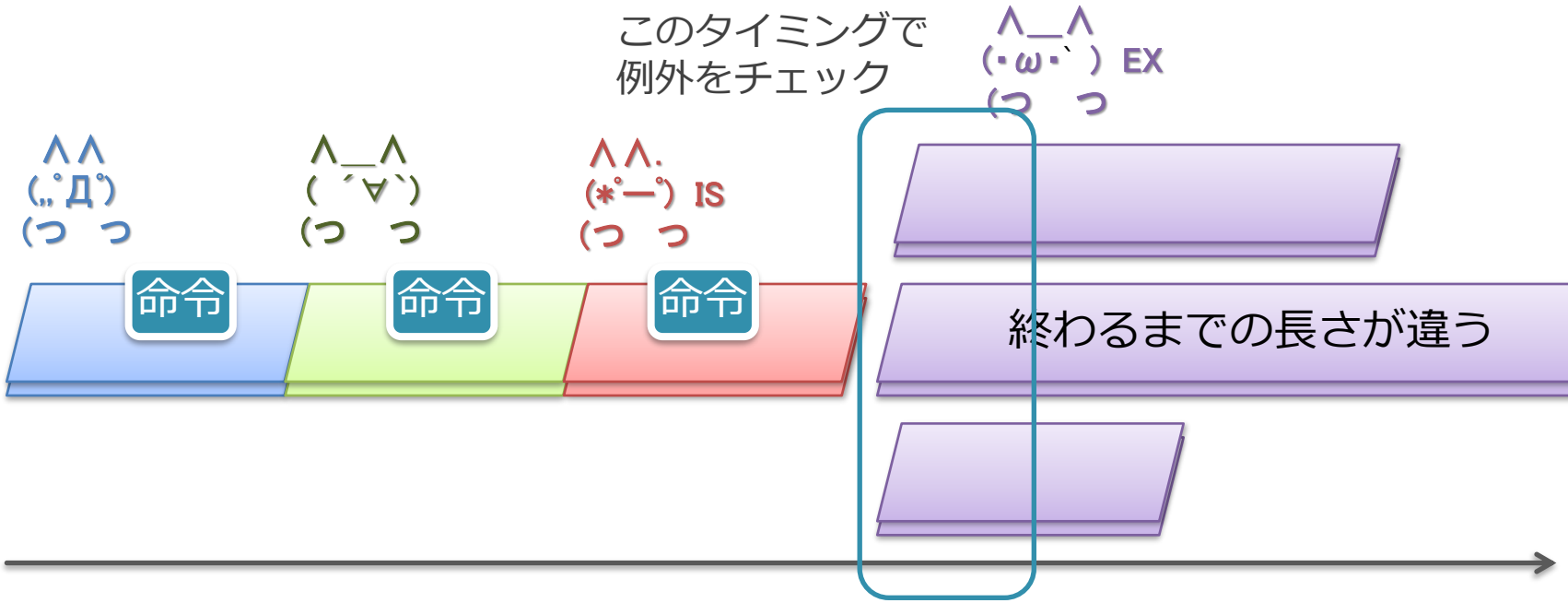
- ブレーク・ポイントの実装
  - ◇ デバッガのブレーク・ポイントも例外を使って実装される
- たとえば，C 言語の加算の文を考える
  1. その文に対応するコンパイル結果の add 命令がどこかにある（gcc なら -g をつけると両者の対応がバイナリに埋め込まれる
     `i = i + 1;  →  add x1, x1, 1`
  2. この add 命令を一時的に例外を発生させる命令に書き換える
     `add x1, x1, 1 → ebreak`
  3. ebreak 命令が実行されると例外ハンドラに飛ぶ
     - □ デバッガにブレーク・ポイントに到達したことを通知

例外への対応：単純なパイプラインの場合

- 本質的には分岐予測ミス時の対処と同じ
  - ◇ 例外を起こした命令以降を取り消し
  - ◇ PC を例外ハンドラに設定して，やりなおす



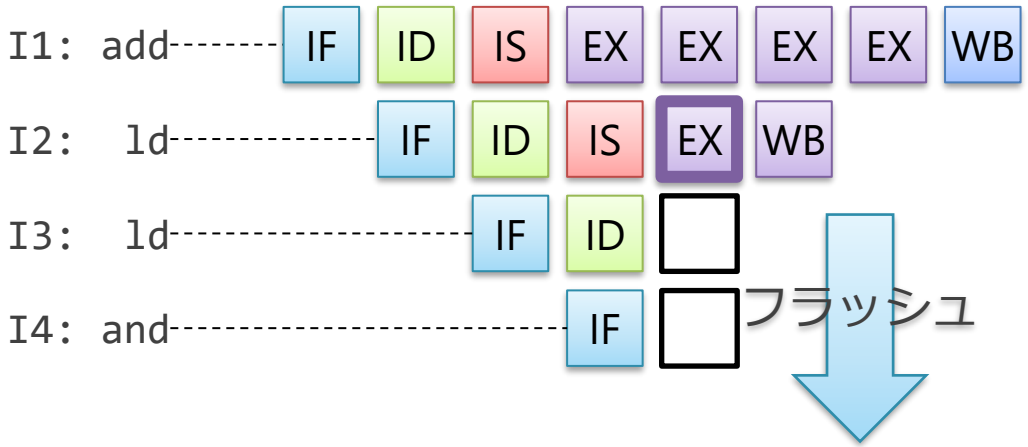
in-order 発行/out-of-order 完了の場合



- (*ﾟーﾟ) IS（発行）までは左側からプログラム順に命令が流れてく
  - ◇ (・ω・`) EX（実行）は開始地点は同じだが，終わるまでの長さが違う
  - ◇ 論理演算は 1 サイクルでおわるが，乗算は時間がかかる… など
- EX の先頭で例外の検出を行えば，左側を全て消すだけで良い
  - ◇ ここを通過した命令は実行が確定される
  - ◇ EX 先頭より後ろでは例外が発生してはならない

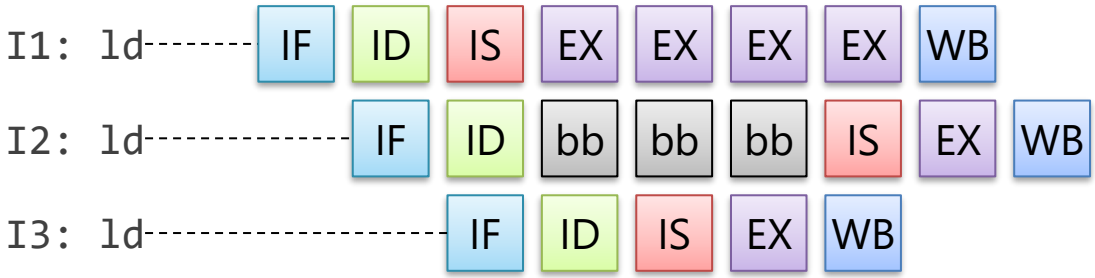
in-order 発行/out-of-order 完了の場合

- in-order 発行/out-of-order 完了の場合
  - ◇ 発行は in-order なので，実行（EX）の開始も in-order
  - ◇ 単純にパイプライン上流を消せば良い
- I2 で例外が発生した場合，I3 と I4 を取り消す
  - ◇ 分岐予測ミス時のフラッシュと同じ

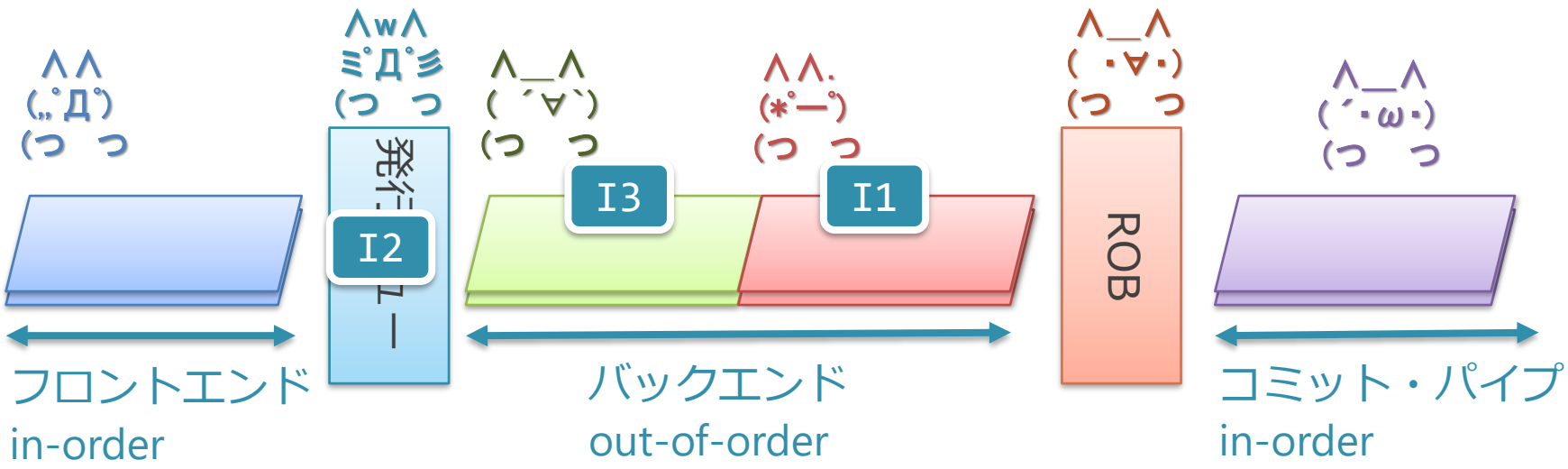


out-of-order 発行/out-of-order 完了の場合

- 命令の実行順序はプログラム順とは無関係
  - ◇ 下の場合，I1 → I3 → I2 の順で実行
- I2 と I3 がそれぞれ例外を発生させた場合，どうなるのか？
  - ◇ I2 で例外ハンドラに飛ぶべき
  - ◇ しかしそれは I3 実行のタイミングではわからない
  - ◇ プログラム順に実行したときと同じ結果になる必要がある



リオーダ・バッファ（ROB: re-order buffer）

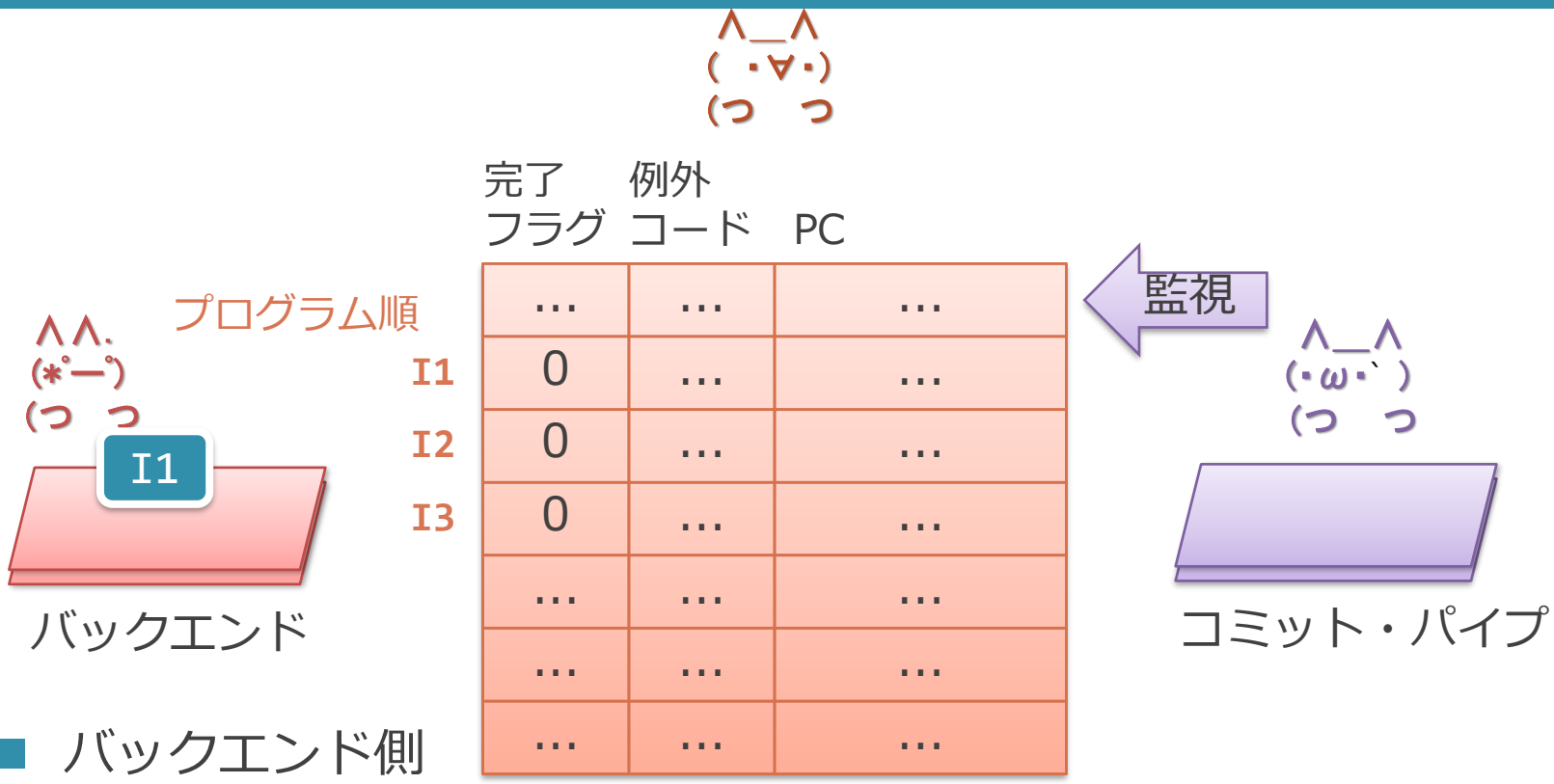


- バックエンドの後ろにリオーダ・バッファ（ROB）を追加
  - ◇ バックエンドで完了した命令は ROB に完了した印をつけていく
  - ◇ コミット・パイプで in-order に印を読み出し，例外を反映
  - ◇ （上記のバックエンド/コミット・パイプをそれぞれ実行コア/バックエンド と呼ぶ文献もある）
- コミット（commit）：実行を確定させる操作

ROB の中身

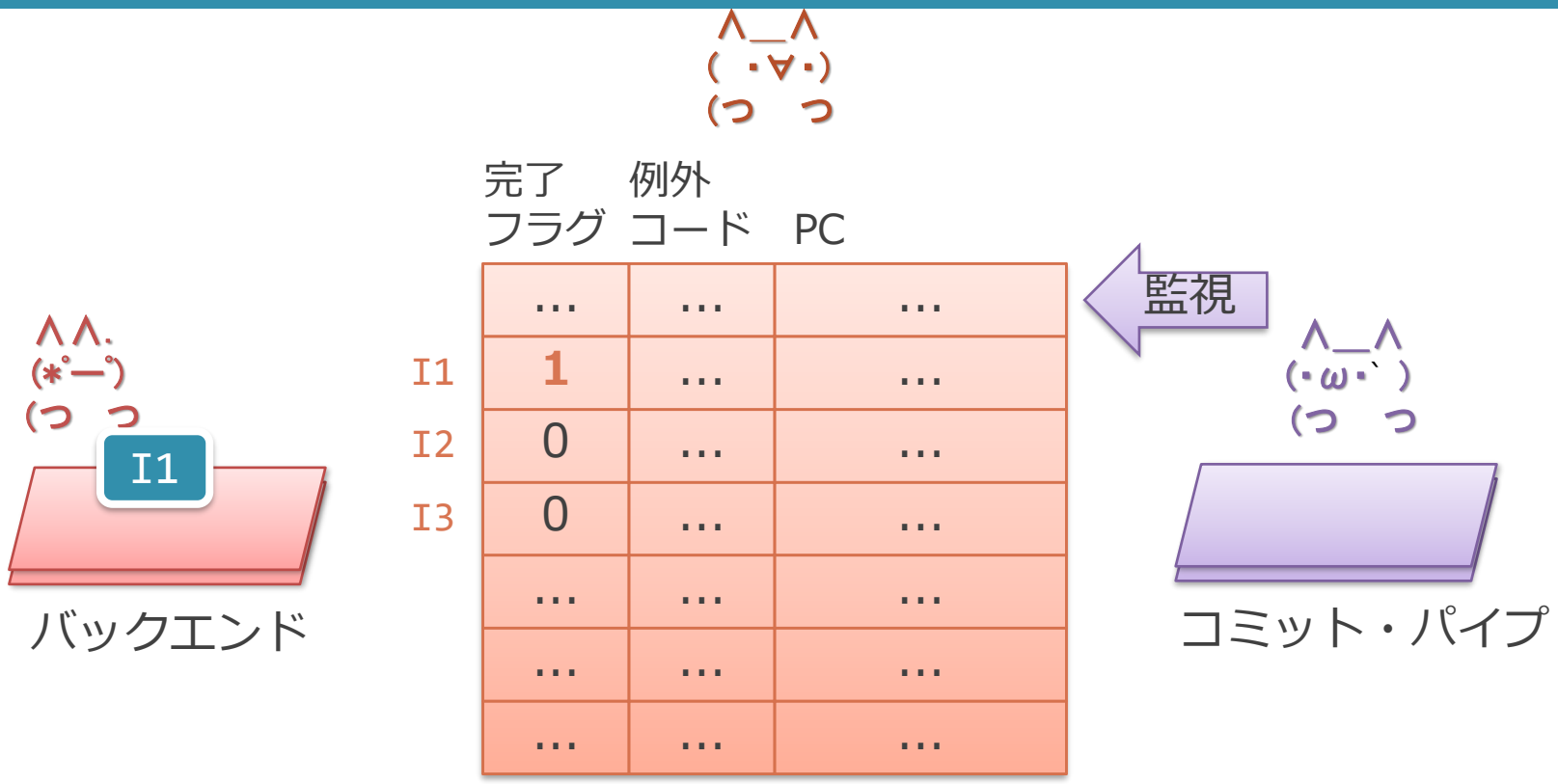
- ROB：
  - ◇ プログラム順にエントリが確保される FIFO
- 内部にあるフィールド
  1. 完了フラグ
     - □ 完了した命令は 1 を書き込む
  2. 例外コード
     - □ 例外を発生させた場合は，その種類を書く
  3. その命令のPC or 分岐先ターゲット
     - □ どっちを入れるかは例外の種類次第

ROB の動作（１）



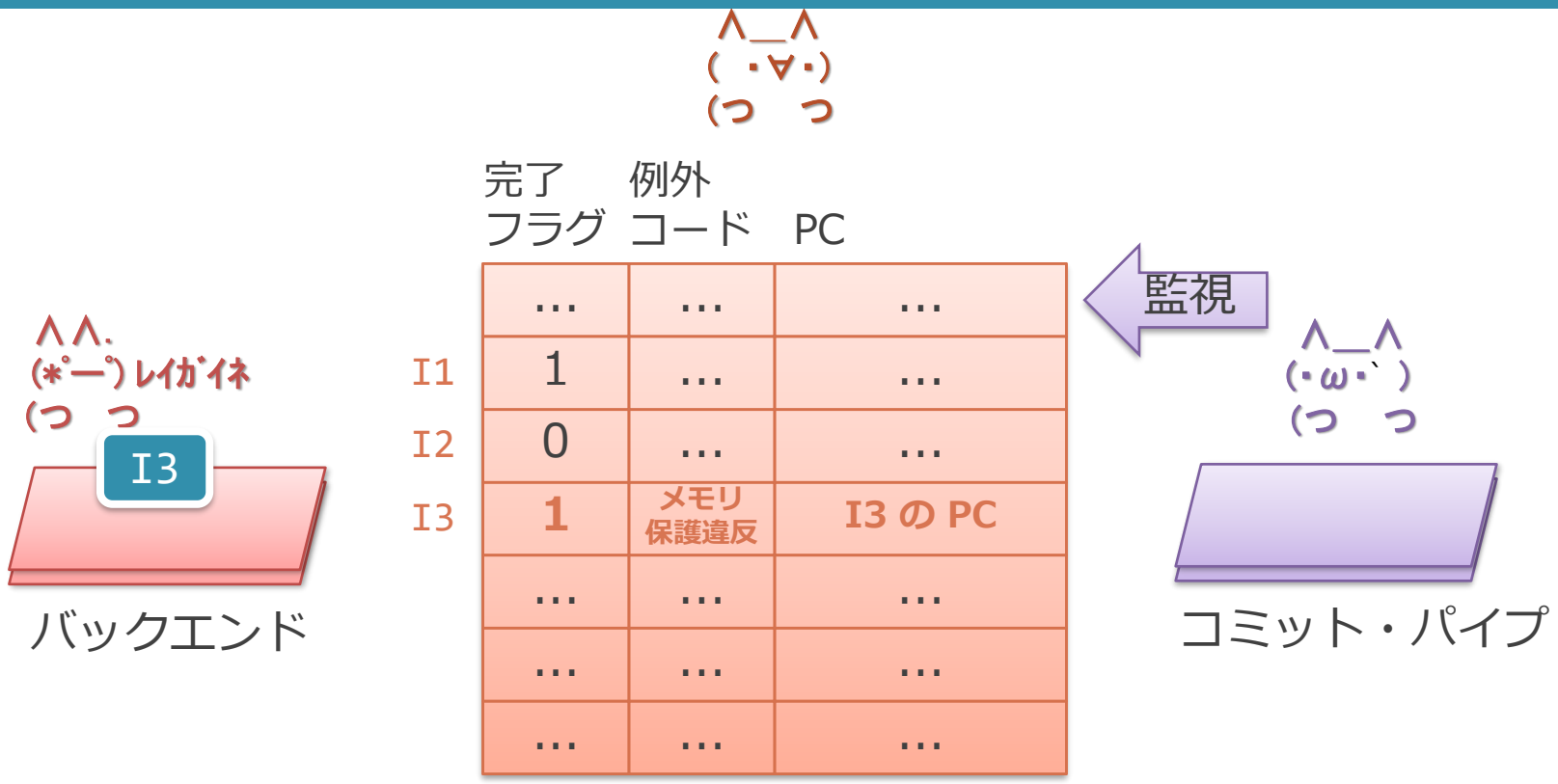
- バックエンド側
  - ◇ 各命令はプログラム順に ROB のエントリを確保
  - ◇ バックエンドで完了した命令は out-of-order に ROB に書き込む
- コミット・パイプ側
  - ◇ in-order に「ここまで完了した」部分を監視

ROB の動作（２）



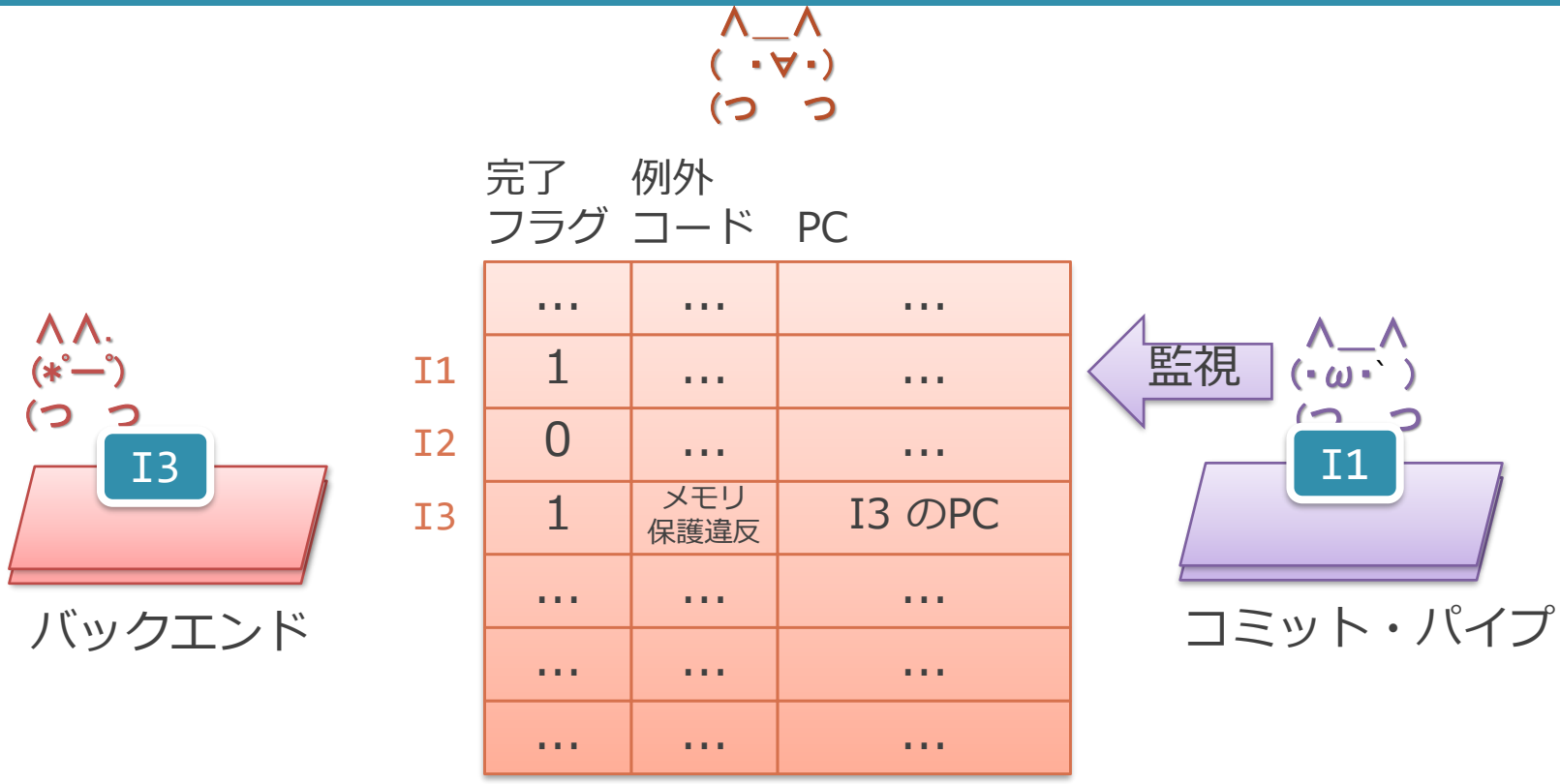
- I1 が完了したので，ROB に書き込みを行う
  - ◇ 正常完了したので，完了フラグに１を書く

ROB の動作（３）



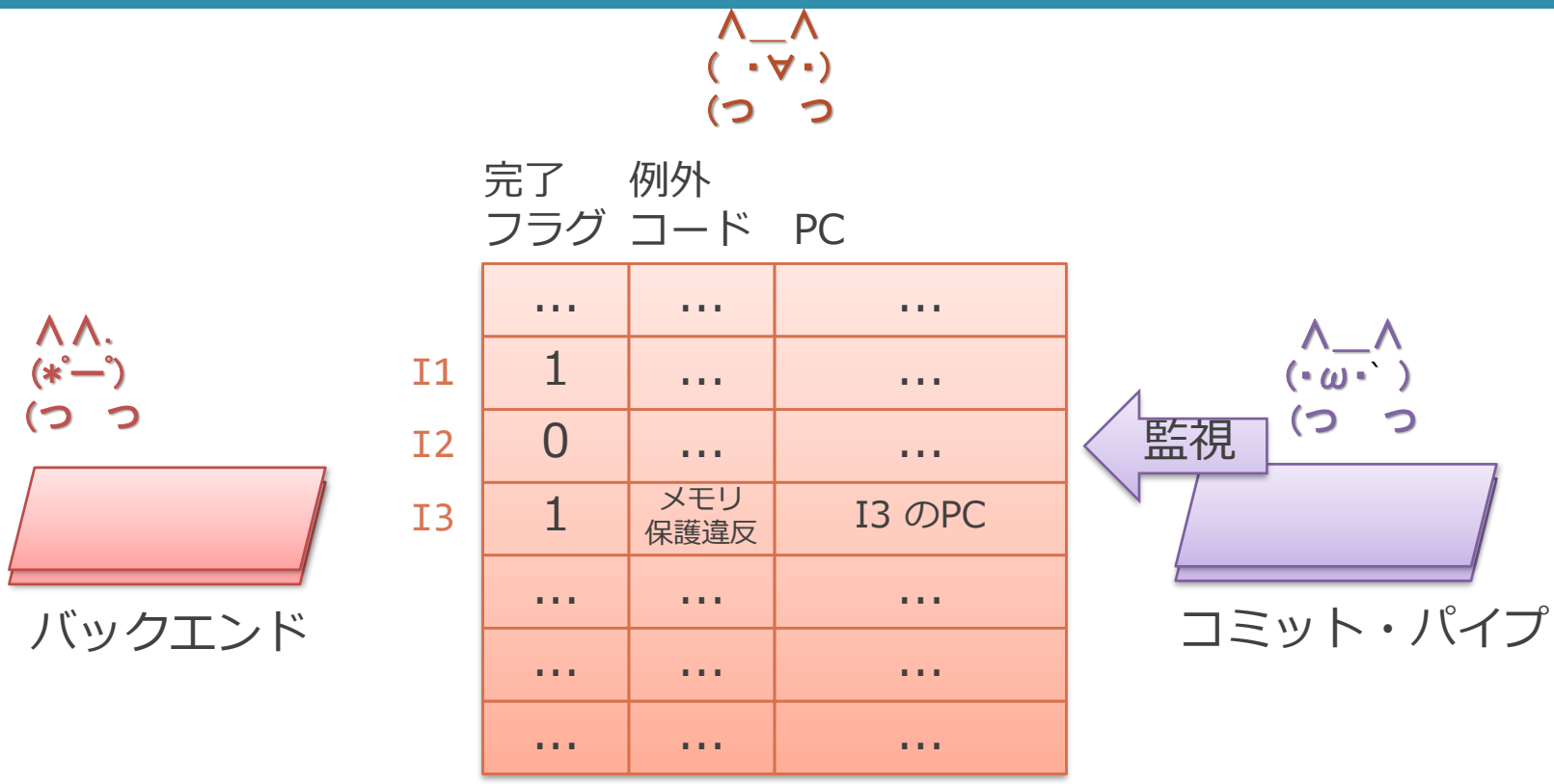
- I3 が完了したので，ROB に書き込みを行う
  - ◇ 完了フラグに１を書く
  - ◇ 例外を発生させていたので，その種類と自分の PC も書く

ROB の動作（４）



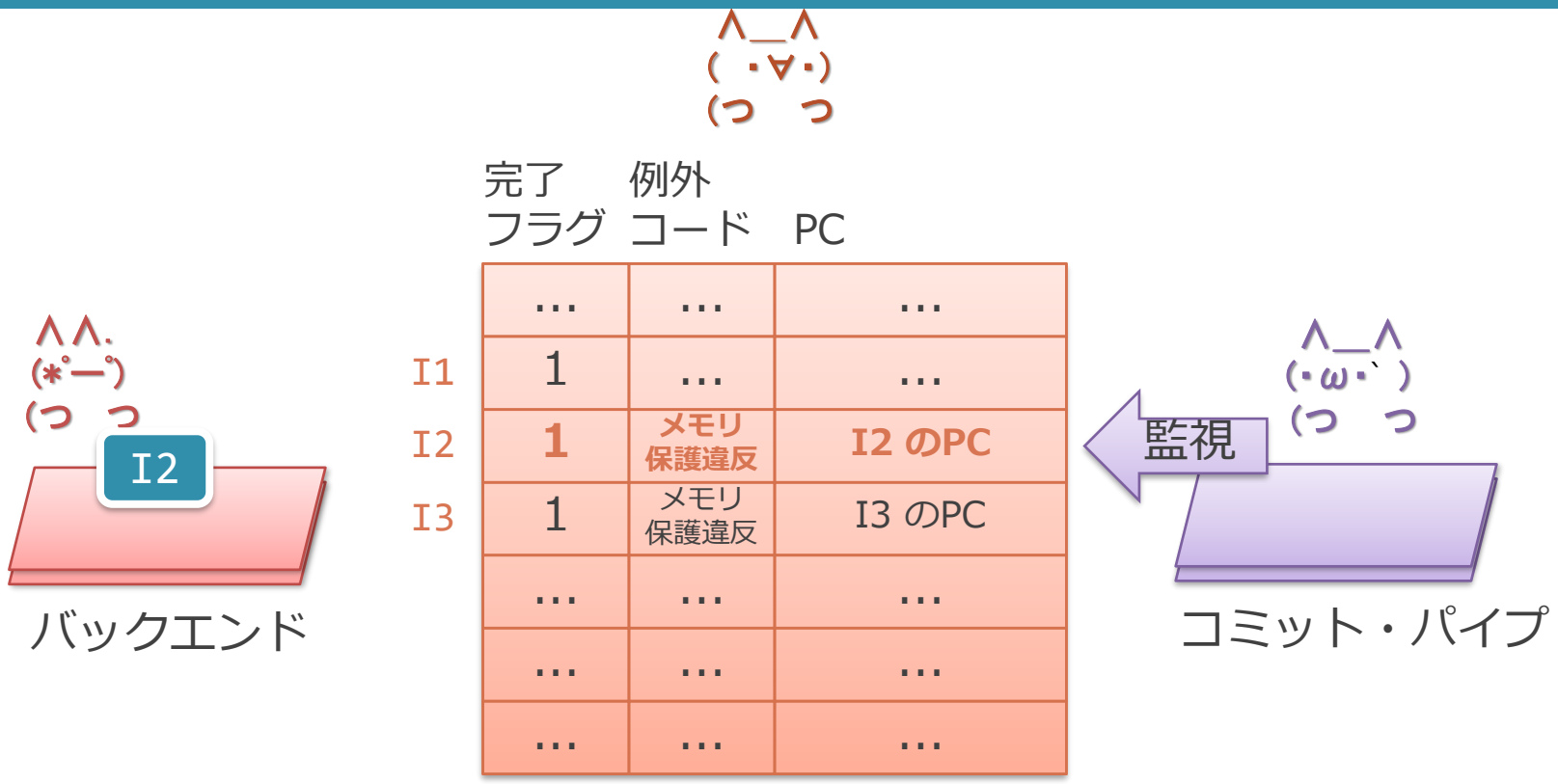
- コミット・パイプの監視ポイントが I1 に移動してきた
  - ◇ 直前の命令のコミットが終わった
  - ◇ I1 の完了フラグが 1 であるため，I1 をコミット
  - ◇ コミット・パイプはバックエンドとは独立して並列に動作

ROB の動作（５）



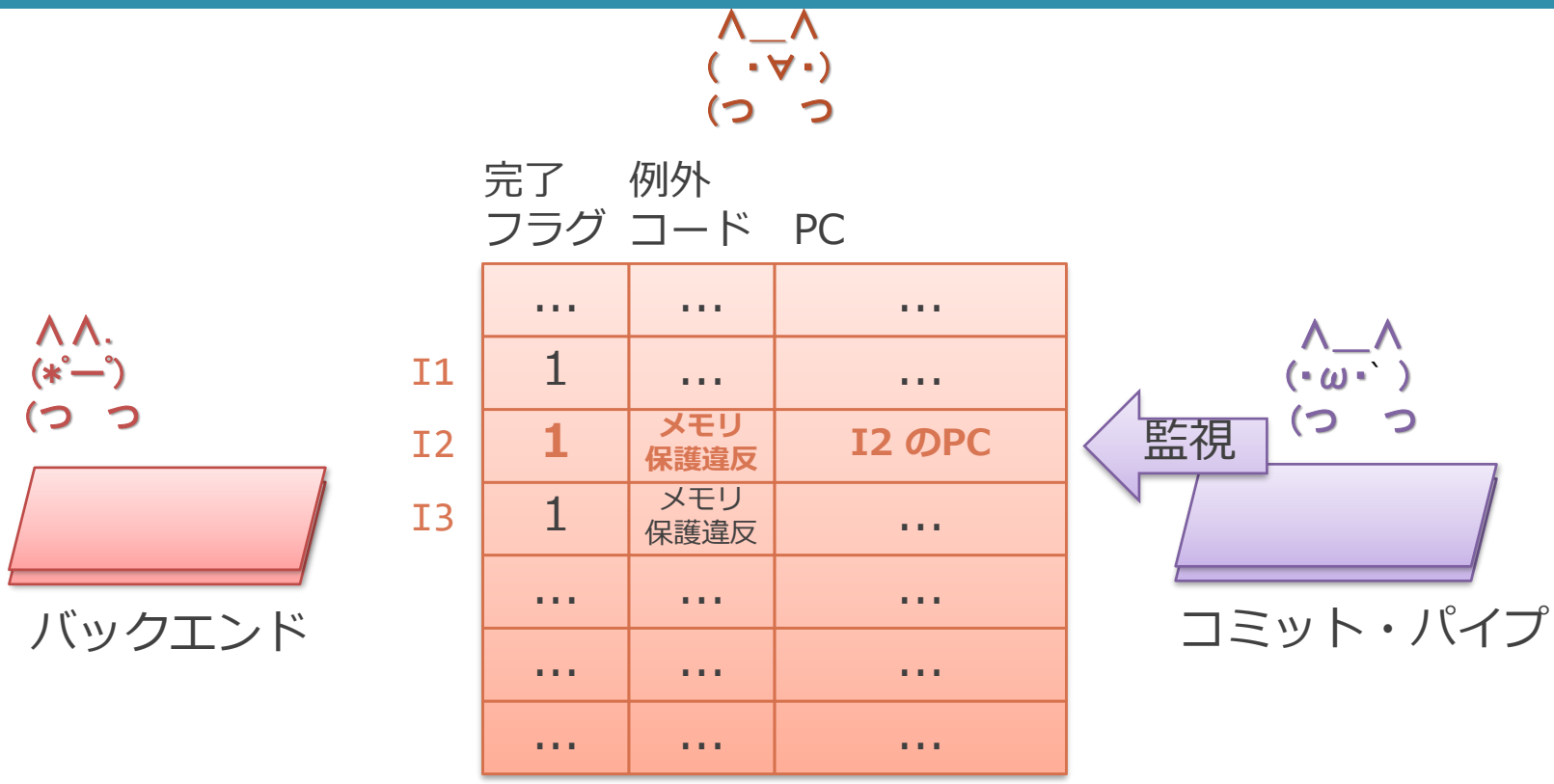
- コミット・パイプの監視ポイントが I2 に移動
  - ◇ I2 の完了フラグはまだ 0 なのでコミットはしない

ROB の動作（５）



- I2 が完了
  - ◇ 例外を発生させていたので，それを ROB に書き込む

ROB の動作（６）

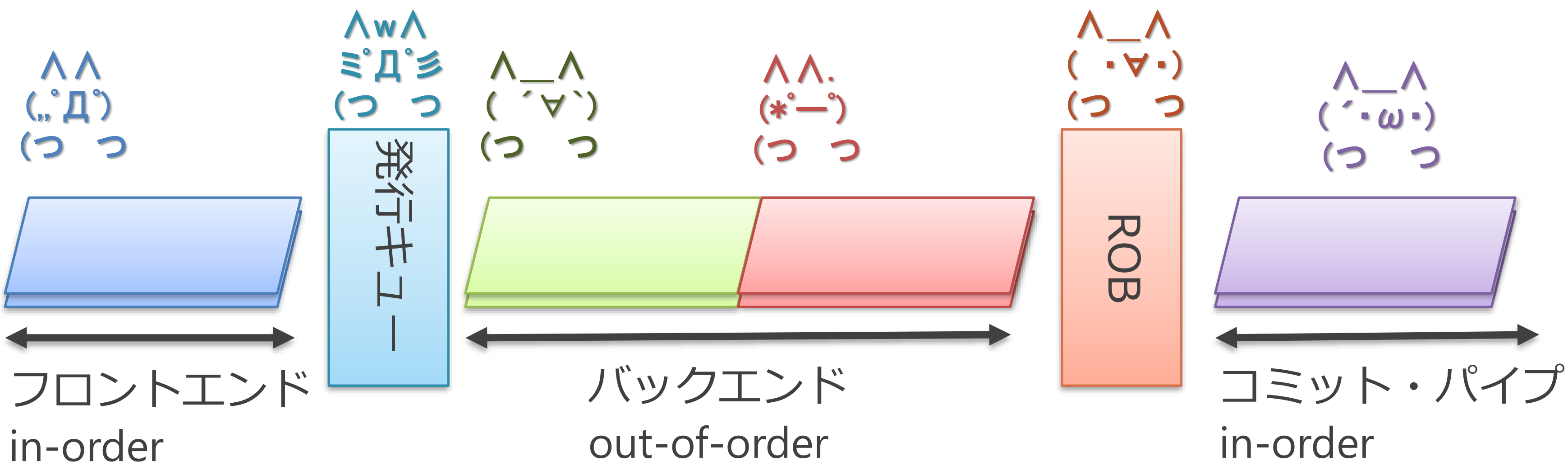


- 監視対象の I2 のエントリが完了している
  - ◇ 例外コードが記録されているので例外の処理を行う
    - □ PC を例外ハンドラのアドレスに書き換え
    - □ 後で戻ってこれるように I2 の PC を特別なレジスタに保存
  - ◇ I3 以降を全部取り消し

分岐予測ミスへの対処

- 分岐予測ミスもこれと同様に回復可能
  - ROB に分岐予測がミスであったことと，正しい飛び先を格納
    - メモリ保護違反では，その命令の PC を格納していた
  - 「予測がミスであった」という例外コードを書く
  - コミット・パイプで PC を正しい飛び先に更新
- これだと回復の開始が遅いため，完了時に out-of-order に回復を行うこともある
  - すべての予測ミスの回復を行えば，最終的に辻褄はあう

ROB のまとめ



- ROB の動作
  - ◇ バックエンドから out-of-order に完了状態を書き込み
  - ◇ コミット・パイプで in-order に読み出す
    - □ 元のプログラム順に実行したときの状態を再現

今回の内容

1. 動的スケジューリングの詳細
    1. 例外への対応
    2. **ロード・ストアへの対応**
2. GPU のアーキテクチャ概要

ロード・ストアへの対応

- 命令の発行が out-of-order
  - ◇ 発行の順序はレジスタによる真の依存のみを守る
  - ◇ ロードやストアのアクセス先アドレスは関知しない
    = 実行の正しさが保たれない場合がある

実行結果がおかしくなる例

1. ストア→ロード間の順序の違反
   - ◇ 同じアドレスに対するストアとロードの順序が変わる場合
   - ◇ I2→I1 の順で発行されると，x1 が書かれる前の値が x2 に

```
I1: sw x1→(0x1000)
I2: lw x2←(0x1000)
```

2. ロード→ストア間の順序の違反
   - ◇ 同じアドレスに対するロードとストアの順序が変わる場合
   - ◇ I2→I1 の順で発行されると，x1 が書かれた後の値が x2 に

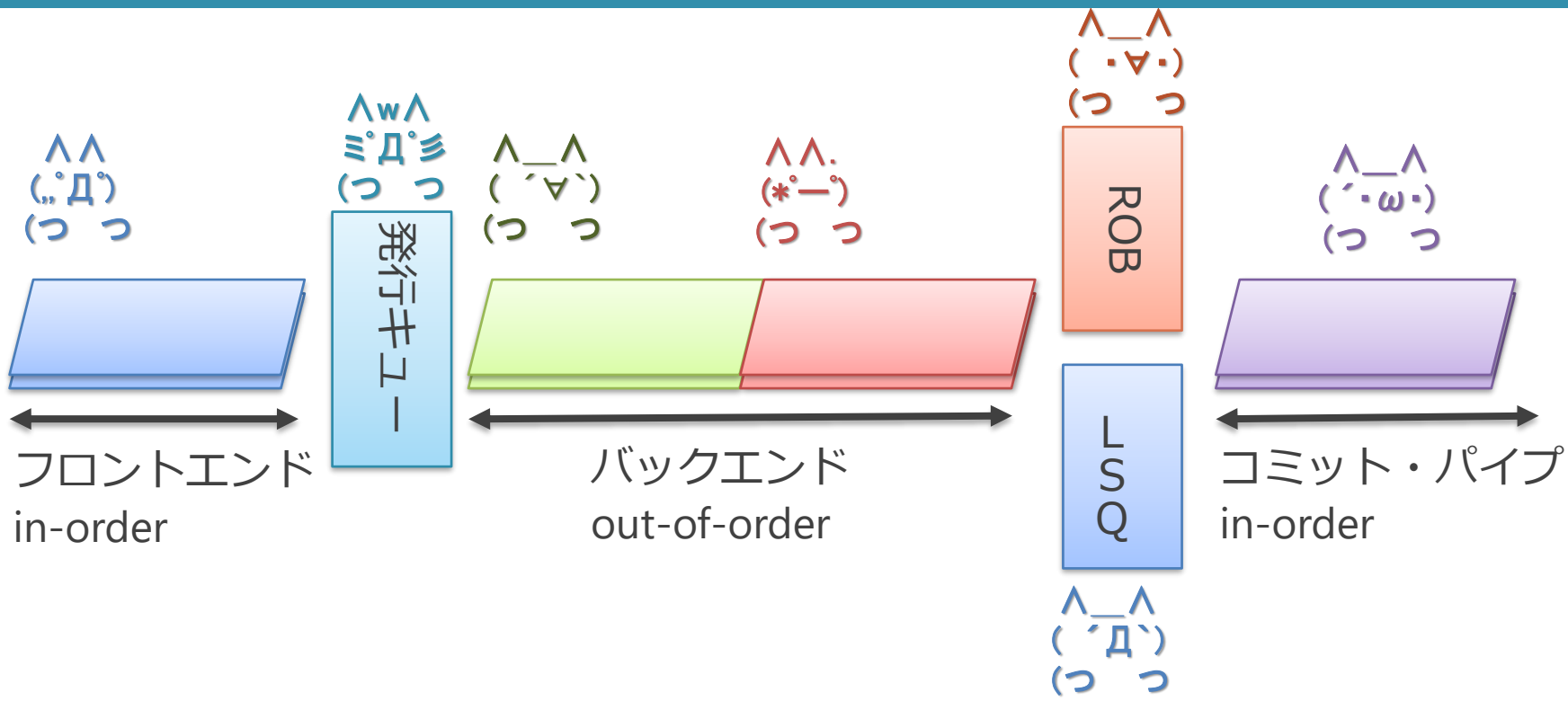
```
I1: lw x2←(0x1000)
I2: sw x1→(0x1000)
```

実行結果がおかしくなる例

3. ストア間の順序の違反
   - ◇ 同じアドレスに対する複数のストアの順序が変わる場合
   - ◇ `I2→I1` の順で発行されると，アドレス `0x1000` に `x1` が残る

```
I1: sw x1→(0x1000)
I2: sw x2→(0x1000)
```

ロード・ストア・キュー（LSQ: Load Store Queue）



- LSQ：
  - ◇ ロードやストアの実行結果を完了時に書き込むバッファ
  - ◇ バックエンドとコミット・パイプ（とメモリ）の間にある

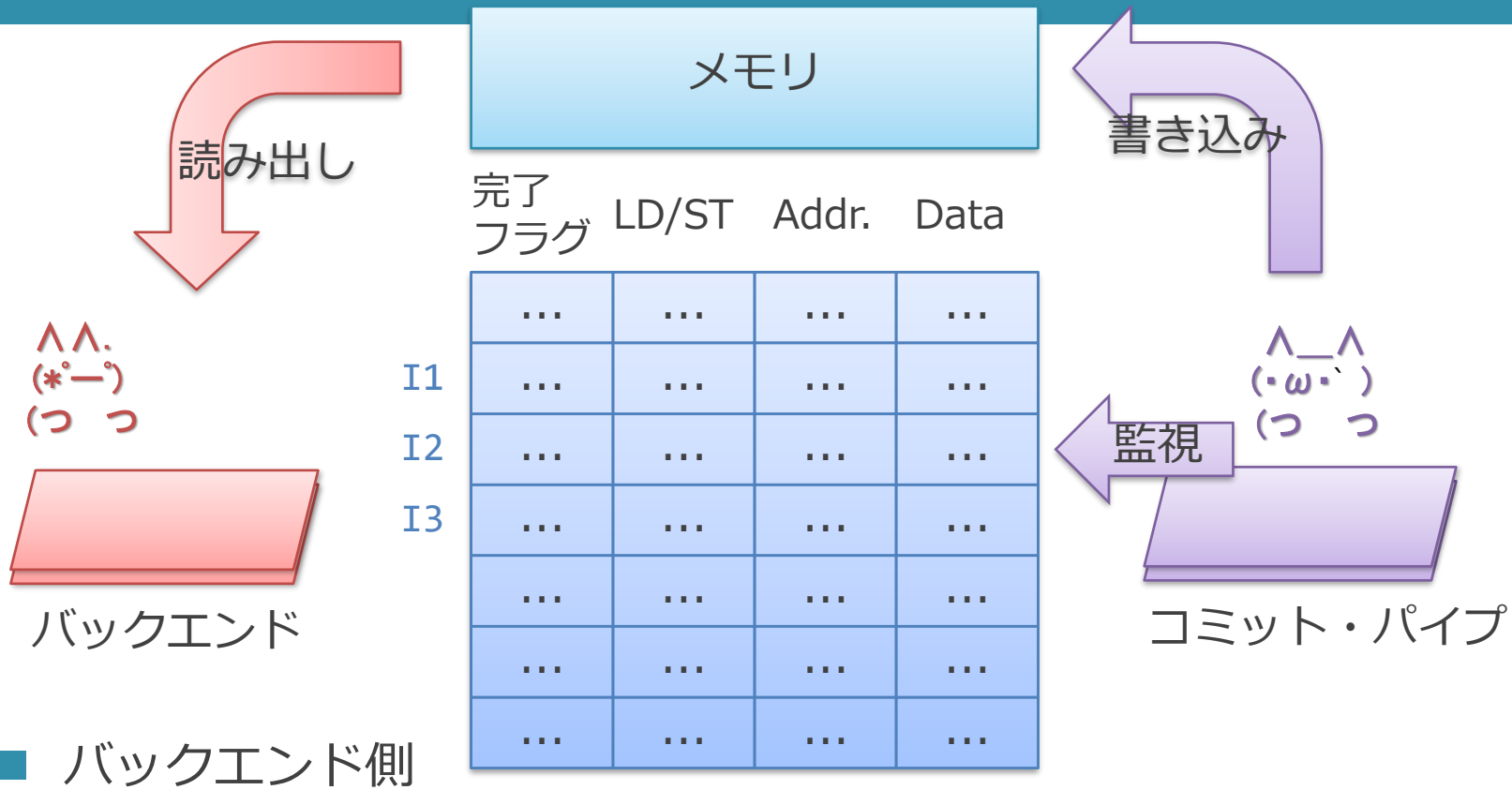
LSQ の役割

1. ストアの整列
   - ◇ 複数のストアのデータを保持し，プログラム順にメモリに書き込む
   - ◇ ストアからロードへのフォワーディング
2. 順序違反の検出
   - ◇ プログラムの意味が保たれない順序でアクセスがあった場合に，それを検出
   - ◇ 検出時は分岐予測時と同様にフラッシュしてやりなおす

LSQ の内容

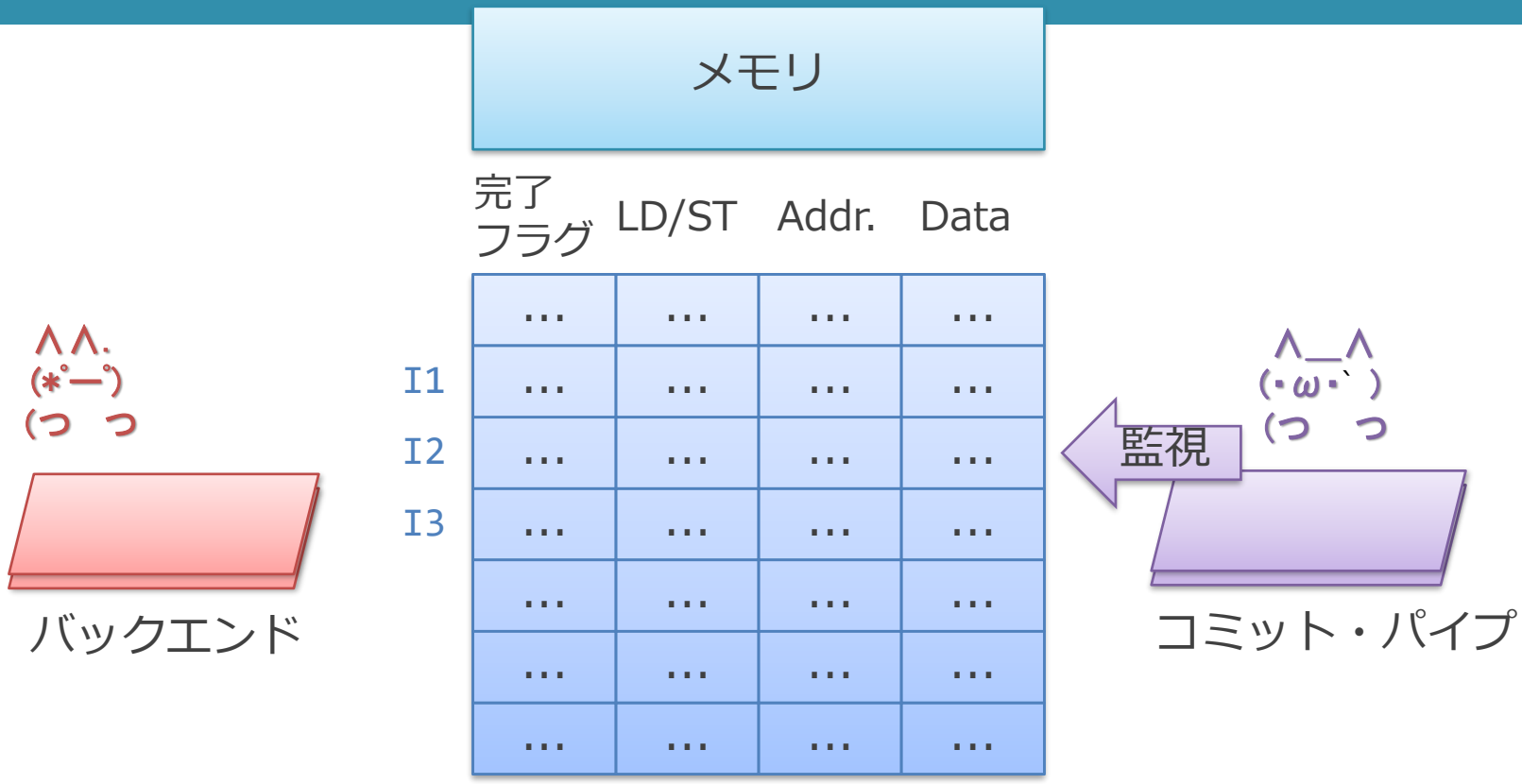
- LSQ：
  - ◇ プログラム順にエントリが確保される FIFO
  - ◇ ROB と違い，ロードとストアにのみ割り当てられる
    - □ ロードとストアのために ROB を拡張したものとも言える
- 内部にあるフィールド：
  1. 完了フラグ：
     - □ 完了した命令は 1 を書き込む
  2. ロード or ストア識別のフラグ
  3. アドレス
  4. データ

LSQ の動作



- バックエンド側
  - ◇ LSQ に in-order に命令ごとにエントリを確保
  - ◇ バックエンドで完了した命令は out-of-order に書き込みを行う
- コミット・パイプ側
  - ◇ in-order に「ここまで完了した」部分を監視

動作例を使った説明



- `I1,I2,I3` は同じアドレス `0x100` に対してアクセスを行う命令
  1. ストアの整列
  2. 順序違反の検出

ストアの整列（１）

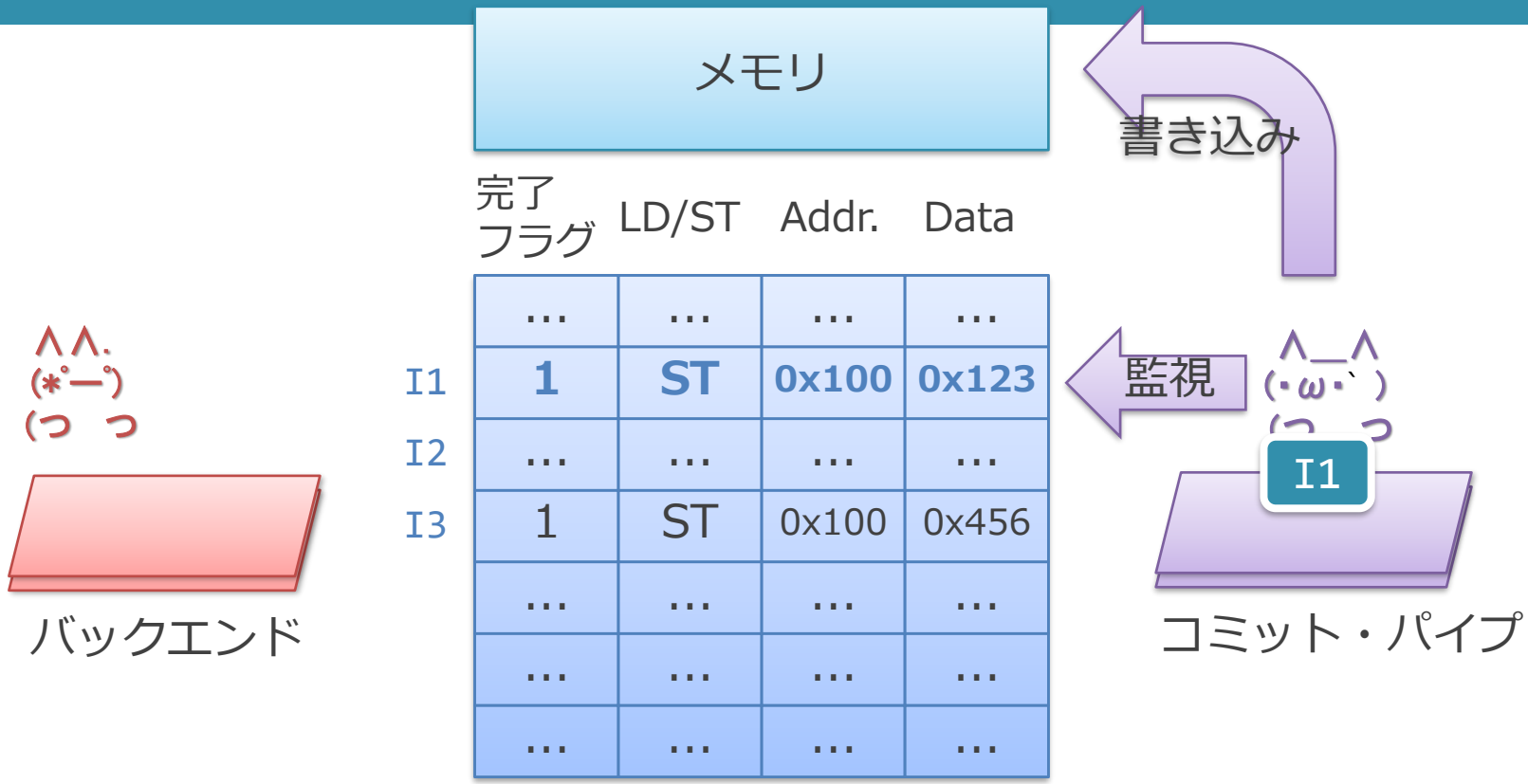
- ストア実行時
  - ◇ バックエンドから LSQ にアドレスとデータを書き込む
  - ◇ バックエンドからはメモリには書き込まない

- `I1:SW x1→(0x100)` が完了

ストアの整列（２）

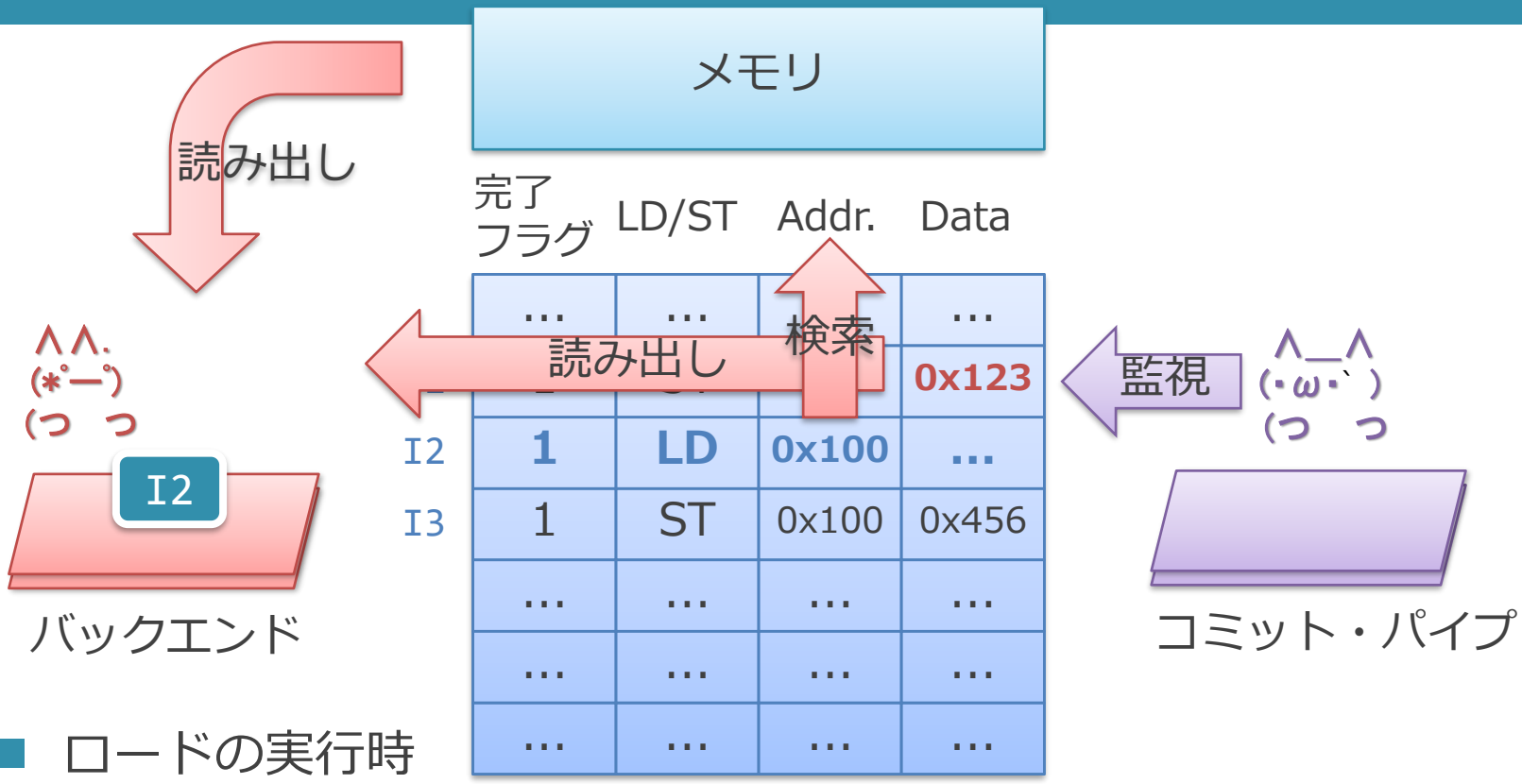
- I3:SW x2→(0x100) が完了
  - ◇ 対応するエントリに同様に書き込む

ストアの整列（３）



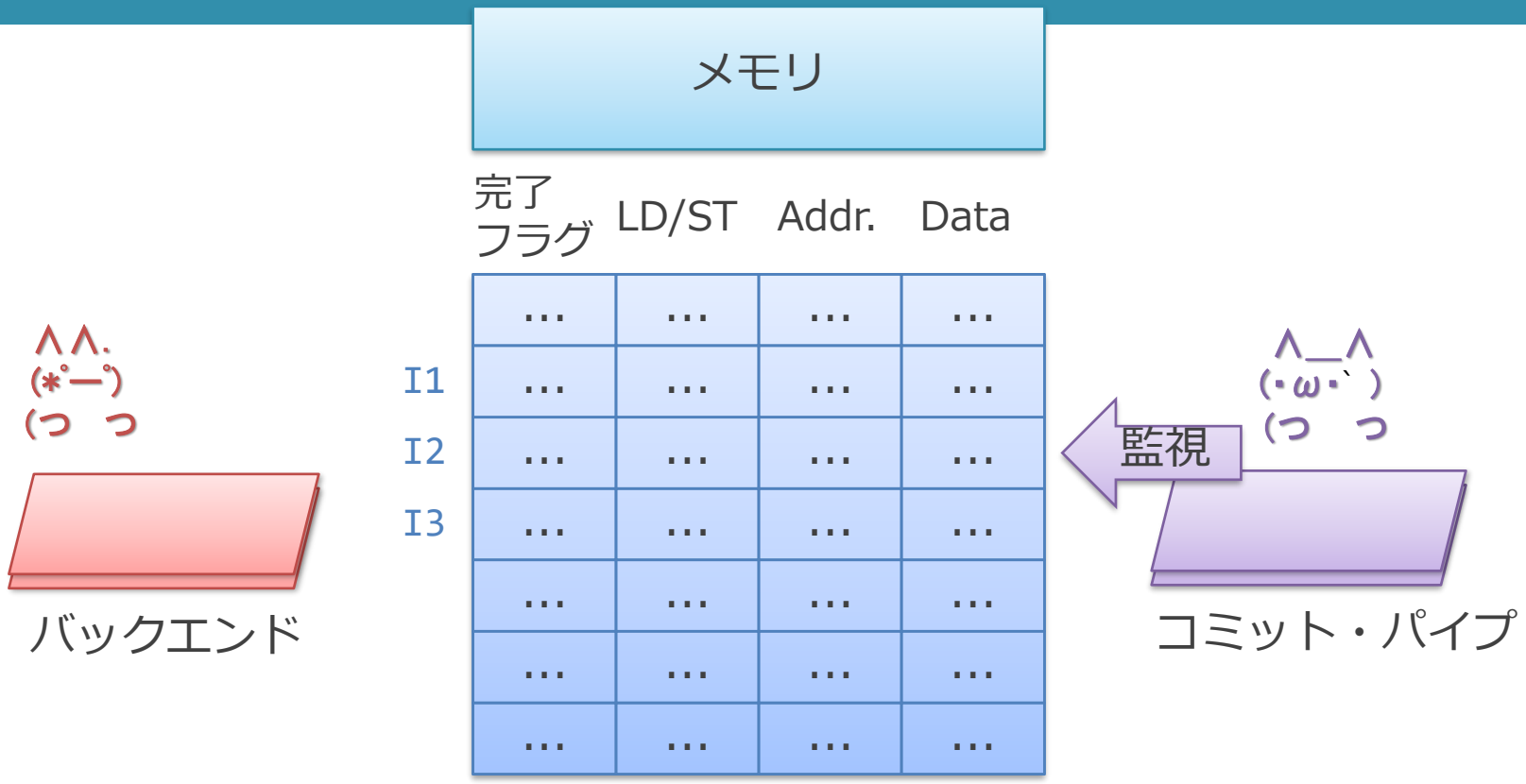
- 監視対象が I1 に移り，完了していることを検出
  - ◇ コミット・パイプからメモリにデータを書き込む
- 監視ポイントがプログラム順に下に移動していく = ストアの整列
  - ◇ 元のプログラム順でメモリに対してデータが書き込まれる

ストアの整列（４）



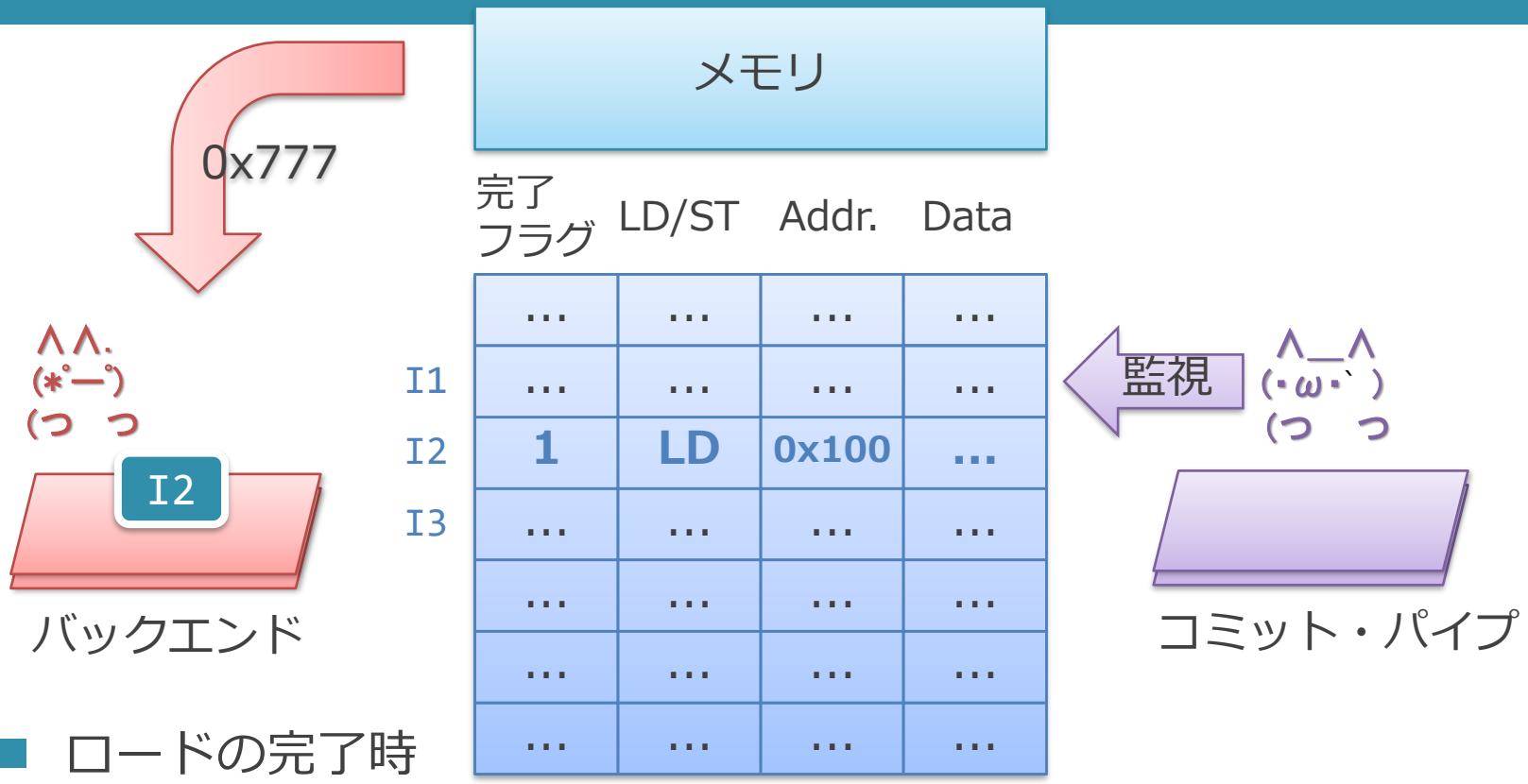
- ロードの実行時
  - ◇ LSQ とメモリの双方にアクセス
- LSQ の自身のエントリより前の部分をアドレスで検索
  - ◇ ヒットした場合，そこにあるストアの値を読み出して使う
  - ◇ ヒットしなかった場合，メモリからとれた値を使う

動作例を使った説明



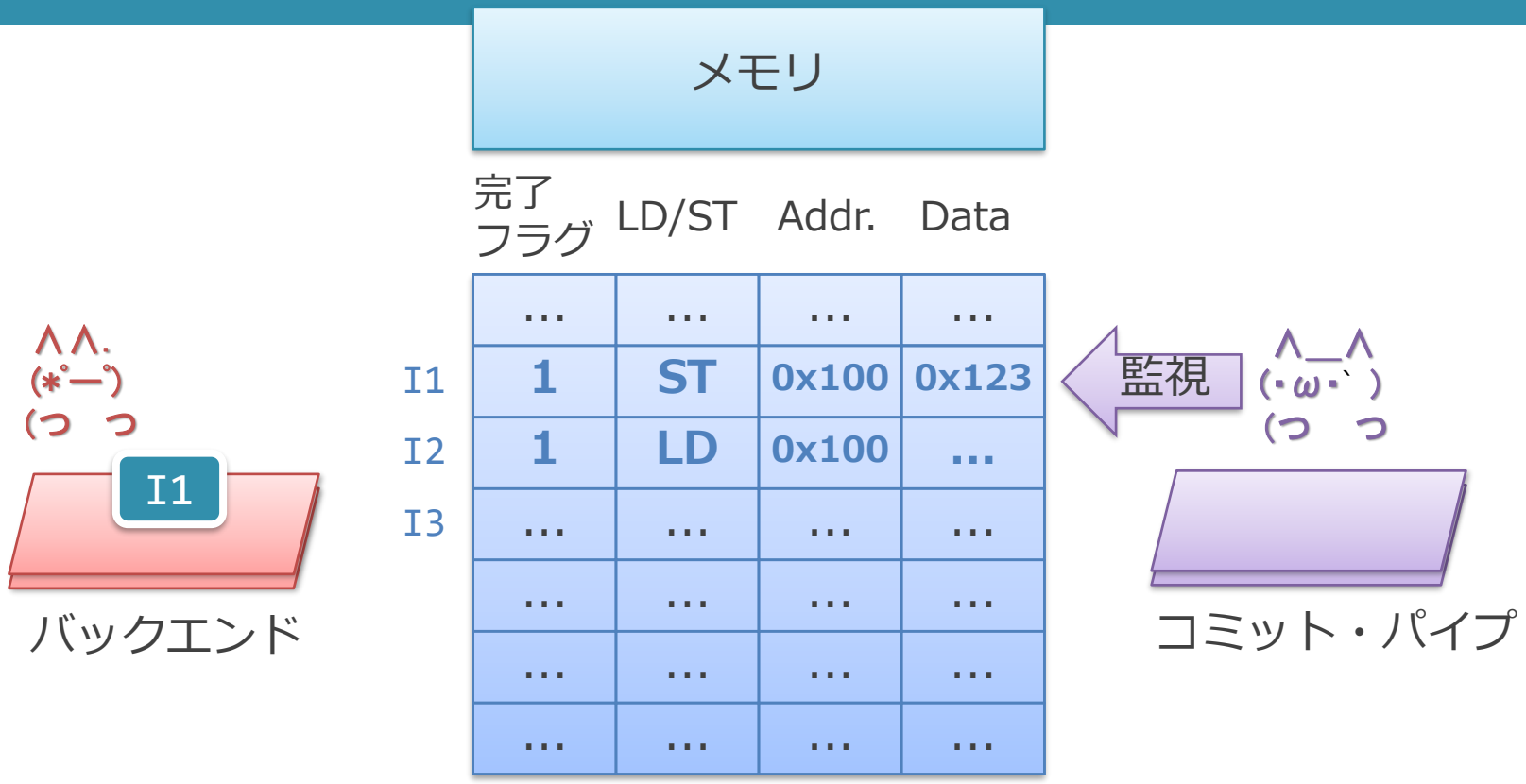
- `I1,I2,I3` は同じアドレス `0x100` に対してアクセスを行う命令
  1. ストアの整列
  2. **順序違反の検出**

順序違反の検出（１）



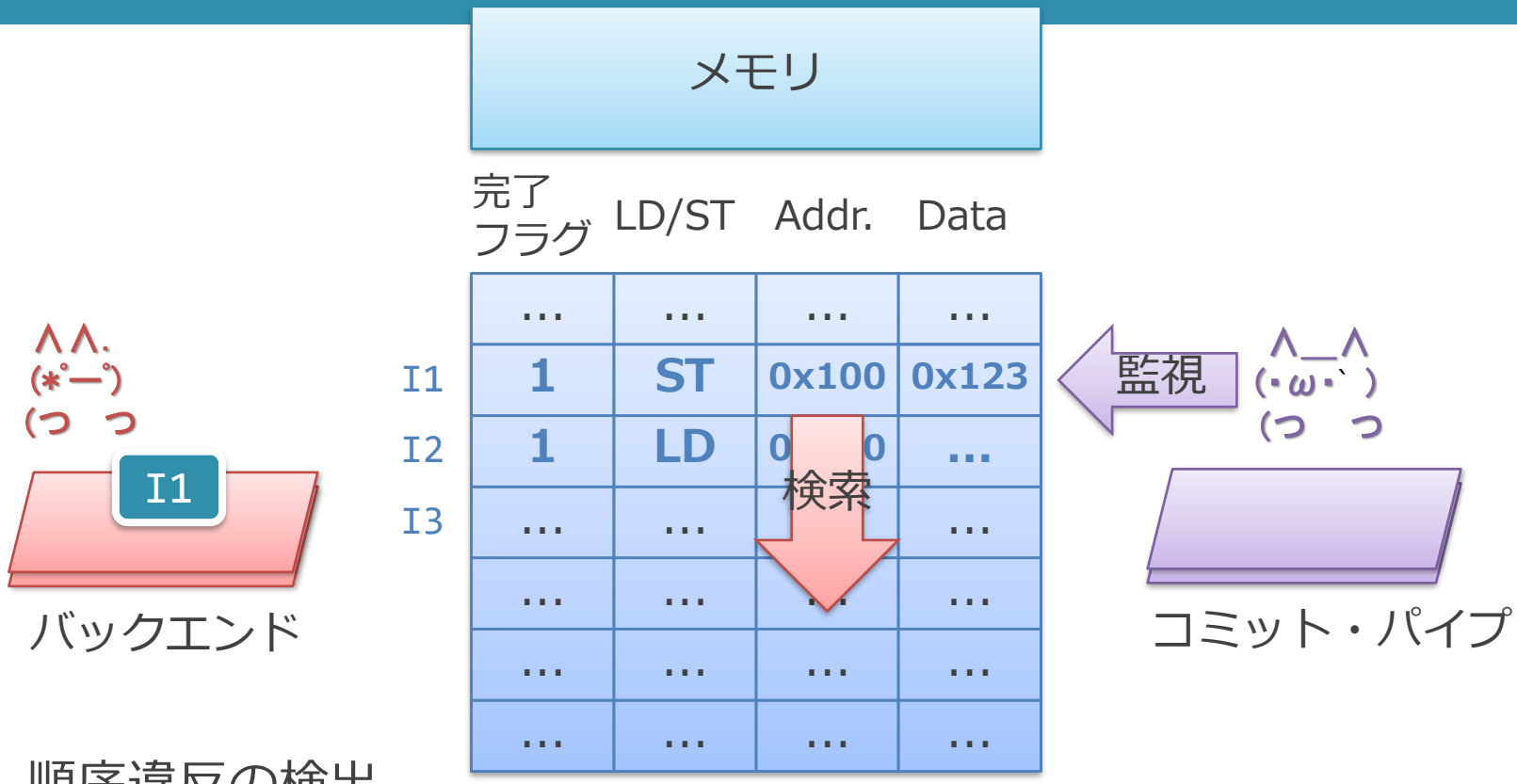
- ロードの完了時
  - ◇ LSQ に完了フラグやアドレスを書き込む
- 上では I2 のロードが I1 よりも先に実行
  - ◇ LSQ 内には検索しても 0x100 に該当するものはない
  - ◇ メモリからは 0x777 がとれたので，これを使用する

順序違反の検出（２）



- I2 の後に I1 が実行
  - ◇ I2 と同じアドレスであった
  - ◇ I2 は本当は 0x777 ではなく 0x123 を読まなければならなかった = 順序違反

順序違反の検出（３）



- 順序違反の検出
  - ◇ ストアの完了時に，自分より後ろのロードのアドレスを検索
  - ◇ すでに完了しているロードでアドレスがヒットした場合
    - □ 本来はそのロードはそのストアのデータを読むべきであった
    - □ 順序違反として検出し，自分より後ろを全部消してやり直す

実行結果がおかしくなる例の解決（１）

- ストア→ロード間の順序の違反
  - I2→I1 の順で発行される場合

    I1: sw x1→(0x1000)

    I2: lw x2←(0x1000)

- 動作：ロードを取り消してやり直す
  - I1 実行時に自分より後ろを LSQ から検索すると I2 がヒット
  - I2 を再実行することにより，ロードで正しい値を得る

実行結果がおかしくなる例の解決（２）

- ロード→ストア間の順序の違反
  - I2→I1 の順で発行されると，x1 が書かれた後の値が x2 に

```
I1: lw x2←(0x1000)
I2: sw x1→(0x1000)
```

- 動作：LSQ からフォワーディングはされない
  - I2 実行時にはメモリではなく LSQ に値が書き込まれる
  - I1 はメモリと，LSQ の自分より前の部分を検索
    - I2 は自分より後ろのエントリにある
    - 検索対象とならずヒットしない = メモリから値が読まれる

実行結果がおかしくなる例の解決（３）

- ストア間の順序の違反
  - I2→I1 の順で発行されると，アドレス 0x1000 に x1 が残る

    I1: sw x1→(0x1000)

    I2: sw x2→(0x1000)

- 動作：SQ から in-order にメモリに書き込み
  - I2 と I1 の順で LSQ にアドレスとデータが書き込まれる
  - コミット・パイプが I1 と I2 の順でアドレスとデータを読み出し，メモリに書き込む

LSQ のまとめ

- out-of-order にロードとストアが実行されてもプログラムの正しさを保つ必要がある
- LSQ：ロードとストアの結果を保持するバッファ
  - ◇ ストアの整列
  - ◇ 順序違反の検出

メモリ依存予測

- ここまでの例：
  - ロードやストアは，レジスタの依存が解決し次第実行
  - 大概，各ストアとロードはアドレスが違うので問題ない
- より高い性能を得たい場合
  - 一部状況では，順序違反が頻発する
  - ロードとストア間で依存関係の予測を行う
    - 依存元のストアが実行されるまでロードの実行を遅らせる
- メモリ依存予測
  - ロードが依存するストアの集合を予測
  - ストア・セット予測器という予測方式が提案されている
    - 理想的な予測を行った場合とほぼ同等の性能がでる

投機的なロード・ストアの発行

- この講義で説明したのは投機的にロード・ストアを発行する方法
  - ◇ 現在の CPU では主流
  - ◇ 教科書にはあまり書かれていないので注意
- よく教科書に「メモリ曖昧性除去（memory disambiguation）」として書いてある別の方法：
  1. ロードやストアを，アドレス計算とメモリ・アクセスに分離
  2. アドレス計算だけとりあえず先にやる
  3. ロードは自分よりプログラム順で前にあるストアのアドレス計算が全部終わっていたら発行
- 上記の方法より投機的に発行する方法の方が速い
  - ◇ ストアのアドレスの確定を待たなくても良いため

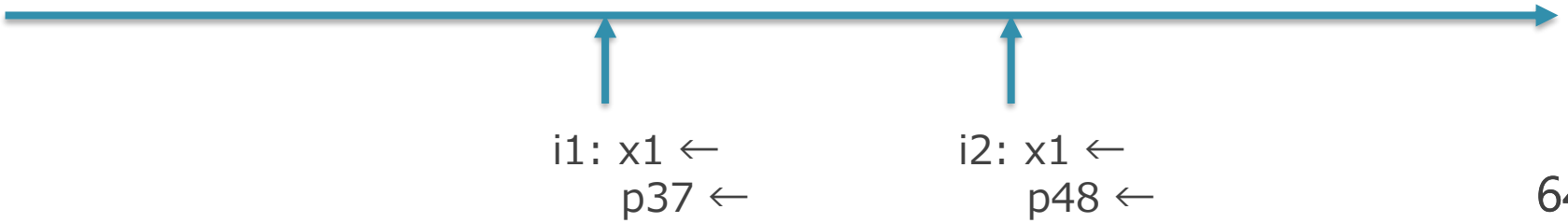
動的命令スケジューリングについての補足

- 今回省いた話題：物理レジスタの解放方法
  - 方法１：参照カウンタを使ってカウンタが０になったところで解放する
    - ハードが複雑になりがち
  - 方法２：論理的にアクセスされないことが確定したタイミングで解放する
    - こっちが主流

論理的にアクセスされないことが確定したタイミングで解放する

- あるレジスタ r に書き込む命令がコミットした際に，
  そのレジスタ r に前回書き込んだ命令に割り当てられた物理レジスタを解放

- 下記の例：
  - ◇ i1 と i2 は同じ論理レジスタ x1 に書き込んでいる
    - □ i1 では p37 に，i2 では p48 に物理レジスタを割り当て
  - ◇ i2 のコミット時に，i1 に割り当てられた p37 を解放
  - ◇ i2 がコミットされるということは，
    - □ そこより左にある命令は全て実行済み
    - □ そこより右にある命令は x1 を参照すると p48 が見えるので，
      左側の p37 が参照されることはもうない



動的命令スケジューリングについての補足

- この講義で扱った out-of-order の方式：Alpha 21264 のもの
  - 古典的な「トマスロのアルゴリズム」とは少し違うので注意
    - 単一の物理レジスタ・ファイルと RMT によるリネーム
    - 値は ROB に書き込まず，ブロードキャストもしない
    - ロードやストアは投機的に実行され，順序違反を検出
- Intel Core i7 の後期や AMD Ryzen もこれと同等
  - ほぼすべての CPU がこのやり方に収斂した
- 教科書等とやりかたが何か違う？と思った場合は，実際違う
  - この講義で書いたやり方ベースで書かれた教科書は
    たぶんまだない

今回の内容

1. 動的スケジューリングの詳細
    1. 例外への対応
    2. ロード・ストアへの対応
2. **GPU のアーキテクチャ概要**

GPU：Graphics Processing Unit

- GPU
  - ◇ もともとはグラフィックを高速に処理するためにあった
- GP-GPU（General Purpose computing on GPU）
  - ◇ グラフィック以外に，汎用の計算に GPU を使用する使い方
  - ◇ 大量の単純な処理の繰り返しが得意
    - □ 行列積 → 機械学習
    - □ 仮想通貨のマイニング
  - ◇ GP-GPU での使用を前提として説明
- ここでは主に NVIDIA の GPU をある程度前提として説明

GPU のプログラム実行イメージ

- 行列積の普通に C で書いたコード

```
for (k = 0; k < 1024; k++) // ３重ループ
  for (j = 0; j < 1024; j++)
    for (i = 0; i < 1024; i++)
      a[j][i] += b[j][k] * c[k][i];
```

- OpenCL や CUDA のイメージ

```
// func が 1024 個起動されて並列に実行される
func(thread_id) {
  i = thread_id; // 0-1023 がスレッド ID として渡される
  for (k = 0; k < 1024; k++) // ２重ループ
    for (i = 0; i < 1024; i++)
      a[j][i] += b[j][k] * c[k][i];
}
```

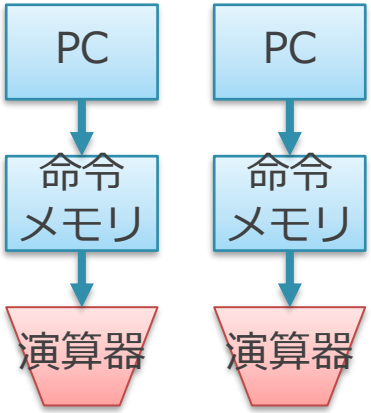
GPU で動くプログラムに期待できる性質

1. 並列に動作する大量のスレッドがある
   - ◇ ループの各イタレーションが各スレッドに相当
2. 各スレッドは比較的短い
   - ◇ 元がループの各イタレーションなので，短い
3. 各スレッドは基本的に同じことをしている
   - ◇ 元がループなので

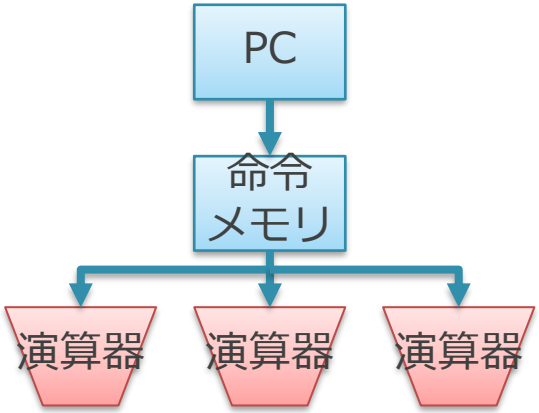
SIMD（Single Instruction Multiple Data）

- 単一の命令で複数のデータを演算（複数の演算器を駆動）
  - ◇ 命令メモリやデコードなどのハードが省ける
  - ◇ 単一の命令で 16 個ぐらいの演算器を駆動
- GPU では各スレッドは基本的に同じことをしている
  - ◇ 基本的には SIMD アーキテクチャで実行可能
  - ◇ 全スレッドが同じプログラムを実行 = 各演算器で並列に実行

CPU （のマルチコア）



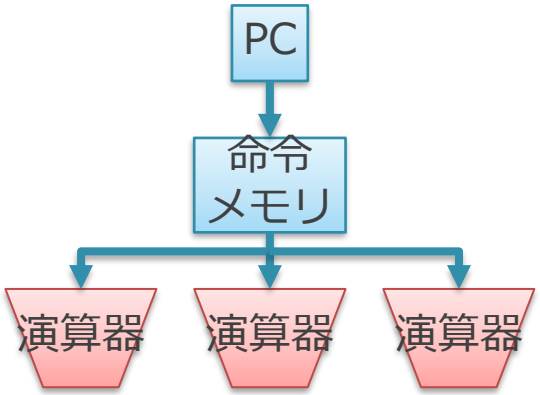
SIMD アーキテクチャ



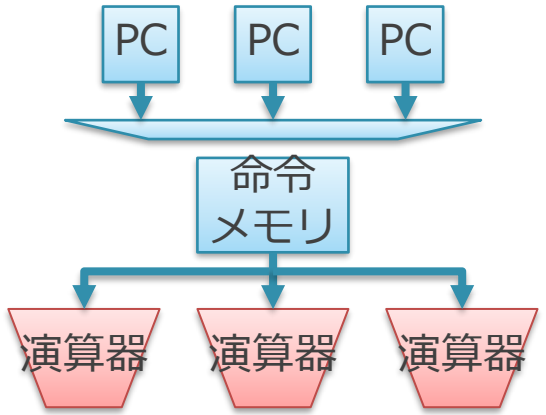
NVIDIA の SIMT（multiple thread）

- SIMD だと，全ての演算器は全く同じ動きしかできない
  - ◇ スレッドごとに個別に分岐ができない
- SIMT アーキテクチャ：各スレッドごとに違う PC を用意
  - ◇ 全員が同じパスにいる限りは SIMD と同じように動作
  - ◇ 分岐して違うパスにいった場合は，それぞれ時分割して処理
    - □ = 分岐が違うパスにいきまくると凄い性能が下がる

SIMD アーキテクチャ

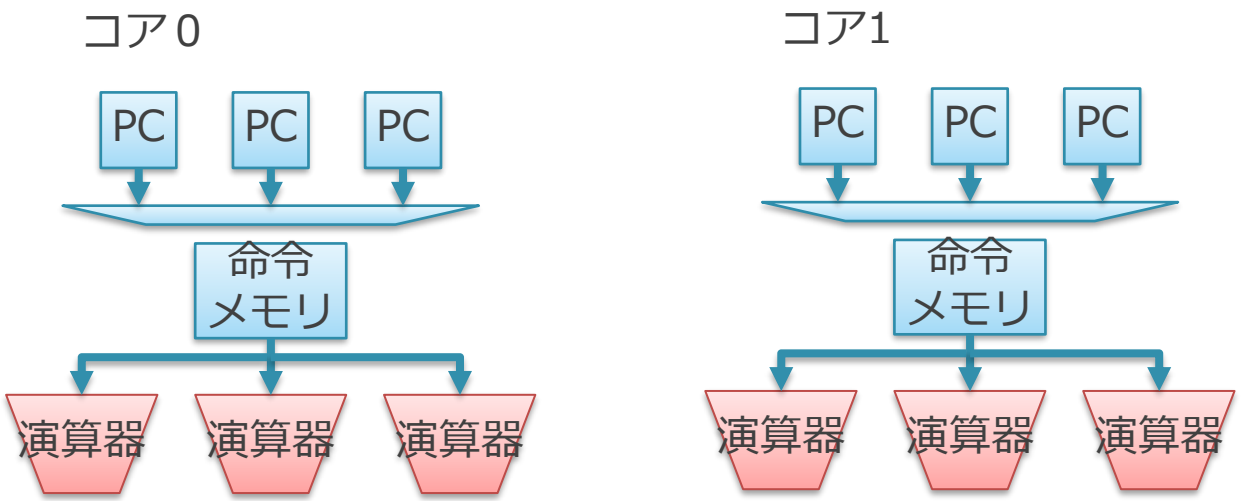


SIMT アーキテクチャ



NVIDIA の SIMT（multiple thread）

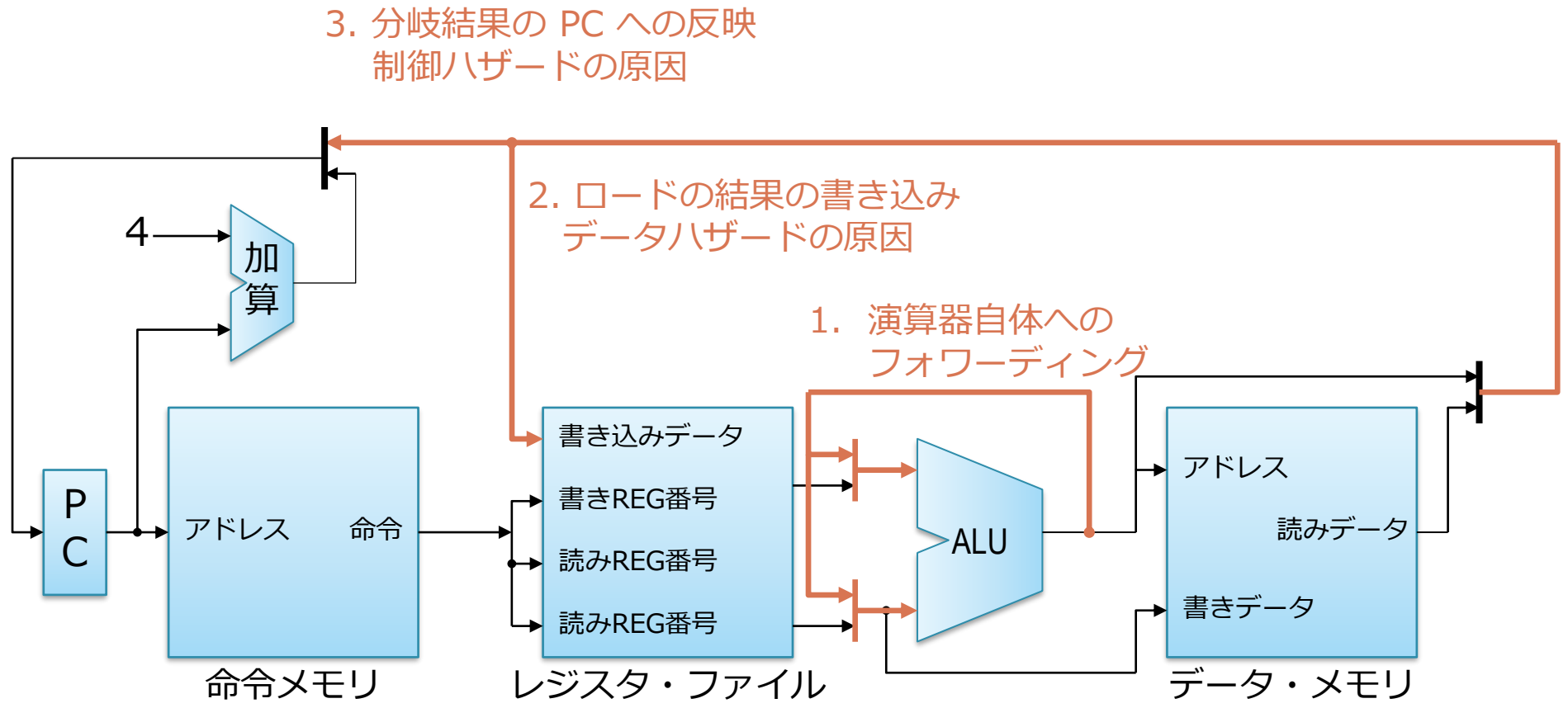
- NVIDIA 用語
  - スレッド（thread）：
    - 下記図の各 PC に対応
  - ウォープ（warp）
    - まとめて実行されるスレッドのまとまり
      - ウォープごとには違う経路の実行を時分割なしに実行できる
    - 下記図のコア単位でのまとまりに対応
      - 実際は各コアは複数のウォープのコンテキストを持っている



バックエッジに対する対処

- CPU の場合
  - フォワーディング，分岐予測，命令スケジューリング…
- GPU の場合
  - 上記を一切行わない
  - 全部マルチスレッディングで対処
    - 前ページの用語に従うと，マルチウォープ実行
    - 以降の説明ではマルチスレッドの言葉で説明

CPU のバックエッジ：逆方向（右から左）

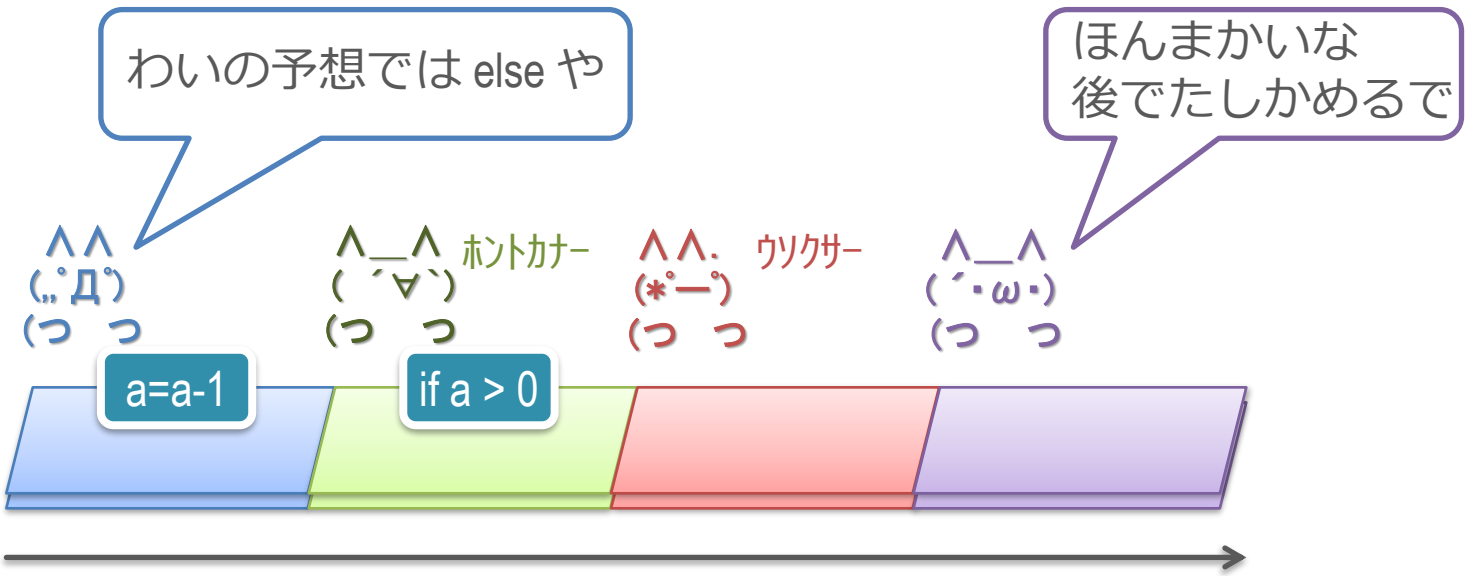


- バックエッジがあるため，命令を単純に流せない場合がある
  - ◇ 工場のラインのように，一方向に流せない

分岐予測

プログラム

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

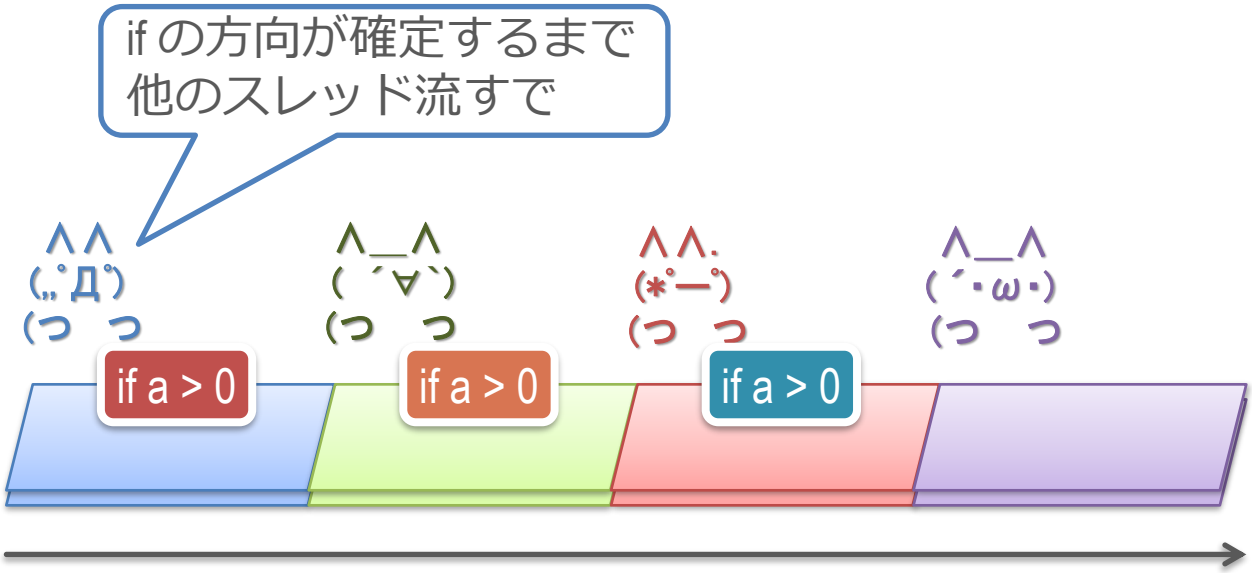


- 動作
  - 「if a > 0」の結果を予測して，命令を取り込む
    - 前回はこっちに行ったので，次もこっちに違いないとかで予測
  - あとから予測が正しいか確認する

マルチスレッドによる解決

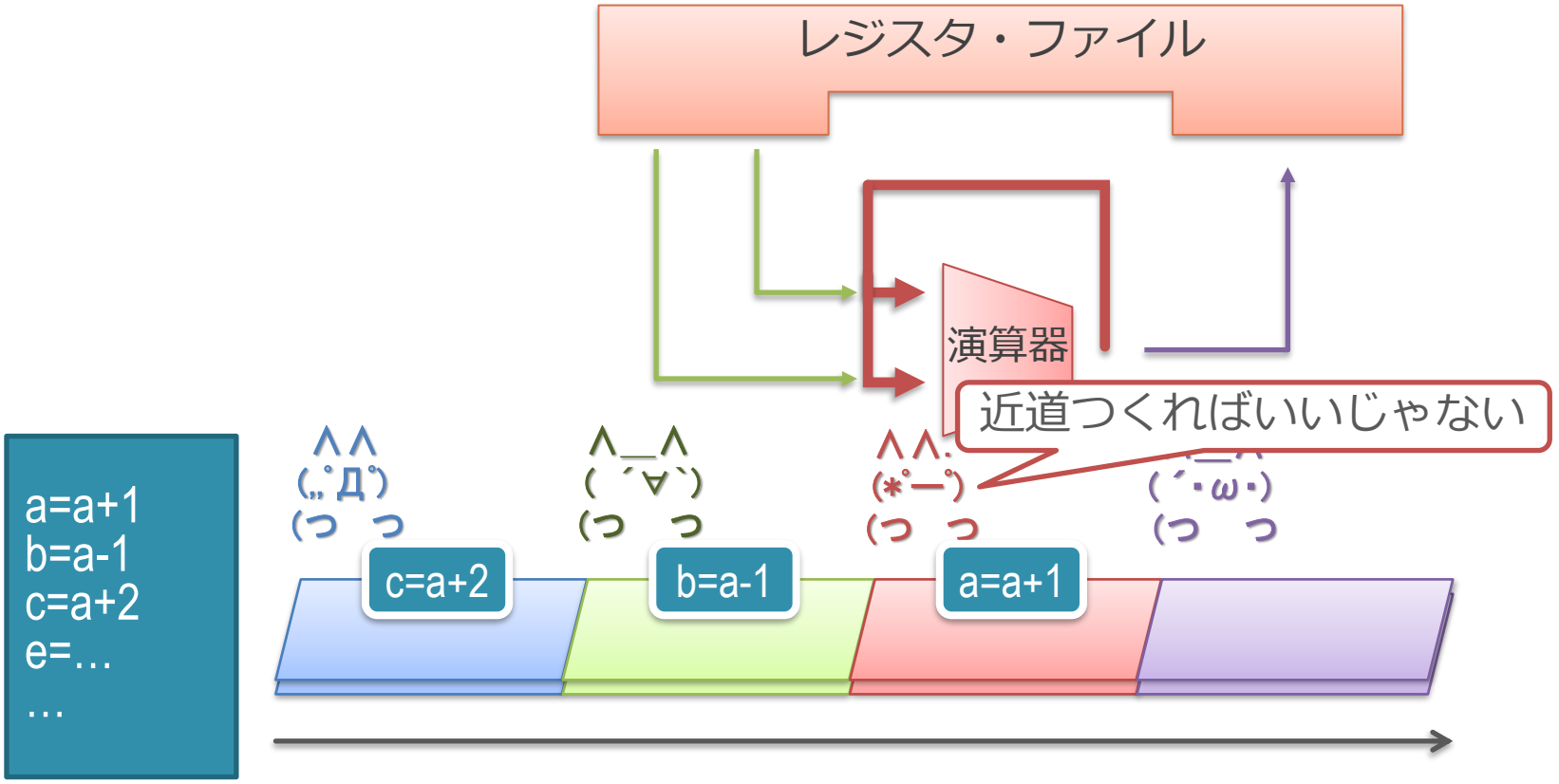
プログラム

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



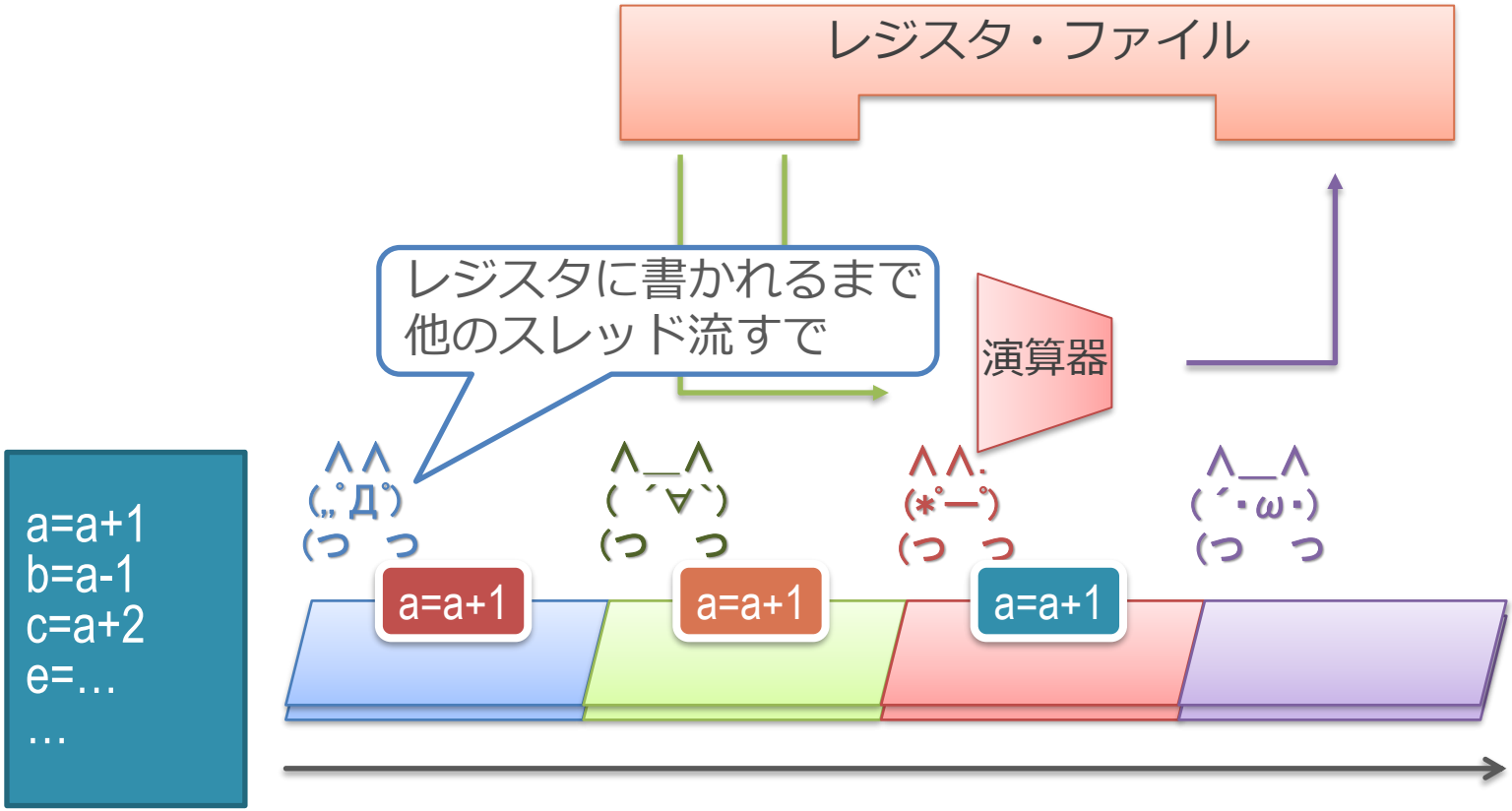
- 動作
  - 「if a > 0」の結果が出るまで他のスレッドの命令をフェッチ
  - フェッチすべき候補のスレッドは大量にある

フォワーディング



- フォワーディング（バイパスとも呼ぶ）
  - (*ﾟ―ﾟ) の人が，次のサイクルに結果を使えるようショートカットを作って手元に結果をおいておく

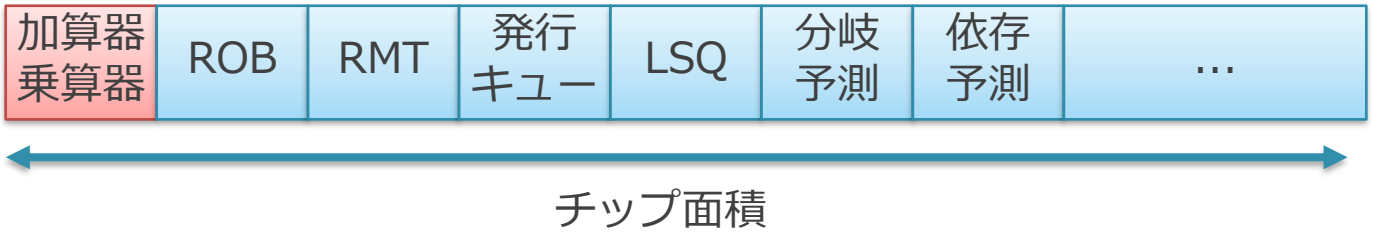
マルチスレッドによる解決



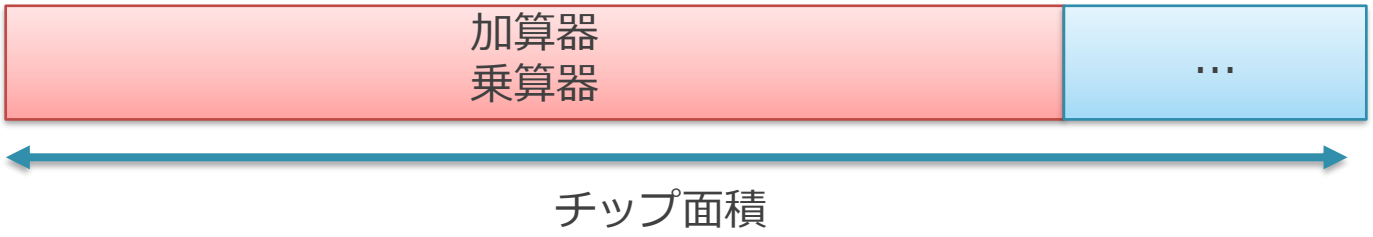
- レジスタ・ファイルに値が書き込まれるまで，他のスレッドを流す
  - ◇ 複数のスレッドの a が同時に存在するので，レジスタは大きくなる

CPU と GPUの違い
GPU では演算器搭載量を重視 = 最大性能が上がる

- CPU
  - ◇ 単一のスレッドの実行時間を短くすることに特化
  - ◇ 各種投機実行や，動的命令スケジューリングに回路資源を投入



- GPU
  - ◇ 大量のスレッドの実行スループットを上げることに特化
  - ◇ 演算器以外の部分をそぎ落とし，なるべく大量の演算器を積む



GPU の概要のまとめ

- 下記の性質を利用して，演算器以外の回路を削減
  1. 並列に動作する大量のスレッドがある
     - バックエッジは全部マルチスレッドで対処
     - 依存関係やキャッシュ・ミスで実行できない場合も，他のスレッドの実行ですます
  2. 各スレッドは比較的短い
     - 命令メモリは少しで良い
  3. 各スレッドは基本的に同じことをしている
     - SIMT により，フロントエンドを統合
- 演算器搭載量をふやして，ピーク性能を向上
  - 同じ回路面積あたりで CPU の数倍

アクセラレータや GPU は何故 CPU より速いのか？

- 演算器そのものは一緒
  - ◇ 加算器や乗算器などはどのアーキテクチャでも共通
- 演算器以外の部分をいかに減らすかにより決まる
  - ◇ 対象のプログラムの性質に特化することで，機能を削っても性能が落ちないようにする
  - ◇ 減らした分だけよりたくさん演算器がつめて電力も回せる

アクセラレータや GPU は何故 CPU より速いのか？

- シストリック・アレイ
  - ◇ GPU はまだ演算1回ごとに命令を読んだり，レジスタ・ファイルにアクセスしている
  - ◇ そもそもやる計算が決まっているなら，演算器を直接繋いでそこにデータを流せば良い
  - ◇ Google の TPU とかはこれ

Norman P. Jouppi et al., In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA 2017 より

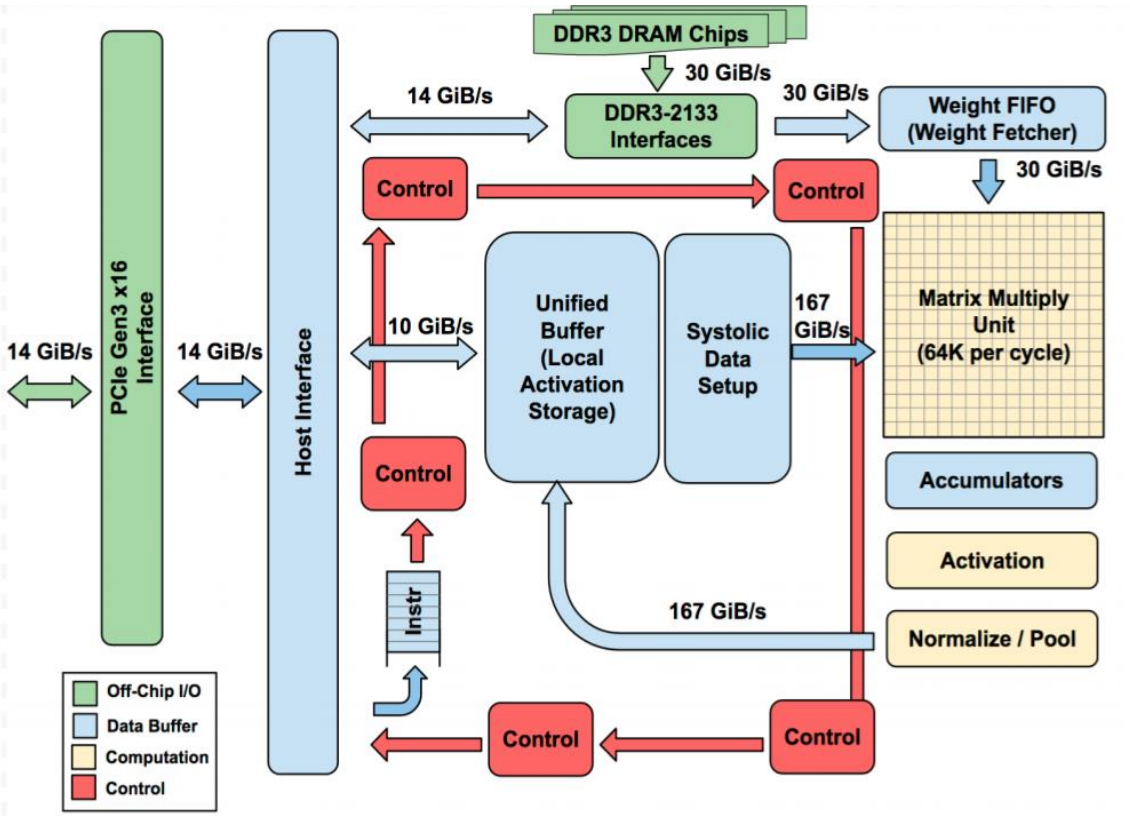
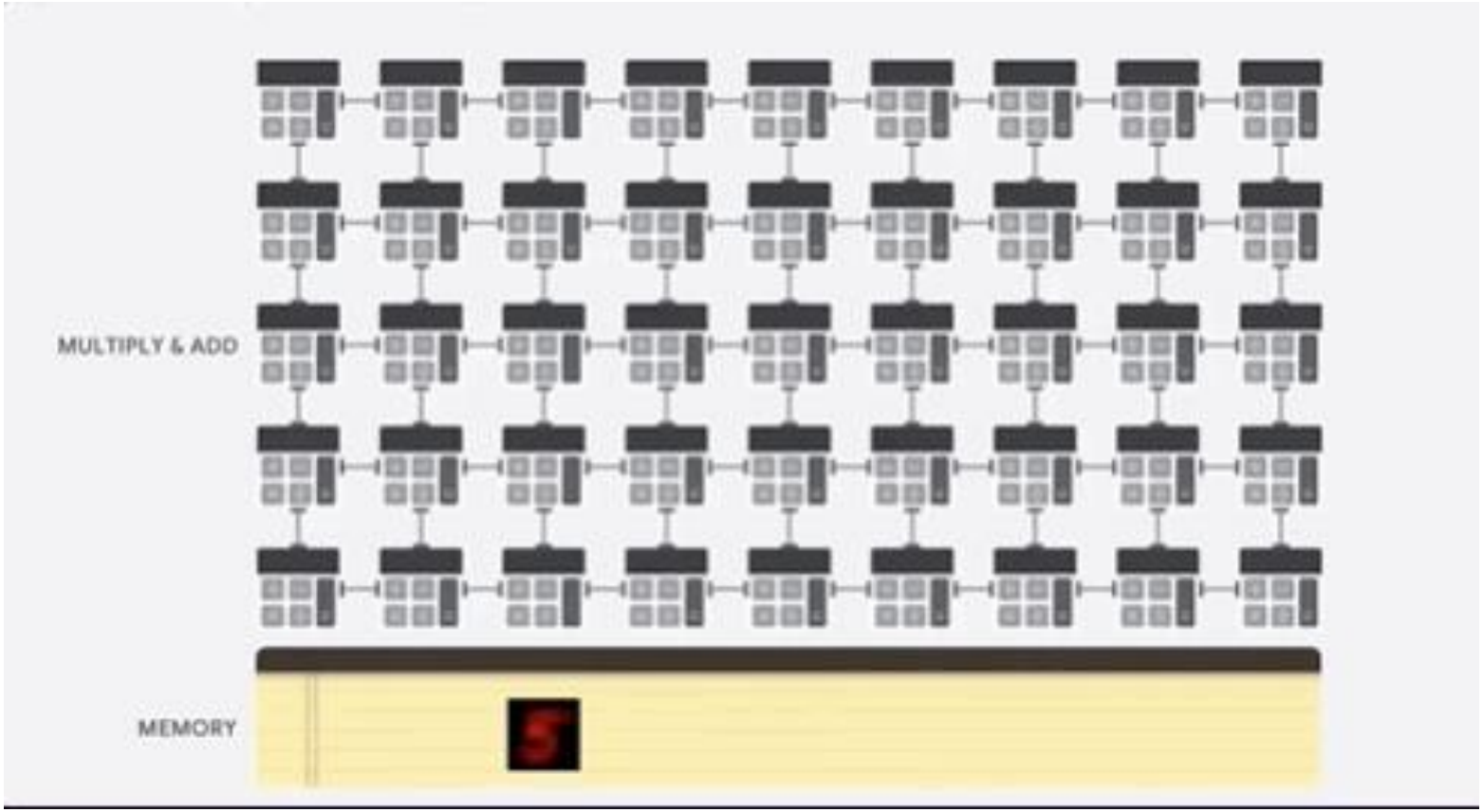


Figure 1. TPU Block Diagram. The main computation is the yellow Matrix Multiply unit. Its inputs are the blue Weight FIFO and the blue Unified Buffer and its output is the blue Accumulators. The yellow Activation Unit performs the nonlinear functions on the Accumulators, which go to the Unified Buffer.

https://cloud.google.com/tpu?hl=ja　より



今回の内容

1. 動的スケジューリングの詳細続き
    1. 例外への対応
    2. ロード・ストアへの対応
2. GPU のアーキテクチャ概要

レポート課題

- MICRO2019/ISCA2020，ないしはこの講義で出てきた何らかの論文を１つ選び読んでまとめる
  - 分量は，日本語なら3000文字，英語なら1500ワード程度を目安
  - ISCA 2020
    - https://www.iscaconf.org/isca2020/program/
  - MICRO 2019
    - https://dl.acm.org/doi/proceedings/10.1145/3352460
- 提出方法：
  - shioya@ci.i.u-tokyo.ac.jp にメールで提出
  - タイトルを「先進計算機構成論レポート (学籍番号)」とすること
- 締め切り：8月10日

出欠と感想

- 本日の講義でよくわかったところ，わからなかったところ，質問，感想などを書いてください（なんか一言書いてね）
  - LMS の出席を設定するので，そこにお願いします
  - パスワード：gpgpu
- 意見や内容へのリクエストもあったら書いてください