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

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（復習）

- 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
```

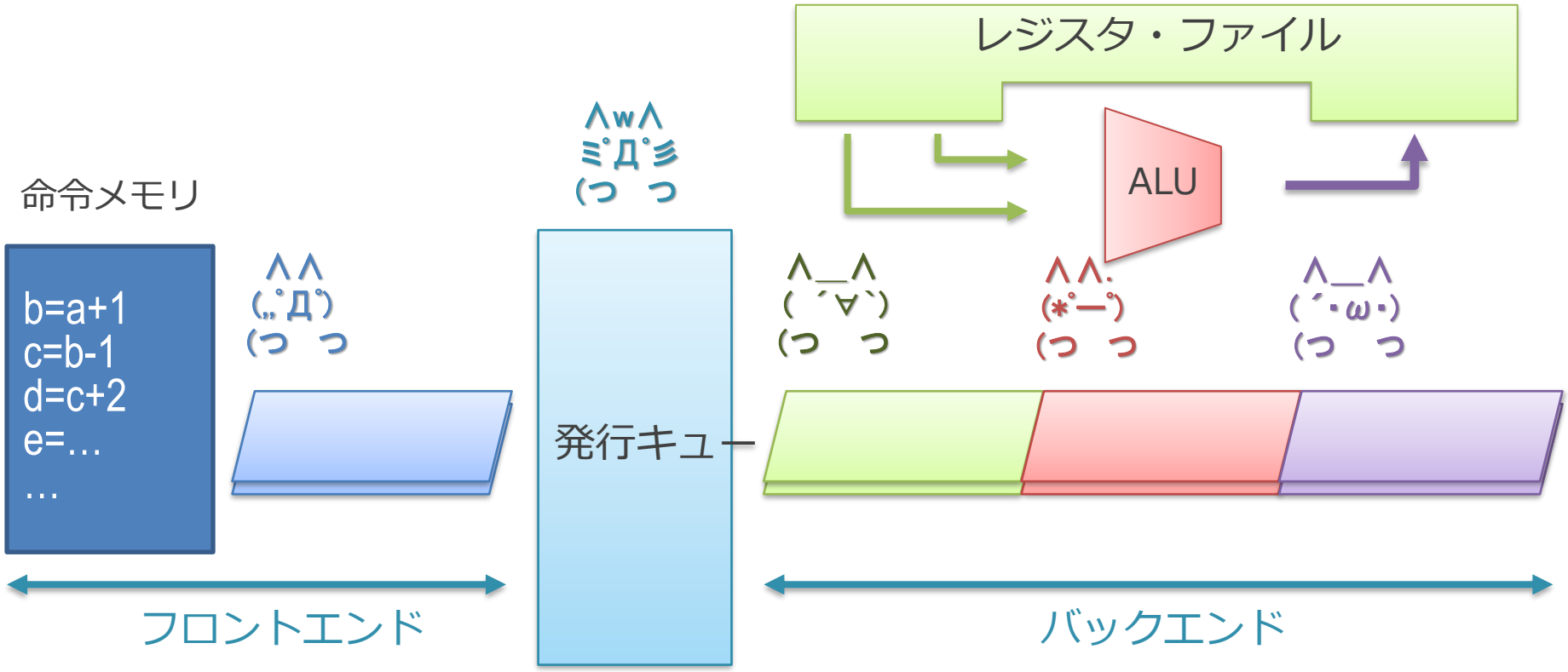
レジスタ・リネーム（復習）

- 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
```

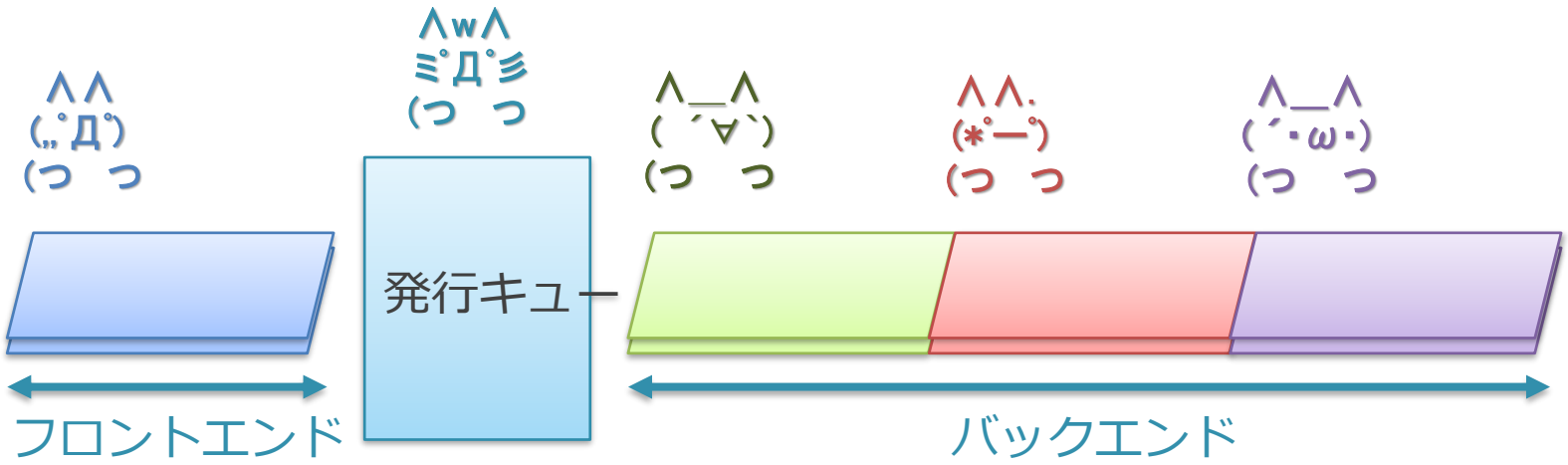
```
I1: mul p20←p12*4
I2: add p21←x11+1
I3: sub p22←p15-1
I4: and p23←p17&1
```

out-of-order 発行を行う CPU の構造（復習）



- 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（復習）



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

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離れたところに飛ぶ
    - （命令サイズが 4 バイト固定の場合）
- 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
```

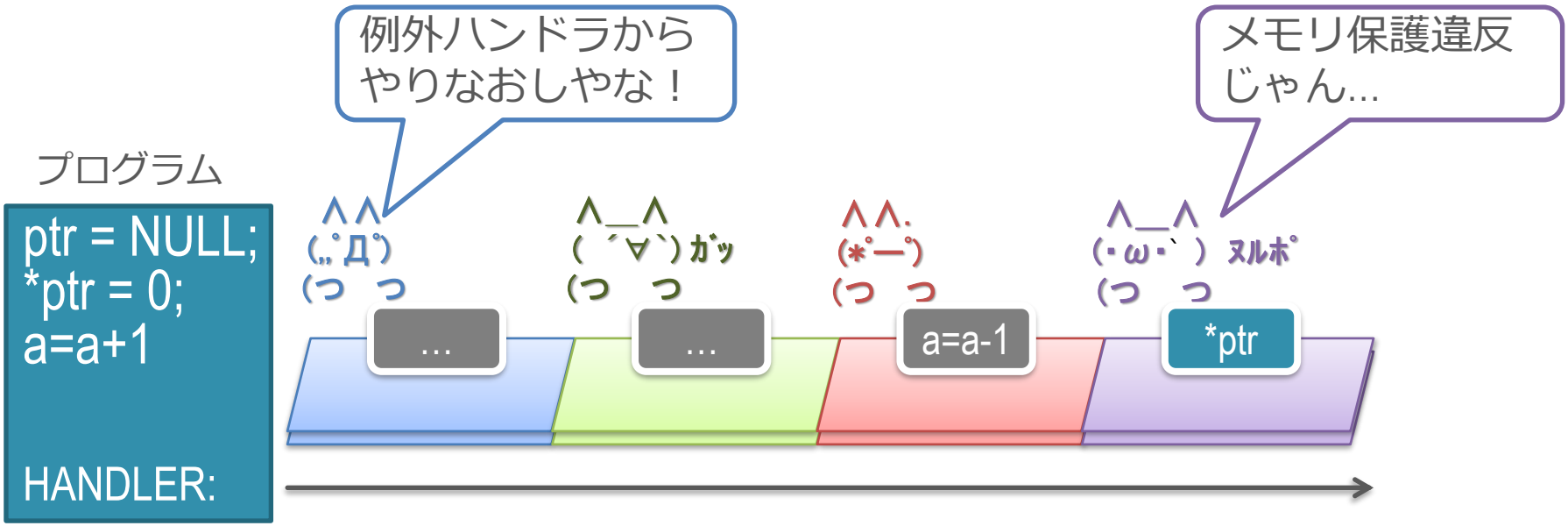
例外の例３

- ブレーク・ポイントの実装
  - ◇ デバッガのブレーク・ポイント機能も例外を使って実装される

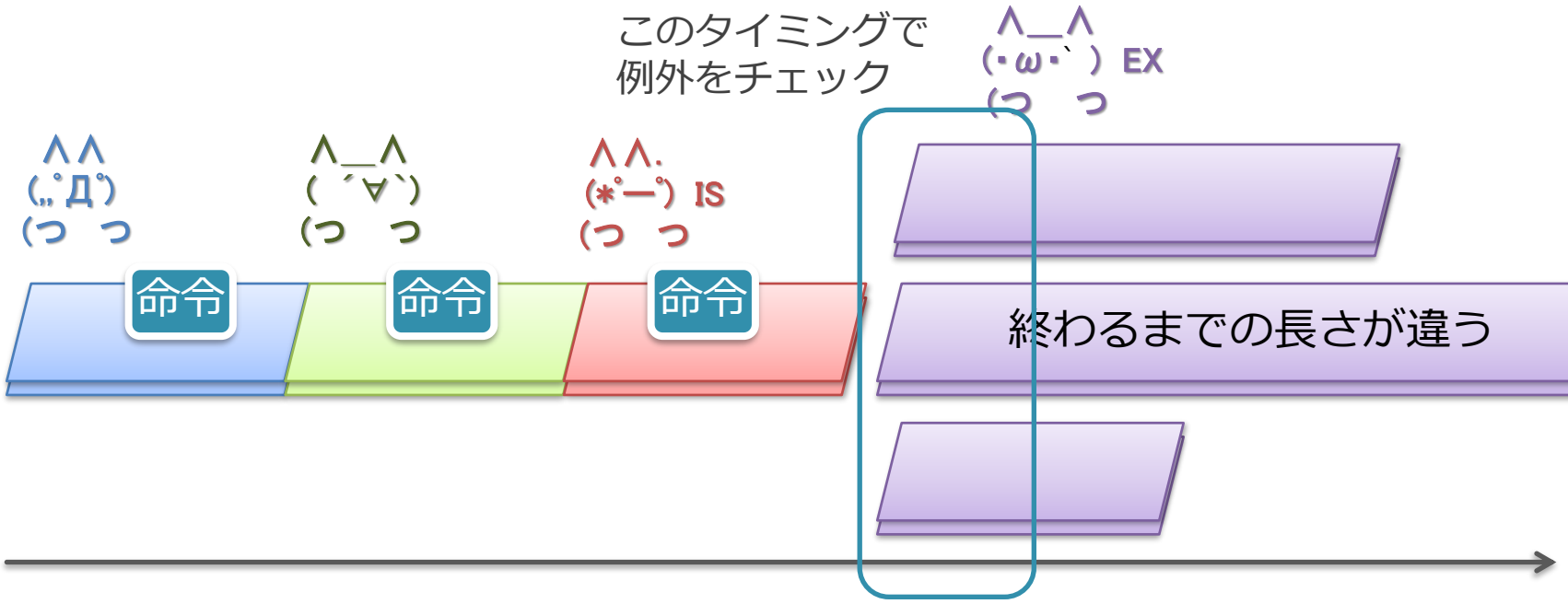
- たとえば，C 言語の加算の文を考える
  1. その文に対応するコンパイル結果の add 命令がどこかにある
     （gcc なら -g をつけると両者の対応がバイナリに埋め込まれる
     `i = i + 1;  →  add x1, x1, 1`
  2. この add 命令を一時的に例外を発生させる命令に書き換える
     `add x1, x1, 1 → ebreak`
  3. ebreak 命令が実行されると例外ハンドラに飛ぶ
     - □ デバッガにブレーク・ポイントに到達したことを通知
     - □ 命令実行毎にブレーク・ポイントを見張る必要がなく高速

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



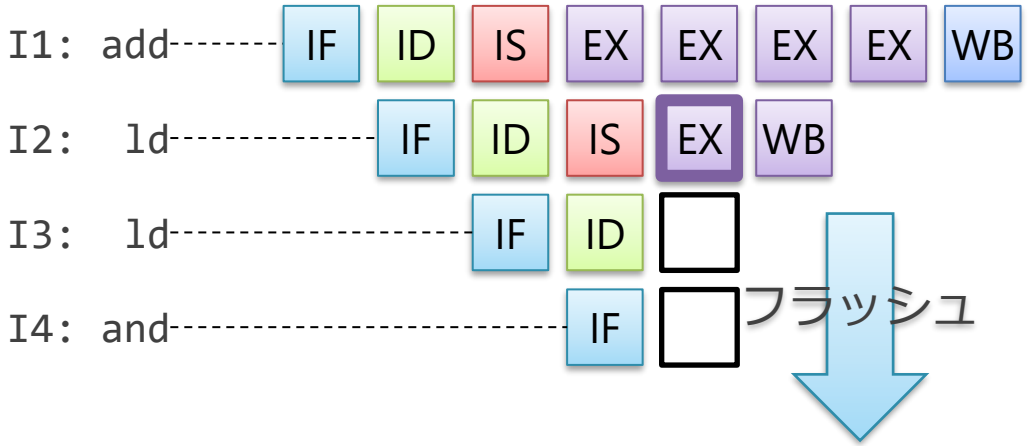
in-order 発行/out-of-order 完了の場合



- (*ﾟ―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

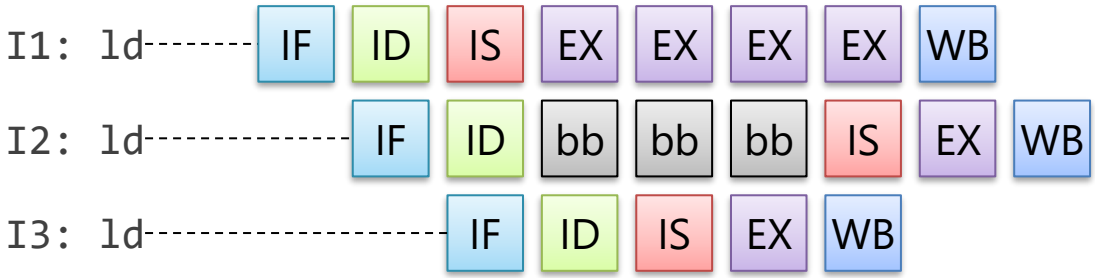
in-order 発行/out-of-order 完了の場合

- 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

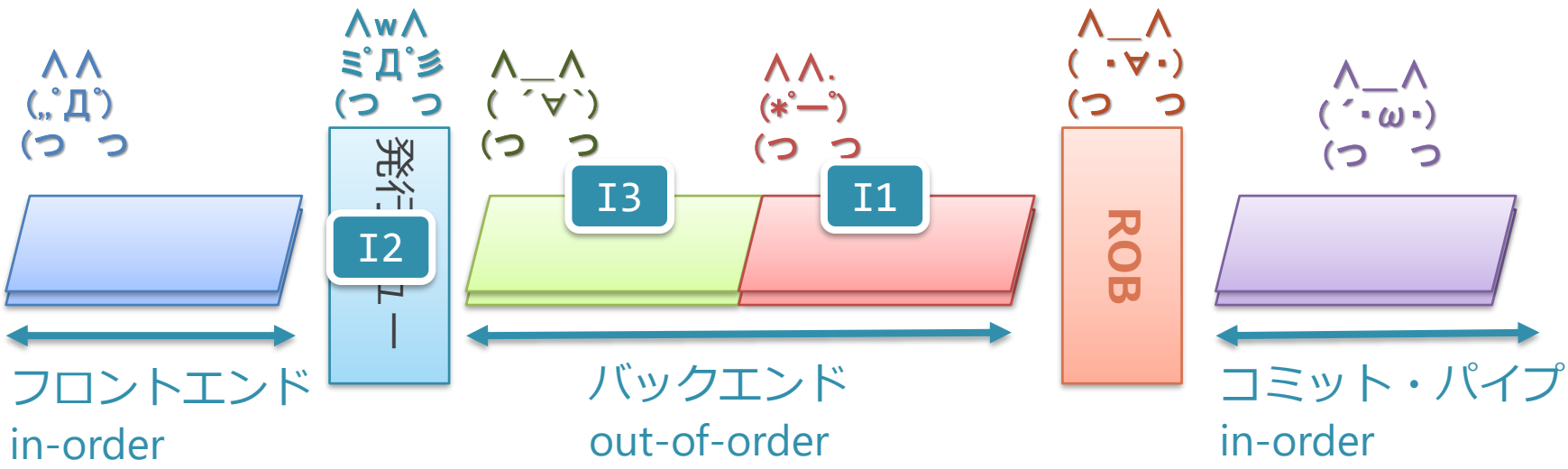


out-of-order 発行/out-of-order 完了の場合

- 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）



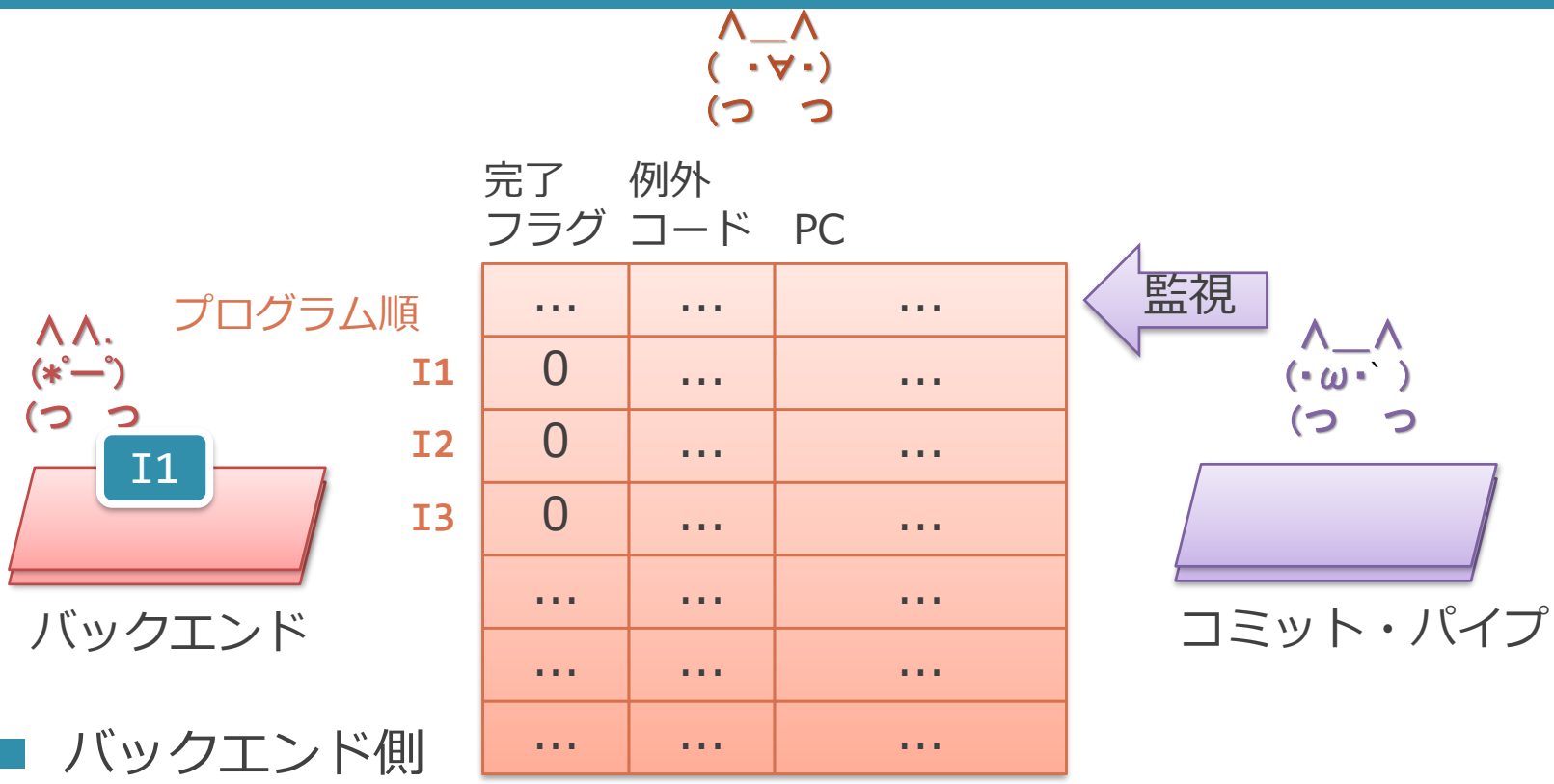
- バックエンドの後ろにリオーダ・バッファ（ROB）を追加
  - ◇ バックエンドで完了した命令は ROB に完了した印をつけていく
  - ◇ コミット・パイプで in-order に印を読み出し，例外を反映
  - ◇ （上記のバックエンド/コミット・パイプをそれぞれ実行コア/バックエンド と呼ぶ文献もある）

- コミット（commit）：実行を確定させる操作

ROB の中身

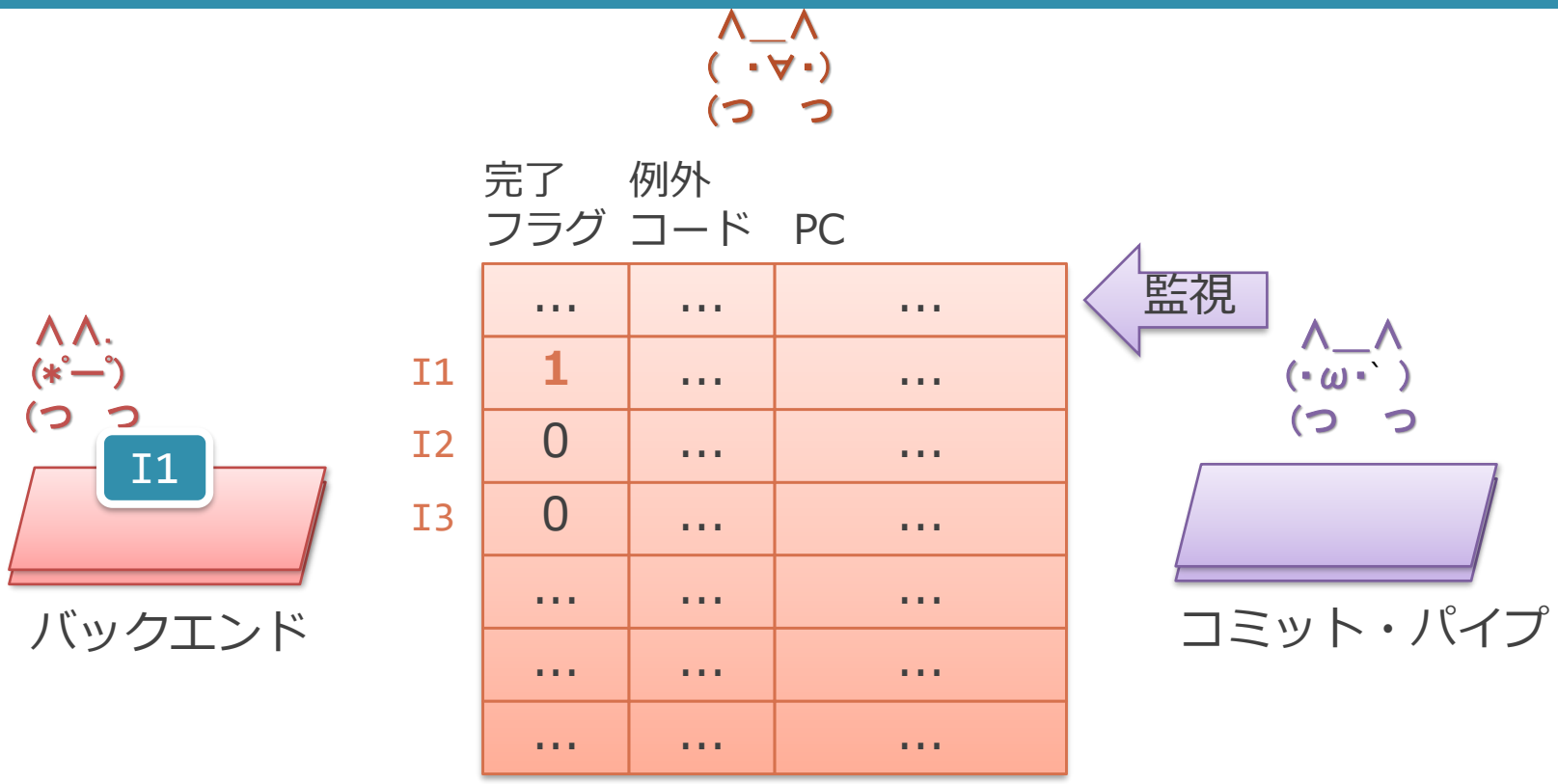
- ROB：
  - ◇ プログラム順にエントリが確保される FIFO
- 内部にあるフィールド
  1. 完了フラグ
     - □ 完了した命令は 1 を書き込む
  2. 例外コード
     - □ 例外を発生させた場合は，その種類を書く
  3. その命令のPC or 分岐先ターゲット
     - □ どっちを入れるかは例外の種類次第

ROB の動作（１）



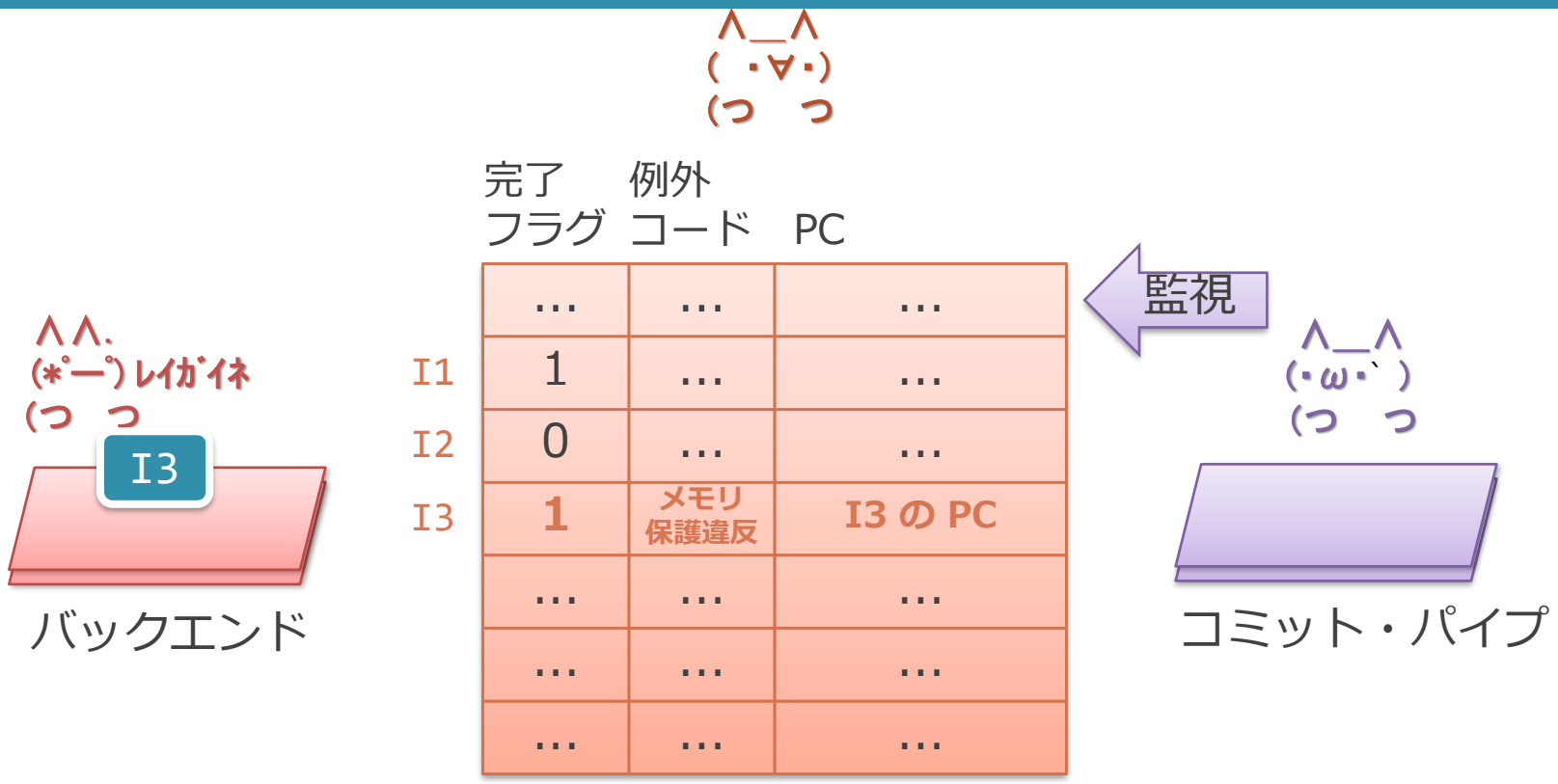
- バックエンド側
  - ◇ 各命令はプログラム順に ROB のエントリを確保
  - ◇ バックエンドで完了した命令は out-of-order に ROB に書き込む
- コミット・パイプ側
  - ◇ in-order に「ここまで完了した」部分を監視

ROB の動作（２）



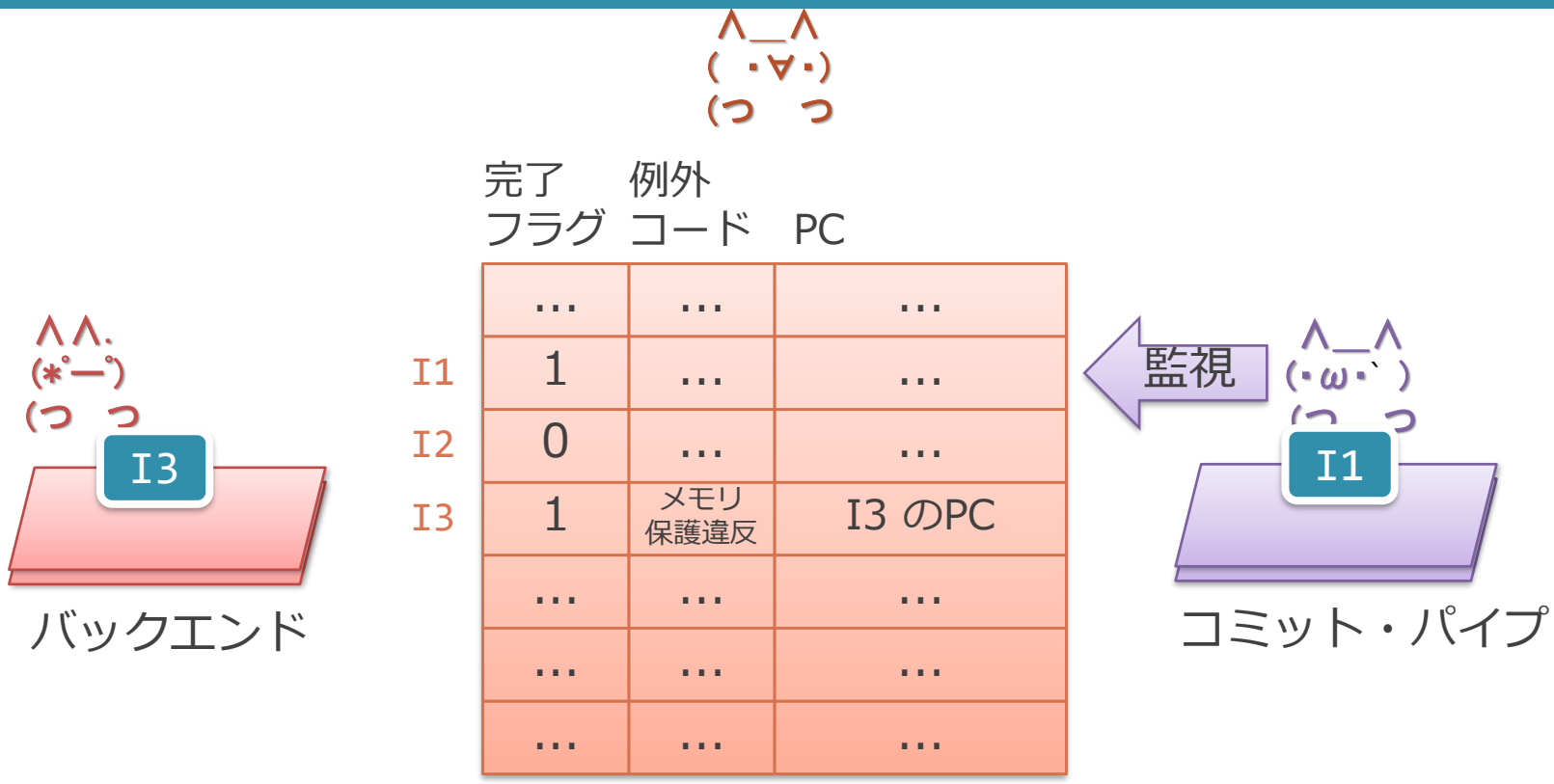
- I1 が完了したので，ROB に書き込みを行う
  - ◇ 正常完了したので，完了フラグに１を書く

ROB の動作（３）



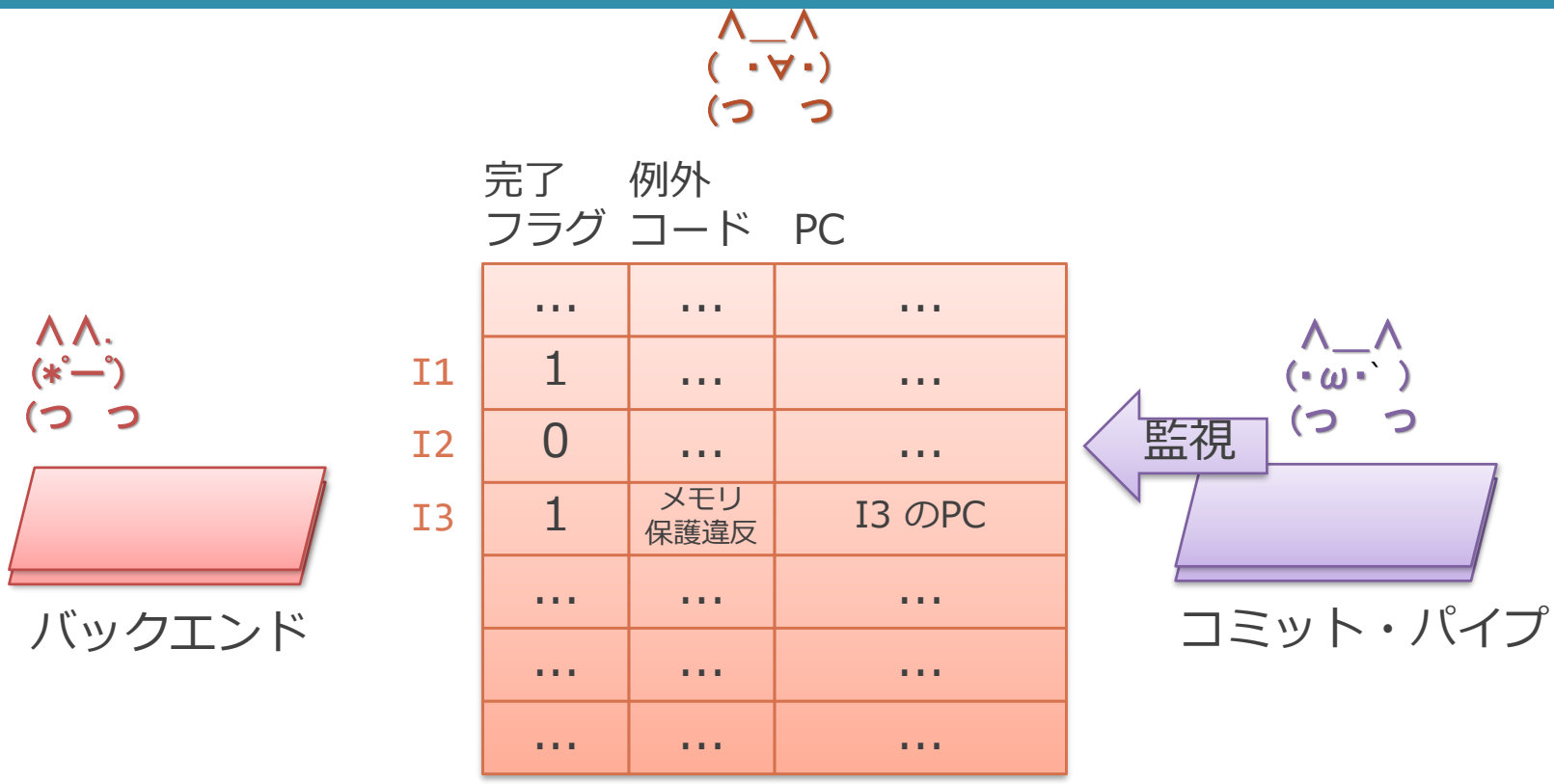
- I3 が完了したので，ROB に書き込みを行う
  - ◇ 完了フラグに１を書く
  - ◇ 例外を発生させていたので，その種類と自分の PC も書く

ROB の動作（４）



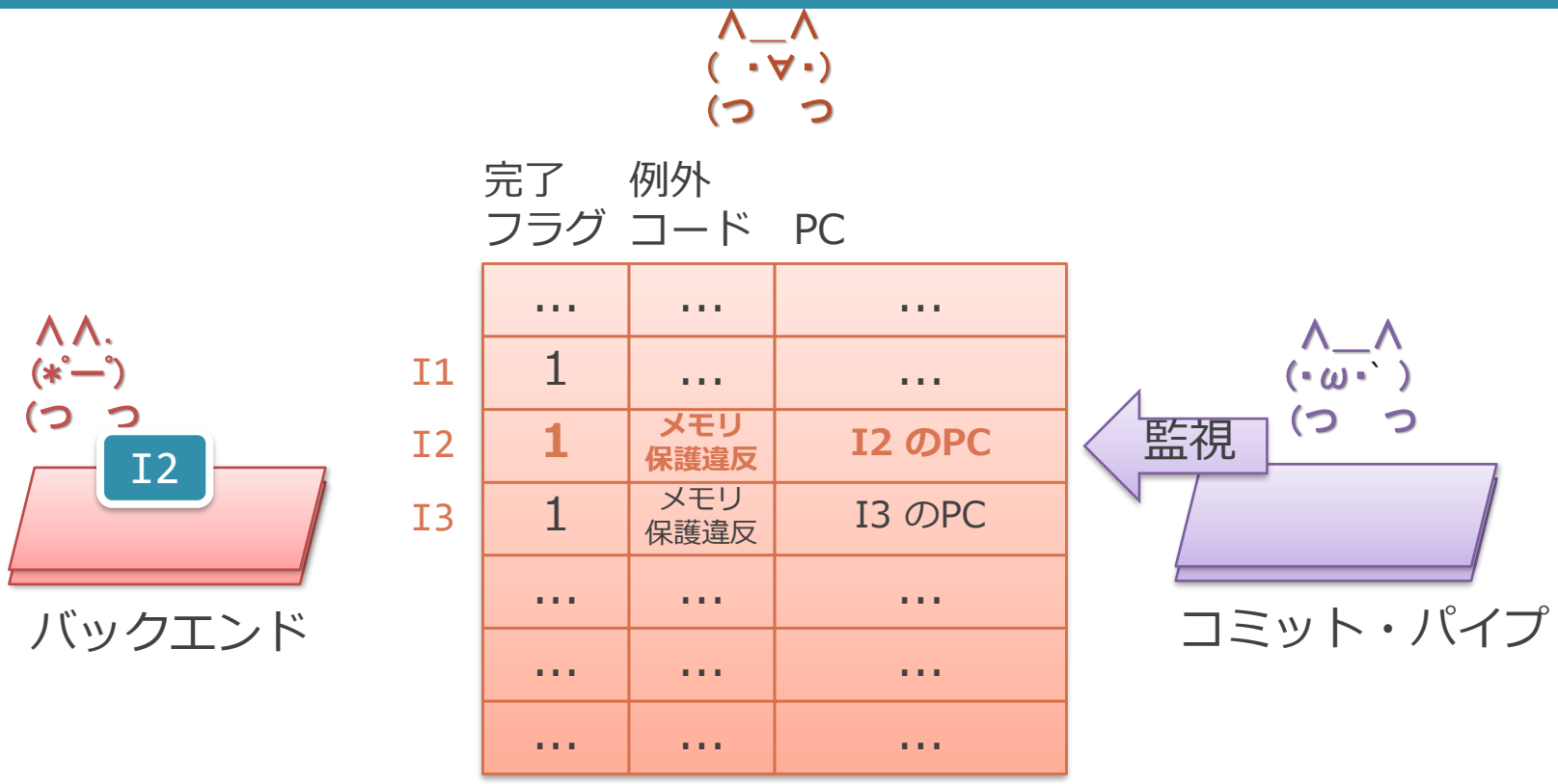
- コミット・パイプの監視ポイントが I1 に移動してきた
  - ◇ 直前の命令のコミットが終わった
  - ◇ I1 の完了フラグが 1 であるため，I1 をコミット
  - ◇ コミット・パイプはバックエンドとは独立して並列に動作

ROB の動作（５）



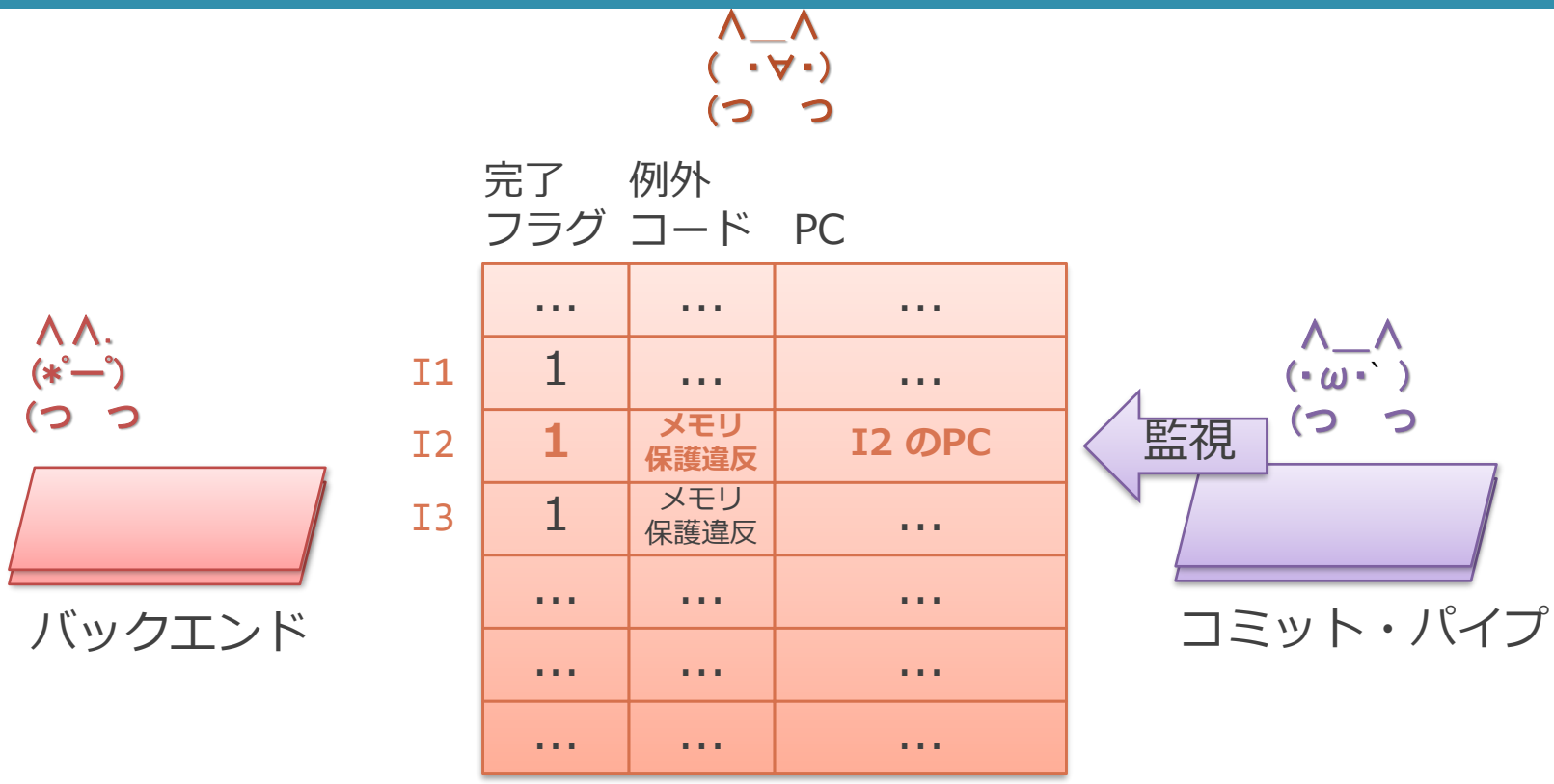
- コミット・パイプの監視ポイントが `I2` に移動
  - ◇ `I2` の完了フラグはまだ `0` なのでコミットはしない

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）

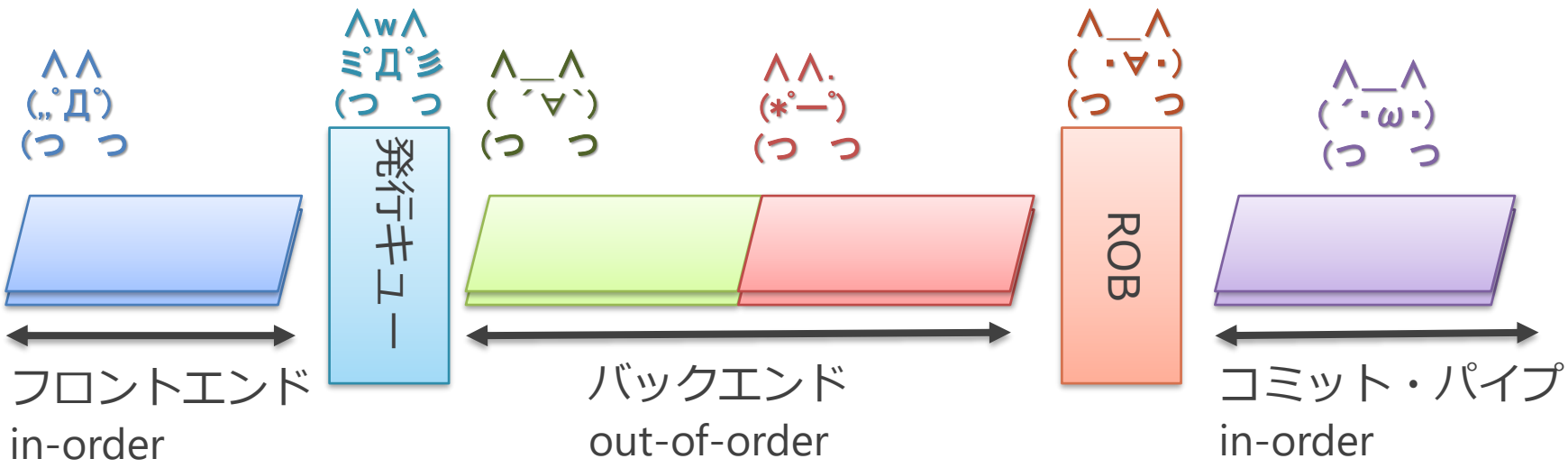


- 監視対象の I2 のエントリが完了している
  - ◇ 例外コードが記録されているので例外の処理を行う
    - □ PC を例外ハンドラのアドレスに書き換え
    - □ 後で戻ってこれるように I2 の PC を特別なレジスタに保存
  - ◇ I3 以降を全部取り消し

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
- コミット時までまってから回復だと遅いため，
  命令の完了時に 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**

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

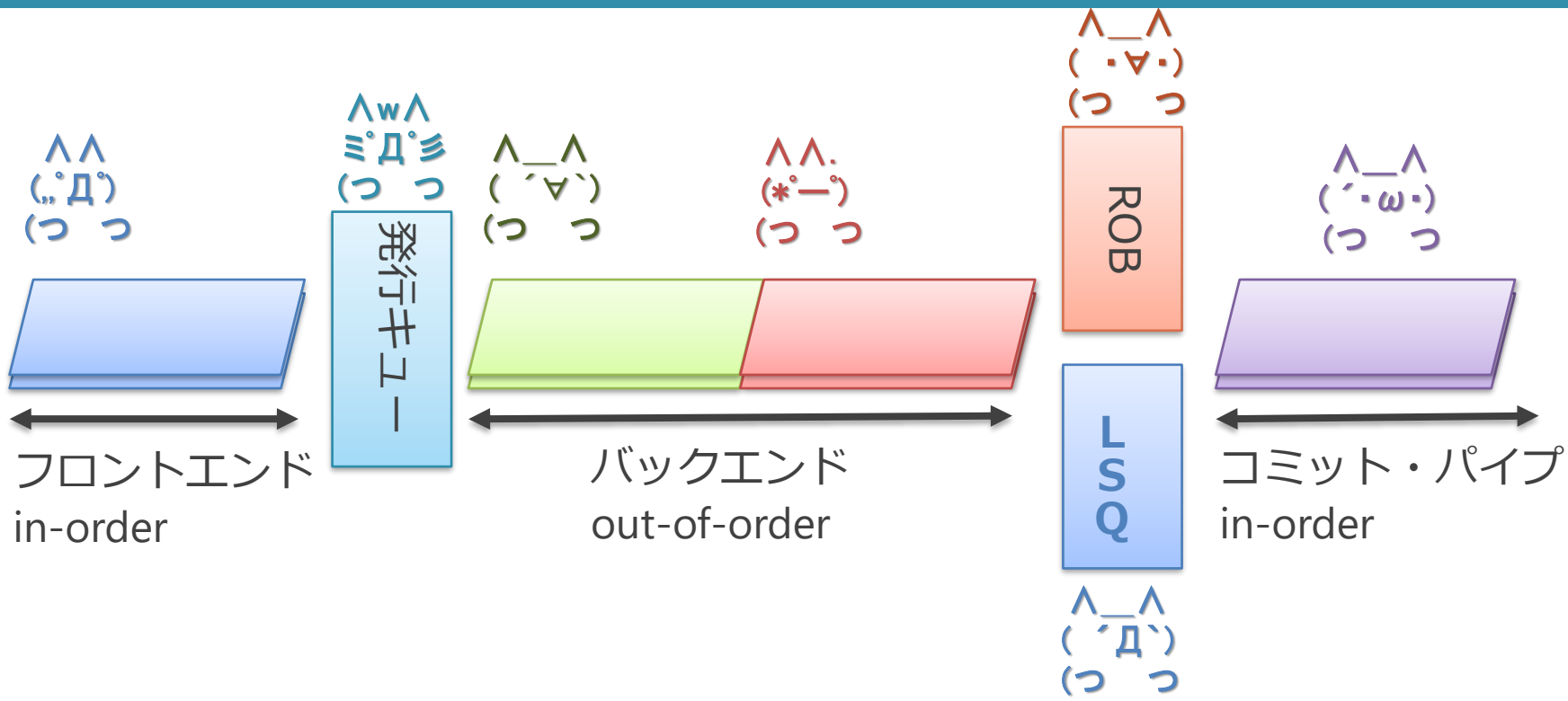
```
I1: lw x2←(0x1000)
I2: sw x1→(0x1000)
```

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

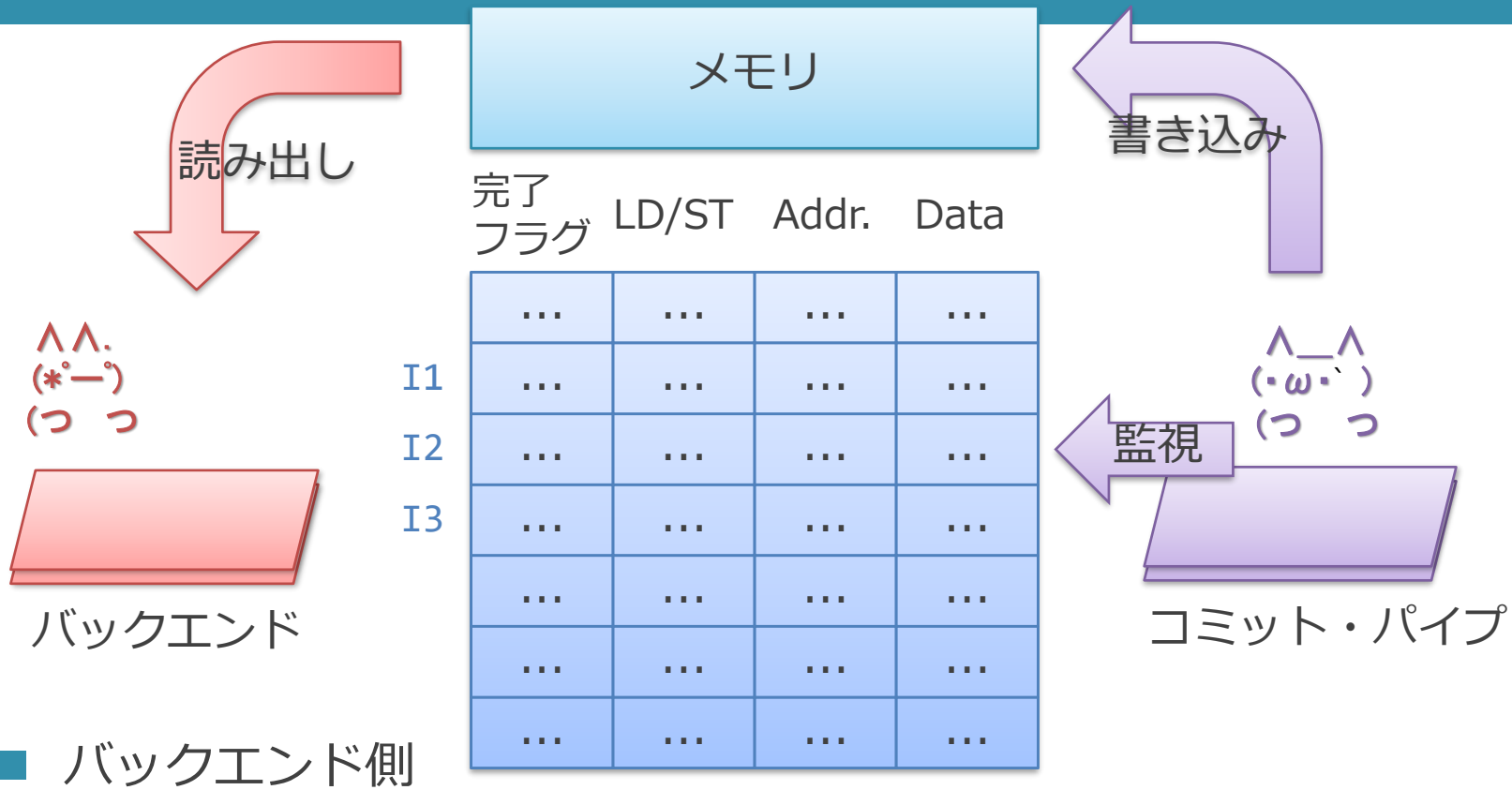
LSQ の役割

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

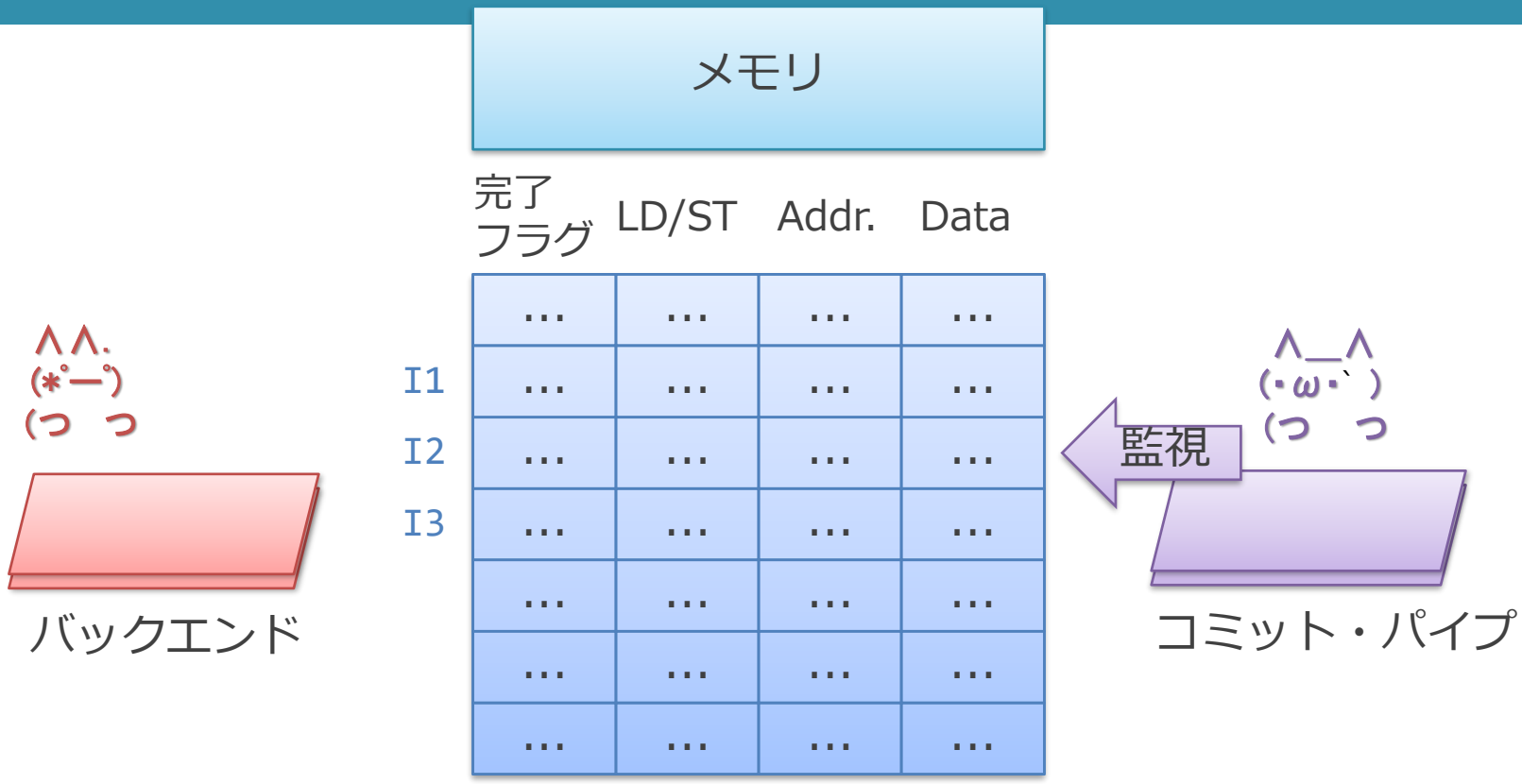
- LSQ：
  - ◇ プログラム順にエントリが確保される FIFO
  - ◇ ROB と違い，ロードとストアにのみ割り当てられる
    - □ ロードとストアのために ROB を拡張したものとも言える
- 内部にあるフィールド：
  1. 完了フラグ：
     - □ 完了した命令は 1 を書き込む
  2. ロード or ストア識別のフラグ
  3. アドレス
  4. データ

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
  - ◇ 以下の動作を順に説明
    1. ストアの整列
    2. 順序違反の検出

ストアの整列（１）

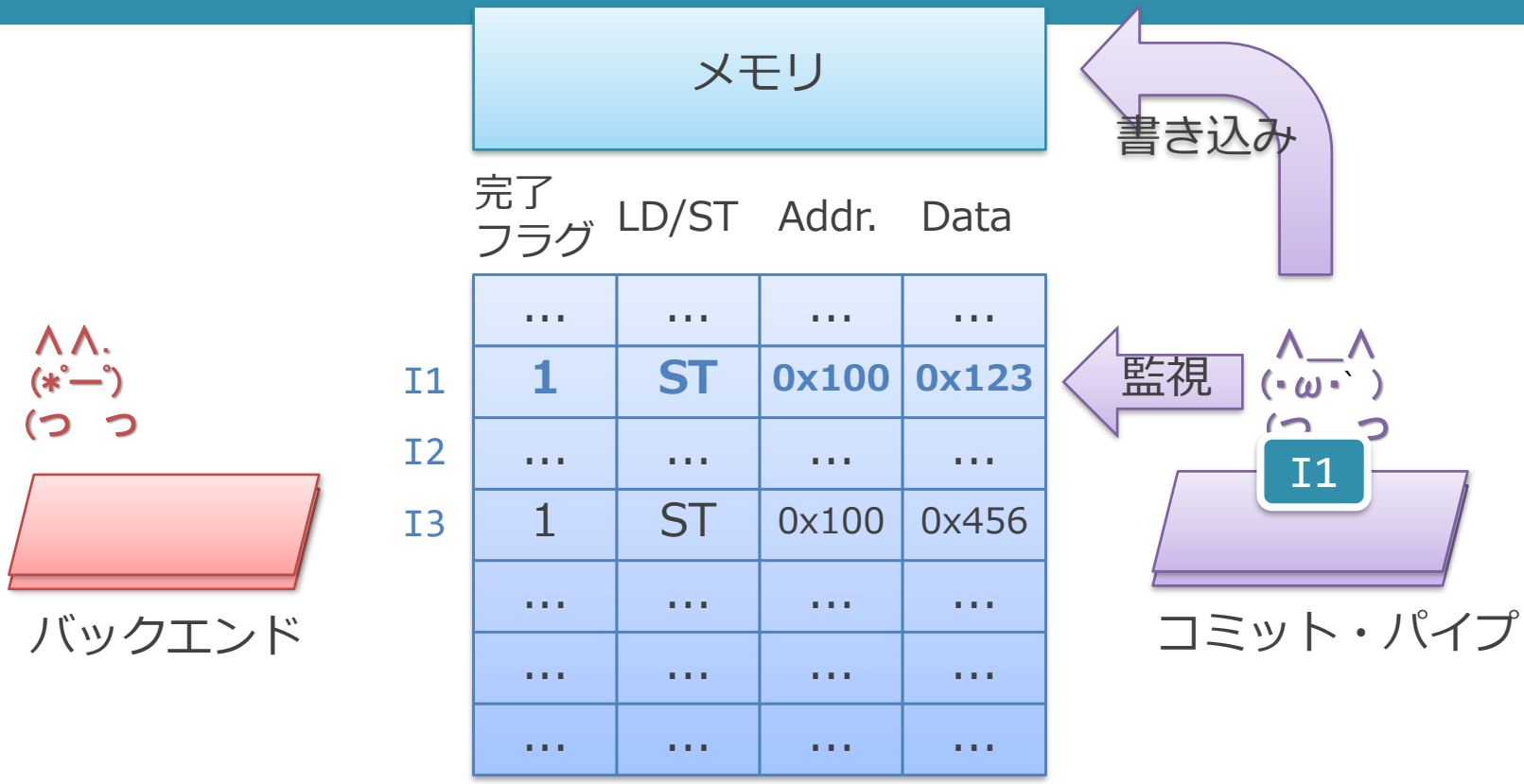
- ストア実行時
  - ◇ バックエンドから LSQ にアドレスとデータを書き込む
  - ◇ バックエンドからはメモリには書き込まない

- `I1:SW x1→(0x100)` が完了

ストアの整列（２）

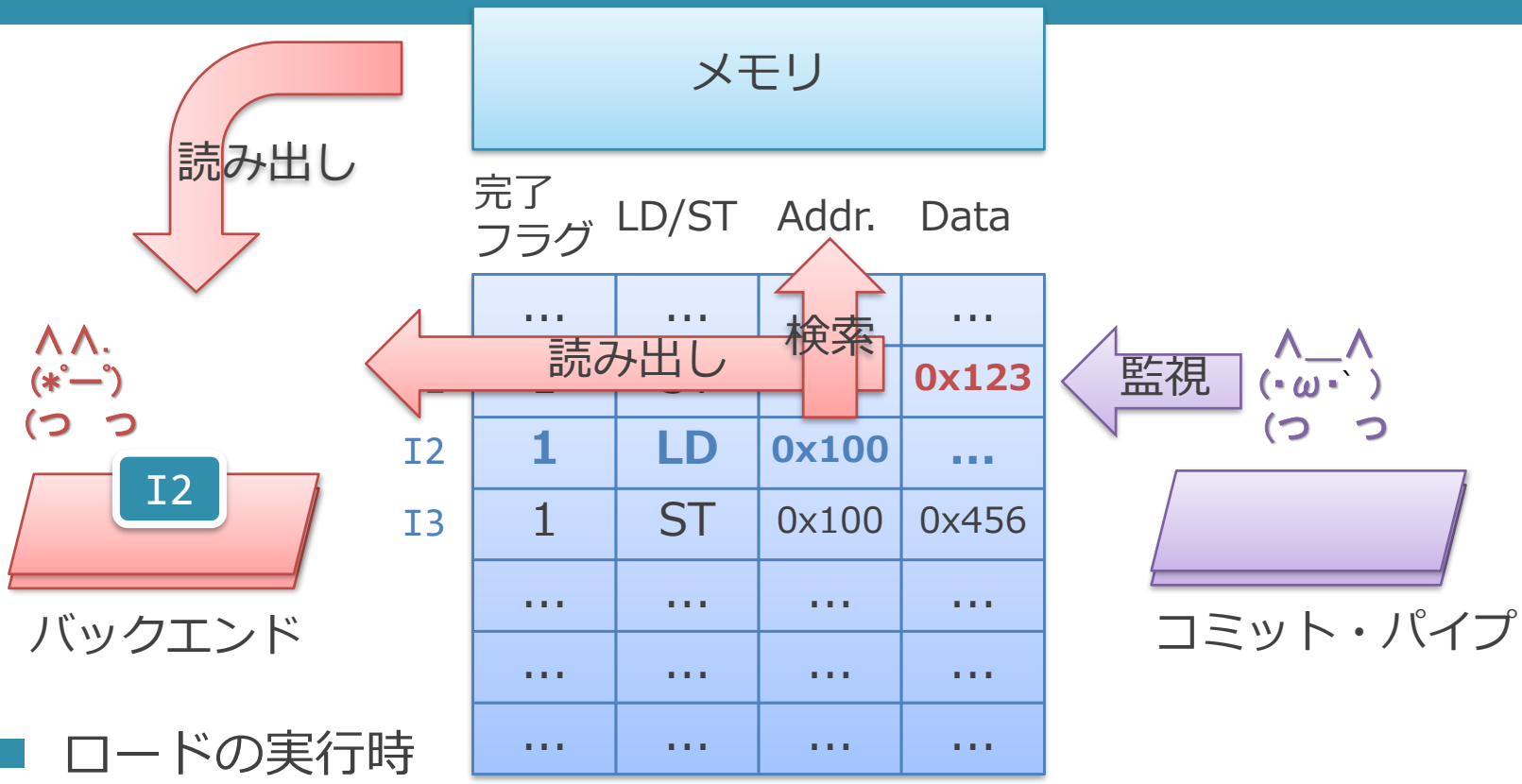
- I3:SW x2→(0x100) が完了
  - ◇ 対応するエントリに同様に書き込む

ストアの整列（3）



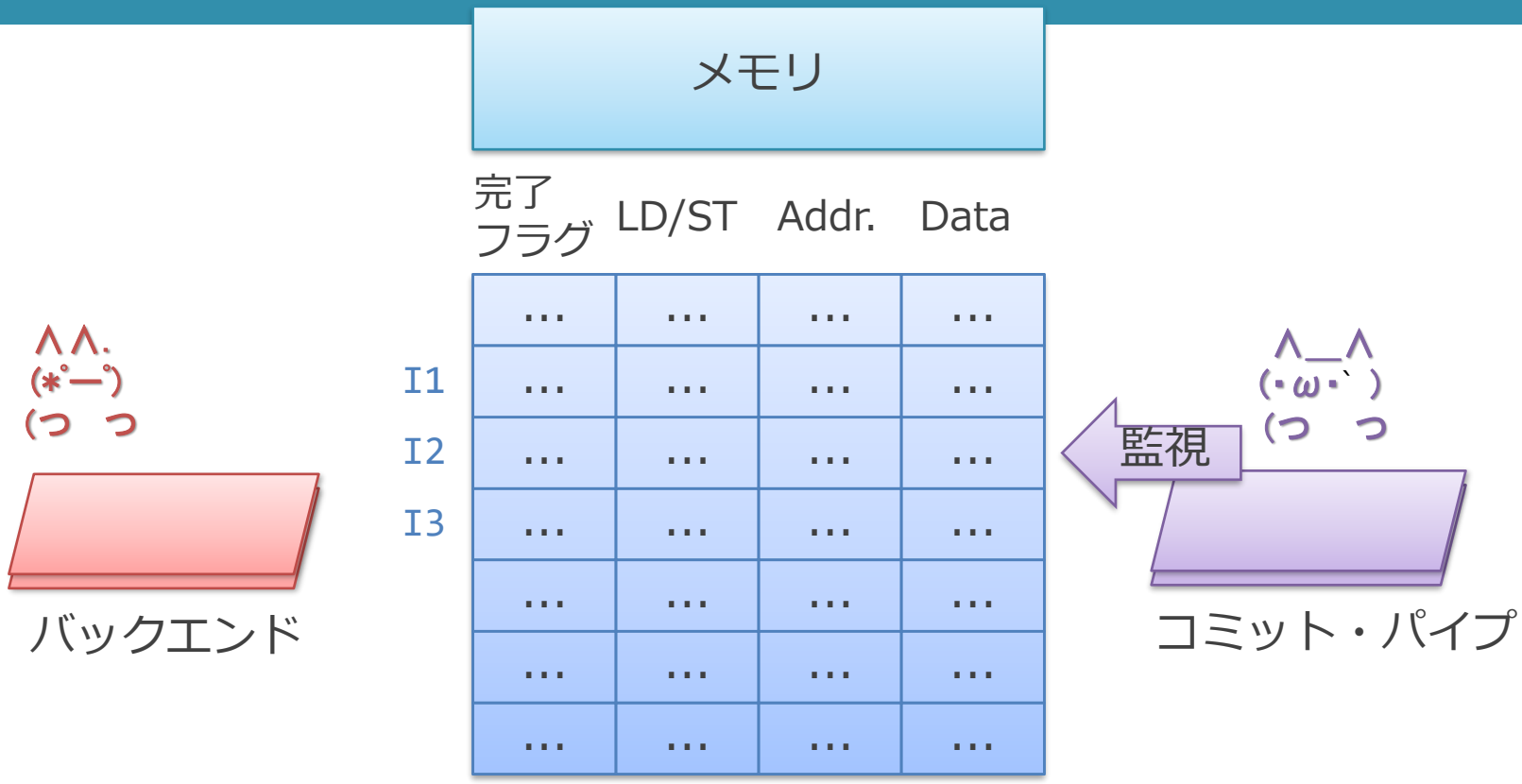
- 監視対象が I1 に移り，完了していることを検出
  - ◇ コミット・パイプからメモリにデータを書き込む
- 監視ポイントがプログラム順に下に移動していく = ストアの整列
  - ◇ 元のプログラム順でメモリに対してデータが書き込まれる

ストアの整列（４）



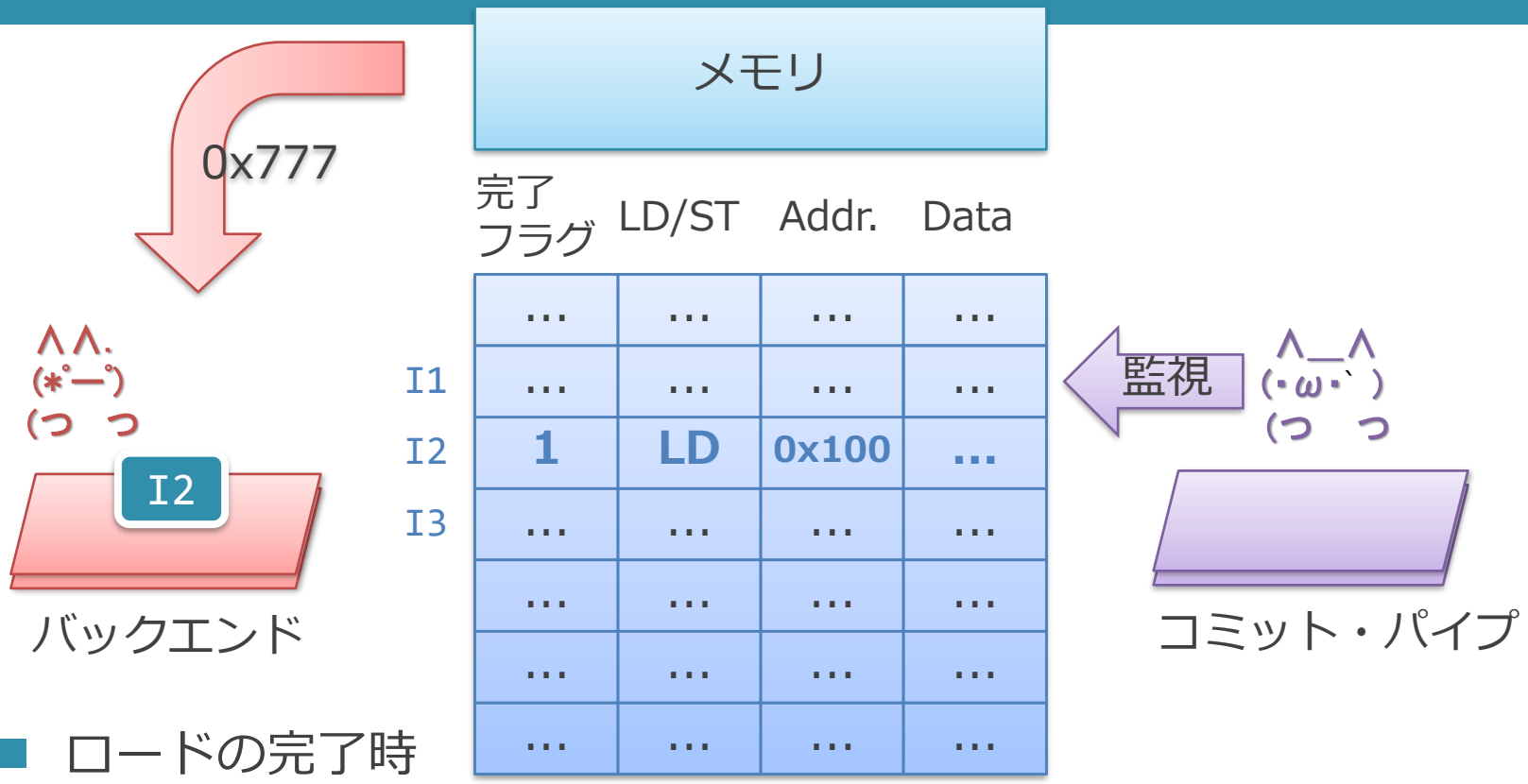
- ロードの実行時
  - ◇ LSQ とメモリの双方にアクセス
- LSQ の自身のエントリより前の部分をアドレスで検索
  - ◇ ヒットした場合，そこにあるストアの値を読み出して使う
  - ◇ ヒットしなかった場合，メモリからとれた値を使う

動作例を使った説明



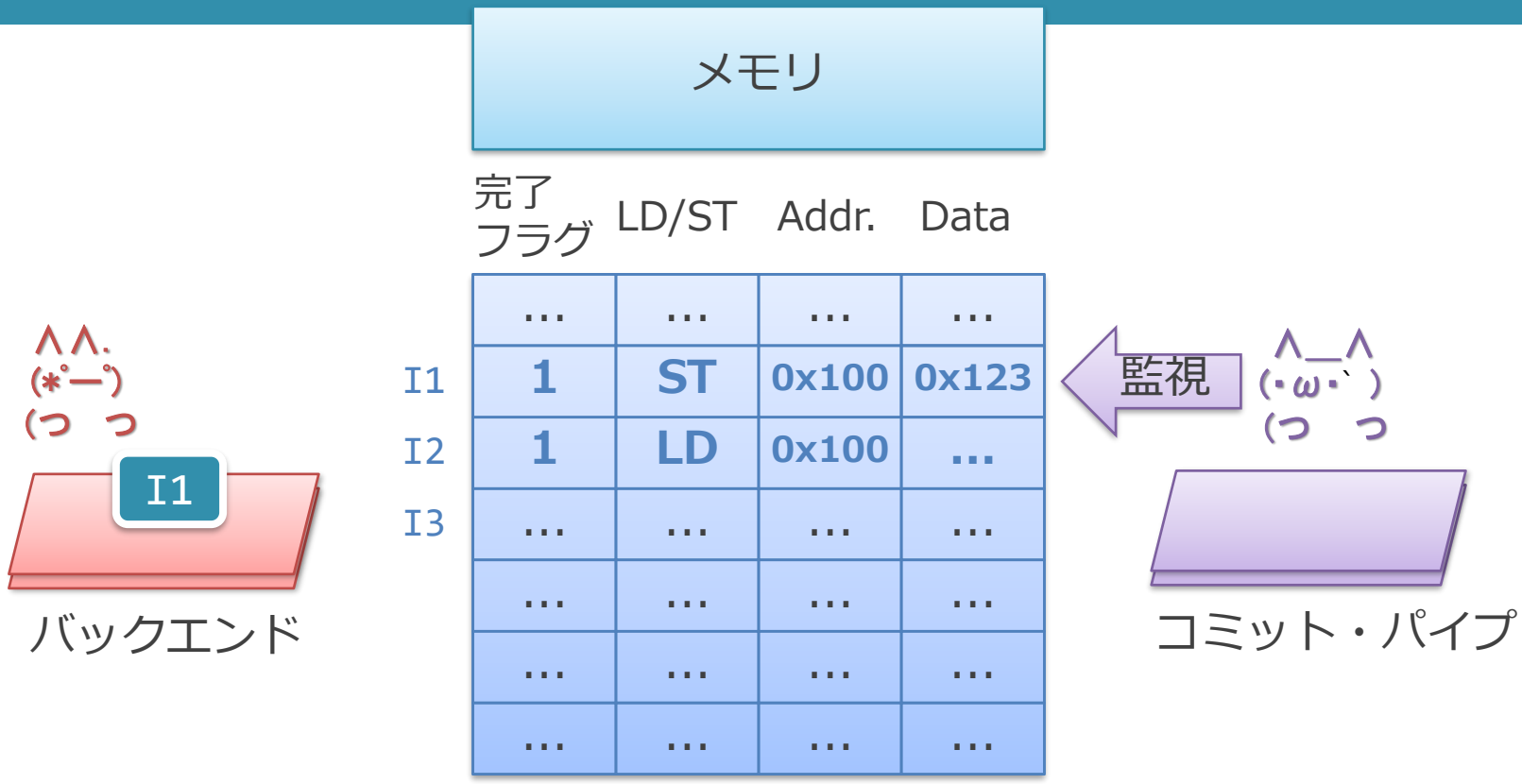
- `I1,I2,I3` は同じアドレス `0x100` に対してアクセスを行う命令
  1. ストアの整列
  2. **順序違反の検出**

順序違反の検出（１）



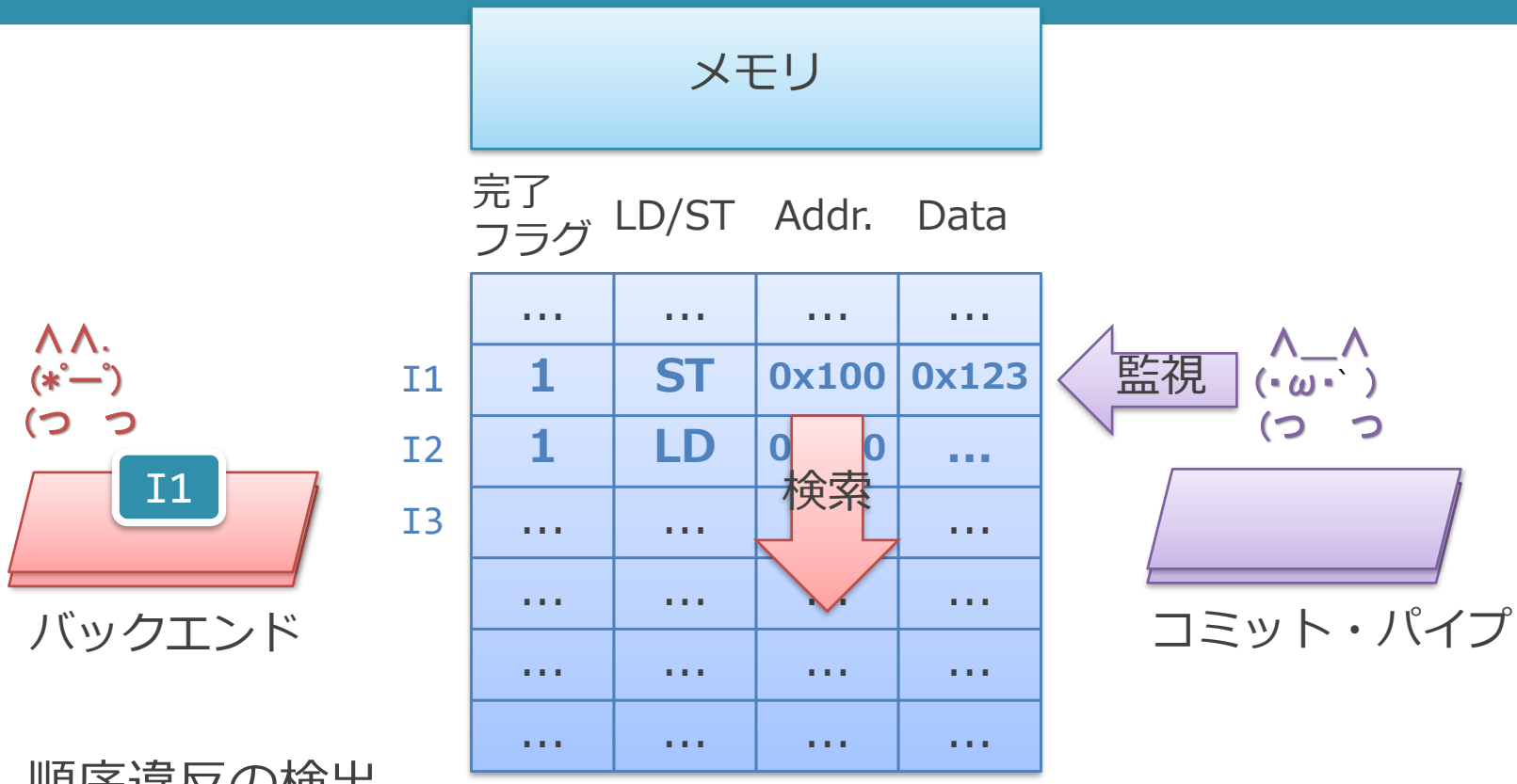
- ロードの完了時
  - ◇ LSQ に完了フラグやアドレスを書き込む
- 上では I2 のロードが I1 よりも先に実行
  - ◇ LSQ 内には検索しても 0x100 に該当するものはない
  - ◇ メモリからは 0x777 がとれたので，これを使用する

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，この講義で説明した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

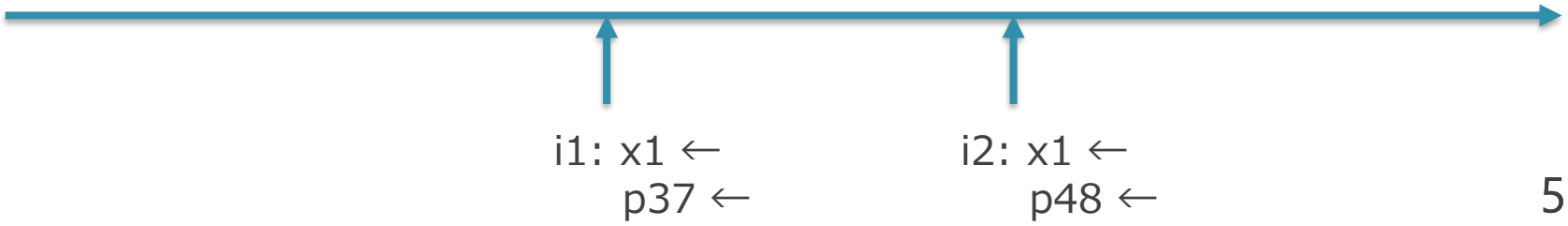
動的命令スケジューリングについての補足

- 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



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
  - 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:
- 意見や内容へのリクエストもあったら書いてください
- LMS の出席の締め切りは来週の講義開始までに設定してあります
  - 仕様上「遅刻」表示になりますが，特に減点等しません
  - 来週の講義開始までは感想や質問などを受け付けます

質問や感想への回答

質問と感想

- 物理レジスタが論理レジスタの数倍の数あるということでしたが、それくらいの数でほとんどの偽の依存は解決出来るんでしょうか？それともももっと載せたいけど物理的な制約でそれくらいしか載せられないということなんでしょうか？

質問と感想

- レジスタリネームでは論理レジスタの数倍程度の数を用意することが分かっているだけど、x86_64だと汎用レジスタが16本で、RISC-Vだと32本で、これらを数倍すると、かなり回路の規模が増大するイメージがします。実際どれぐらいの回路規模が増えるでしょう？
- また、SIMD用のレジスタにもリネームリされていますか？
- 物理レジスタは論理レジスタの数倍の数とありますが，最近のCPUだと何倍くらいでしょうか？

「AMD Next Generation “Zen 4” Core and 4th Gen AMD EPYC™ Server CPUs」より

Table 1 Comparison of “Zen 3” and “Zen 4” Microarchitectural Metrics

	“Zen 3”	“Zen 4”
L1 BTB[2]	2 x 1 K	**2 x 1.5 K**
L2 BTB	2 x 6.5 K	**2 x 7 K**
L1 I-Cache	32 KB	32 KB
Op Cache	4 K ops	**6.75 K ops**
Issue width (integer + FP/SIMD)	10 + 6	10 + 6
Integer physical registers	192	**224**
Integer scheduler	4 x 24	4 x 24
FP/SIMD physical registers	160	**192**
FP/SIMD scheduler (non-pickable)	64	64
FP/SIMD scheduler (pickable)	2 x 32	2 x 32
Retire Queue	256	**320**
Load Queue	72	**88**
Store Queue	64	64
L1 D-Cache	32 KB	32 KB
L1 D-TLB	64	**72**
L2 D-TLB	2 K	**3 K**
L2 Cache	512 KB	**1 MB**
L2 Latency	12 cycles	**14 cycles**
L3 Cache per core	4 MB	4 MB
L3 Latency	46 cycles	**50 cycles**

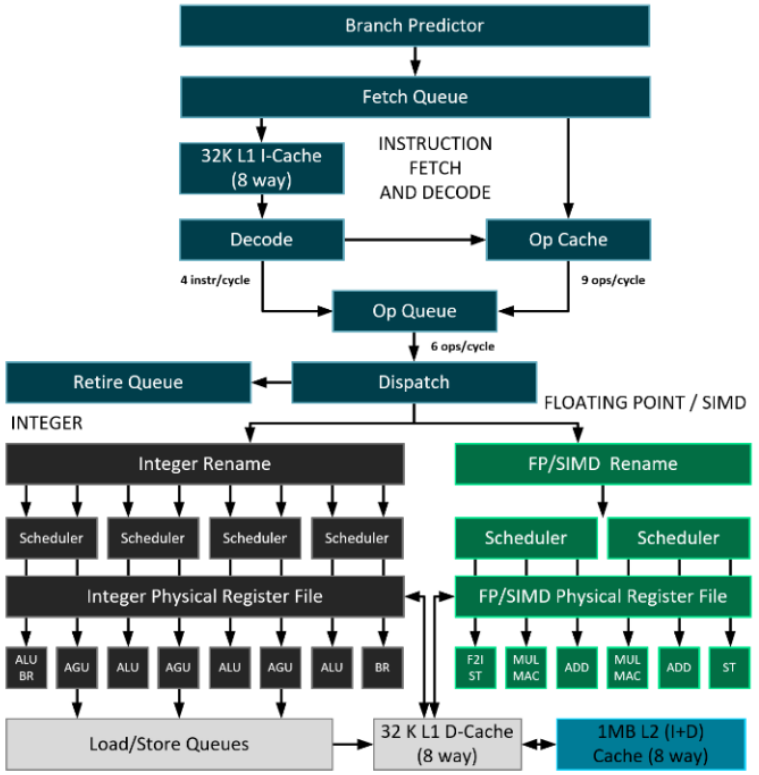
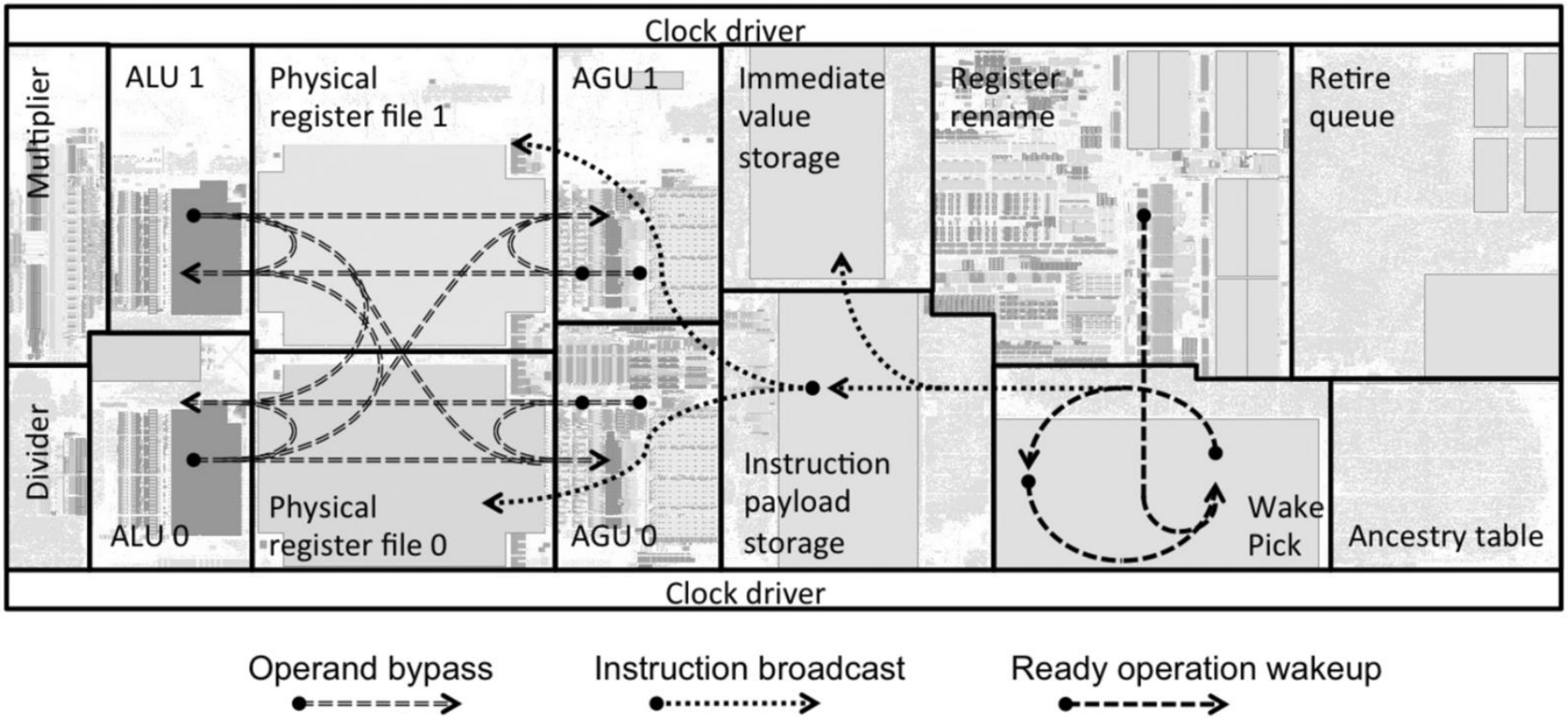


Fig. 1 “Zen 4” Microarchitecture Block Diagram.

DESIGN OF THE TWO-CORE x86-64 AMD "BULLDOZER" MODULE IN 32 nm SOI CMOS より



質問と感想

- 分岐予測と組み合わせてoutoforder実行を利用した場合に、分岐予測を間違えた際にレジスタに意図しない値が書き込まれてしまうリスクはないのか疑問に思いました

質問と感想

- 初めてOoO実行を習った時はトマスロ方式だったのでより構造が単純で性能も出るアルゴリズムがあるとは知らなかった。

質問と感想

- 物理レジスタ/論理レジスタという命名について，少ないリソースが物理で，限りあるリソースを上手く沢山見えるようにする仕組みが論理，みたいなイメージがありやや混乱しました

質問と感想

- 連想検索issueした直後にブロードキャストするとのことですがこれはレジスタへの書き込み直後という認識であっていますか？
- また、srcが複数の場合には命令をsrcが一つになるように分解する等の処理が必要なのでしょうか？

質問と感想

- マトリクススケジューラも素人目には単純そうに見えますが，これより良いスケジューラってないのですか？

質問と感想

- トマスロ方式って具体的にどのようなシステムで採用されているのでしょうか？

質問と感想

- なんで教科書はトマスロ方式の記載が多いのでしょうか？（教育的な意義が強いからとかですか）

質問と感想

- トマスロ方式のIBM System/360が1964年発表、物理レジスタ方式のMIPS R10000が1996年発表と、30年以上間が空いていますがその間にも別の方式が採用されたりしたのでしょうか？

質問と感想

- 初歩的な質問で恐縮ですが、前回と今回扱ったスケジューラは回路的に実装されているのでしょうか、それともCPUを間借りして走っているのでしょうか？話し振りから回路的に実装されているような印象を受けたのですが、それにしては実装すべき処理がかなり複雑な気がします。

  - ◇ すべて回路に実装されています

質問と感想

- 論理レジスタと物理レジスタの大きさを一緒にしないのは命令セットとの互換性を保つ目的ということでしょうか。
  - ◇ 初期状態で論理レジスタ個数分は物理レジスタが確保されているので，それより多い数の物理がないと投機的な実行結果が書き込めないです

質問と感想

- 物理的に多くレジスタを配置できるのであれば、それをなんとかしてソフトウェアから利用可能にすることが最も性能向上につながる気がしました。

質問と感想

- OoOでクロスコアの場合、注意すべき点は何ですか？

質問と感想

- register renameではWAW依存はメモリ書き戻し時に先送りしたという認識であっていますか？その場合、どのような実装がなされるのかは気になりました。
  - ◇ 先送りしたわけではなく，別の場所に書くようにして，後から書いた方が解放されずに残る形です