内容

- レジスタ・リネーミング
  - リオーダ・バッファ＋レジスタ・ファイル
  - 大きなレジスタ・ファイル（のみ）

- メモリ命令のスケジューリング
  - ストア・バッファ
  - ロード／ストア・キュー

レジスタ・リネーミングの目的

- 出力依存と逆依存の除去
  - 少数のレジスタで多くの変数を保持
    →レジスタ再利用→出力依存と逆依存
  - レジスタをできるだけ再利用しない
    ←物理的にレジスタを増やす

- 例

リネーミング前

```
i1: r1 = ...
i2: r2 = r1 ...
i3: r1 = r2 ...
i4: r2 = r1 ...
```

リネーミング後

```
i1: p10 = ...
i2: p11 = p10 ...
i3: p12 = p11 ...
i4: p13 = p12 ...
```

論理レジスタと物理レジスタ

- 論理レジスタ
  - ISA（命令セット・アーキテクチャ）でのレジスタ
  - プログラムに現れるレジスタ

- 物理レジスタ
  - 実行時に実際に値を保持するレジスタ
  - マイクロアーキテクチャ・レベルでのレジスタ

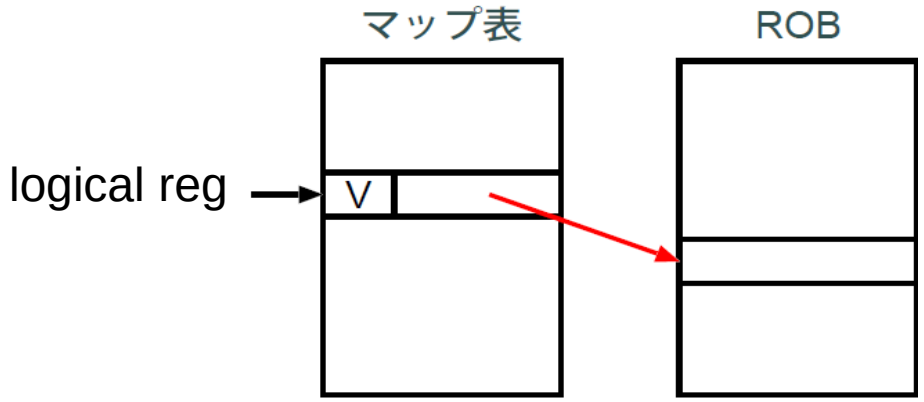
- 抽象レベルの階層化
  - 各レベルは他のレベルに大きな影響を与えない
  - ソフトウェア資産価値の保持
  - ハードウェアを独自に高速化する

論理レジスタと物理レジスタの関係

- マップ表
  - 論理レジスタ → 物理レジスタ
- 物理レジスタの実現方法
  - リオーダ・バッファ＋レジスタ・ファイル
  - 大きなレジスタ・ファイル（のみ）

ROBとRFによる方法

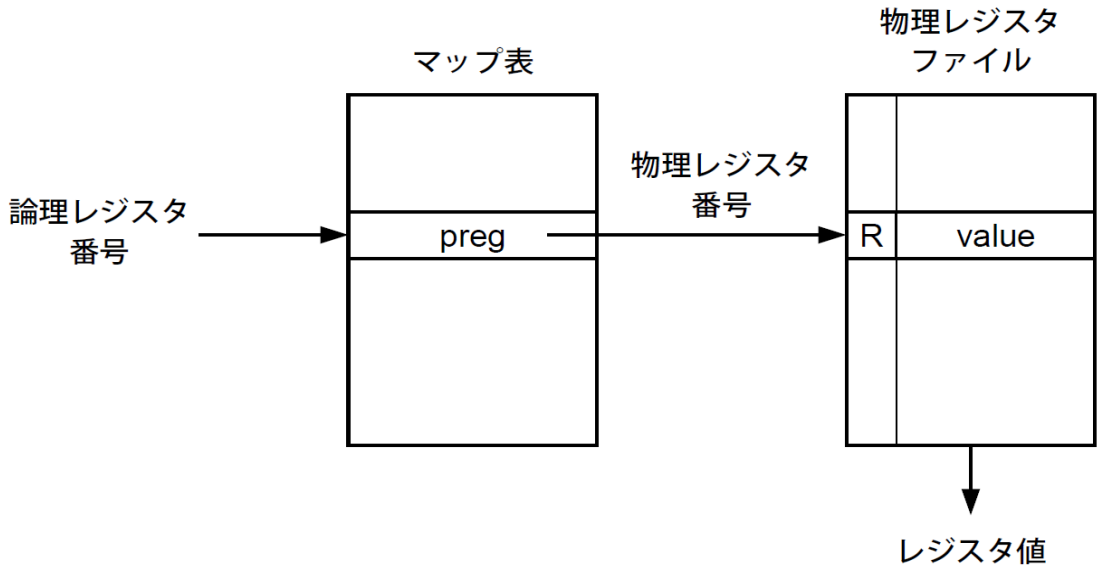
- マップ表：論理レジスタ→ROBエントリ



  - V:有効ビット
  - if (map[reg].V)
    物理レジスタ = ROBのmap[reg].pregエントリ
    else
    物理レジスタ = RFのlogical reg
  - ROBのタグによる検索不要←→表を2回参照

レジスタ・ファイルによる方法

- 大きなレジスタ・ファイルとマップ表



  - ROBに実行結果を記憶するのではなく、1つのRFに記憶

課題

- 物理レジスタの再利用

- 正確な例外

物理レジスタの再利用

- 物理レジスタの再利用
  - ROB: コミットされれば再利用可能
  - RF: ？？？

- 空き（使い終わった）のレジスタ
  - 空きになったら、フリー・リストに返す
  - 再利用される（ 新たな命令に割り当てられる）

- 「空き」とは
  - そのレジスタ値は、以後、参照されることは絶対ない
    - 死んだレジスタ

フリー・リストへのレジスタ返却

- レジスタが死ぬ
  - 生存期間の最後の時点
  - ハードウェアには正確にはわからない
  - 論理レジスタが再定義されるとき→以前の物理レジスタをFLへ

```
ix: r1 = ...
    ...
iy: ... = r1
    ...
iz: r1 = ...
```

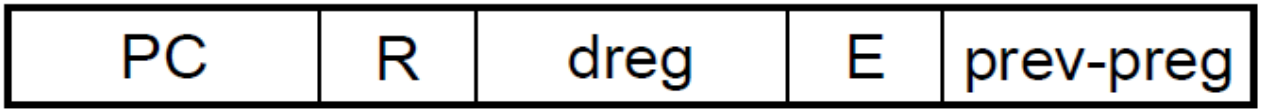
```
ix: p10 = ...
    ...
iy: ... = p10   ← p10死だがHWには？
    ...
iz: p11= ...    ← p10をFLに
```

- 例外を考慮した死
  - 死んだレジスタも例外により再使用される可能性がある
  - 再定義がコミットされたとき、以前に割り当てられていた物理レジスタは死ぬ
    - iyより前の命令が例外を起こしたとき

ROBの修正

- デスティネーション・レジスタに以前に割り当てられていた物理レジスタ
  - prev-preg
  - コミット時にprev-pregをフリー・リストへ

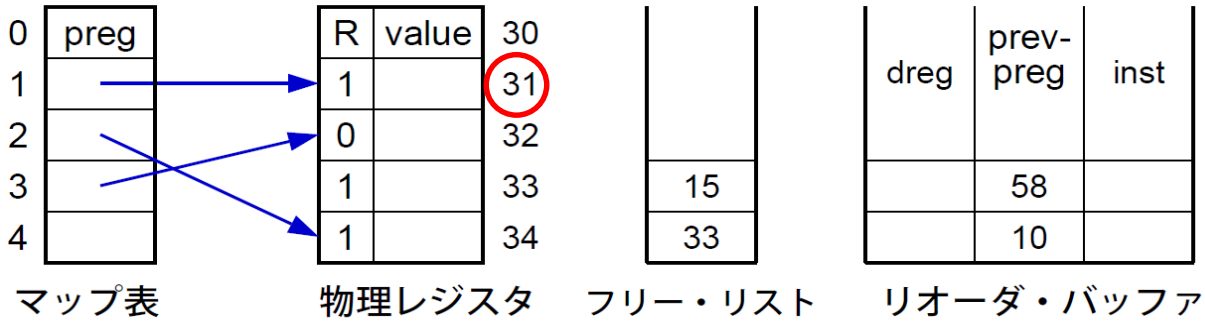
- 不要なフィールド
  - 実行結果→ レジスタ・ファイル
  - タグ　　→ マップ表の物理レジスタ番号をタグ



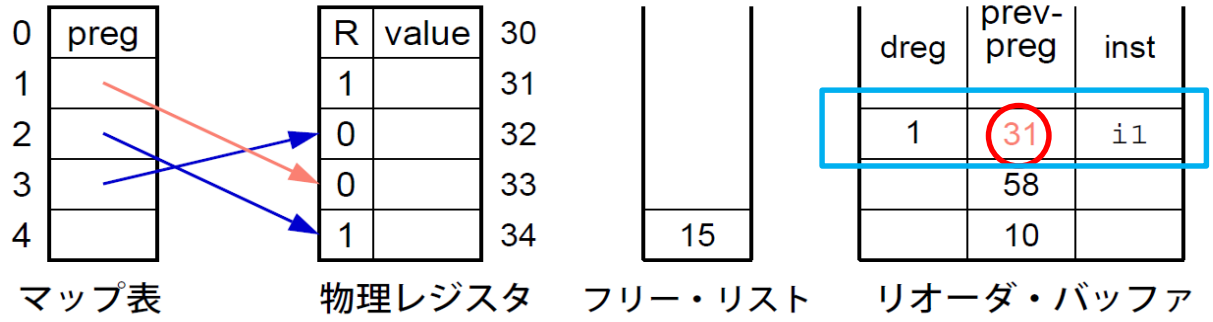
例

```
i1: r1 = r2 + r3
```

初期状態

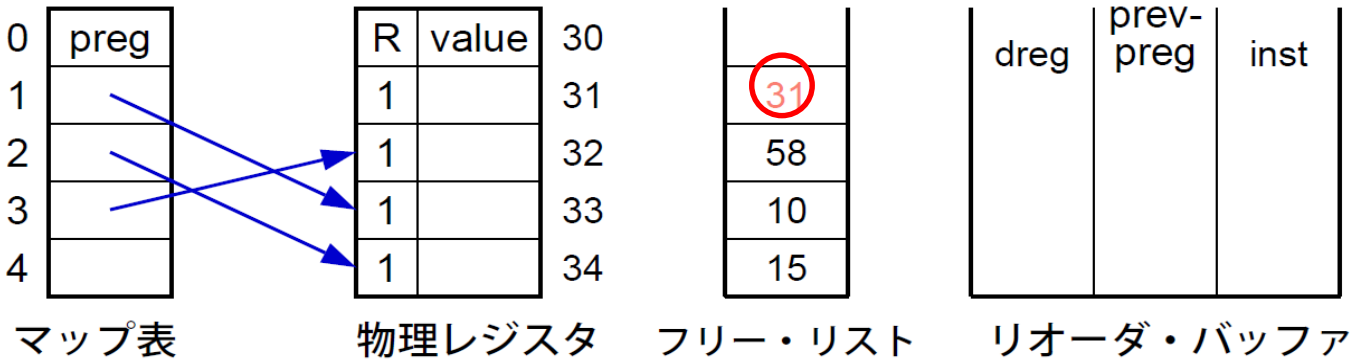


リネーミング



例(cont'd)

コミット


0
1
2
3
4
preg
R
value
1
1
1
1
30
31
32
33
34
31
58
10
15
dreg
prev-
preg
inst
マップ表
物理レジスタ
フリー・リスト
リオーダ・バッファ

課題

- 物理レジスタの再利用
- 正確な例外

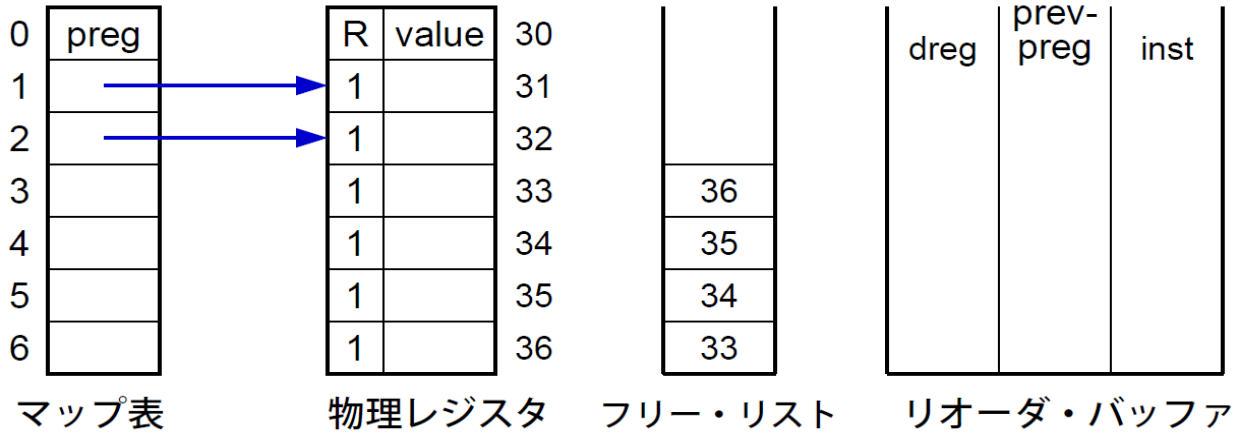
正確な例外

- ROB+RF
  - RFに正確な状態
  - ROBには未コミット状態→ROB無効化
  - H/Wは単純

- RFのみ
  - 正確な状態と将来の状態は混ざっている
  - マップ表は最新状態を表す
    - マップ表を例外発生点に戻す
  - 巻き戻し
    - ROBの末尾から先頭に向かって
    - prev-pregがマッピングの履歴を持っている
      → prev-pregでマッピングを戻せばよい

巻き戻しの例

- 初期状態

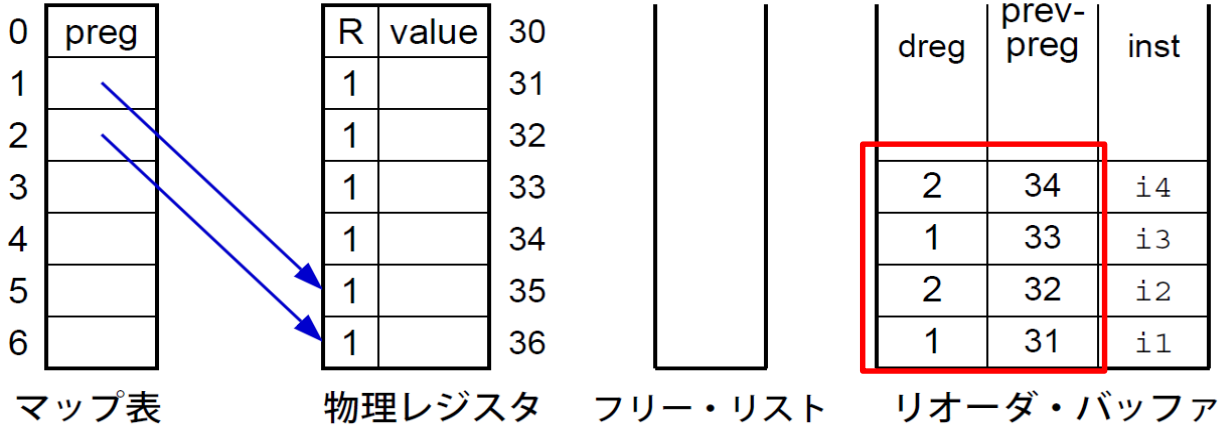



```
i1: r1 = ...
i2: r2 = ...
i3: r1 = ...
i4: r2 = ...
```

巻き戻しの例(cont'd)

- 実行後

```
i1: r1 = ...
i2: r2 = ...
i3: r1 = ...
i4: r2 = ...
```



- i1例外から巻き戻し
  - ROBの末尾から先頭に向かって順に、prev-preg, dregを見て、マップ表を巻き戻し

リタイアメントRAT

- 巻き戻し
  - MIPS R10000
  - 設計がやや複雑→有限状態機械
  - 分岐予測ミスからの回復に時間がかかる
    - R10000では、例外からの回復に用いられているが、分岐予測からの回復には別の方法がとられている

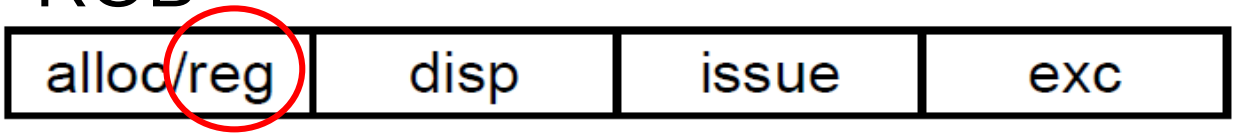
- リタイアメントRAT (register alias table)
  - コミット時のマップ：正確な状態を表す
  - 通常のマップ表をフロントエンド RATと呼ぶ
  - 例外をコミットしたら、リタイアメント RATをフロントエンド RATにコピー→単純
  - Intel Pentium 4

物理レジスタがレジスタ・ファイルのみの方式の利点

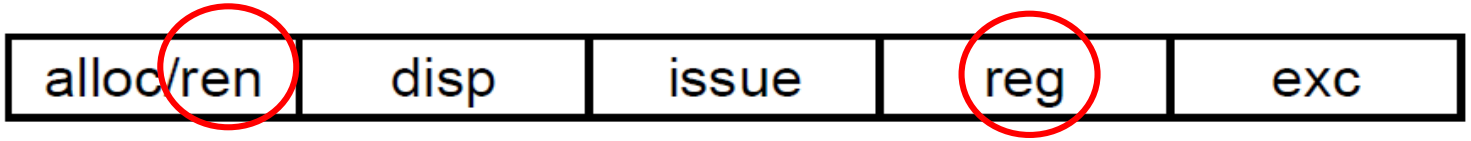
- 物理レジスタ数を節約
  - すべての命令がデスティネーション・レジスタを持つわけではない
- 命令ウィンドウにレジスタ値を保持しなくてもよい
  - 命令ウィンドウが小さくなる
  - 命令ウィンドウへのフォワーディングも不要
  - 単純
  - 高速
  - 発行キュー（issue queue)と呼ばれる

パイプライン

- RF+ROB



- RF



  - フロントエンドではリネームだけ行う
  - 発行後にレジスタ読み出し
  - 命令ウィンドウにレジスタ・オペランドを保持する必要なし
    →小型化、高速化
  - 機能ユニットから命令ウィンドウへのフォワーディングなし
    →配線が必要ない
    →命令ウィンドウをさらに小型化、高速化

商用プロセッサ

- RF+ROB
  - Intel Pentium Pro(1995), Pentium M(2003), Core(2006)
  - P6タイプと呼ばれている
- RF
  - MIPS R10000(1994), DEC/Compaq Alpha 21264 (1997), Intel Pentium 4(2000), Sandy Bridge(2011), Haswell(2013), Skylake(2015)
  - 最近のプロセッサはこの方式

内容

- レジスタ・リネーミング
  - リオーダ・バッファ＋レジスタ・ファイル
  - レジスタ・ファイル（のみ）

- メモリ命令のスケジューリング
  - ストア・バッファ
  - ロード／ストア・キュー

メモリ依存に関する命令スケジューリング

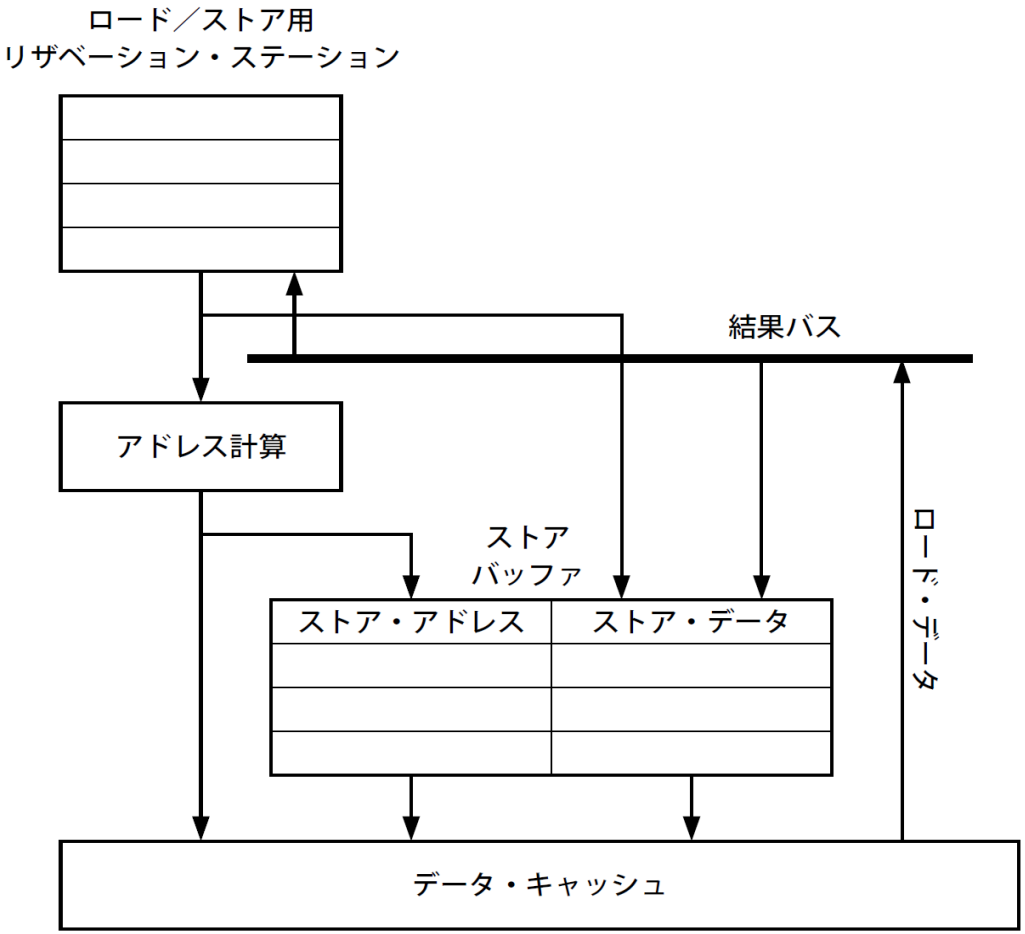
- メモリ・ロケーションは非常に多い
  - 論理空間≧$2^{32\sim64}$
  - 比較の複雑さ大
  - 表のコスト大

- アドレスは実行時にしか得られない
  - フロントエンドで依存を解析できない
  - スケジューリング→実行？
    実行→スケジューリング？
    - 卵と鶏

スケジューリング方法

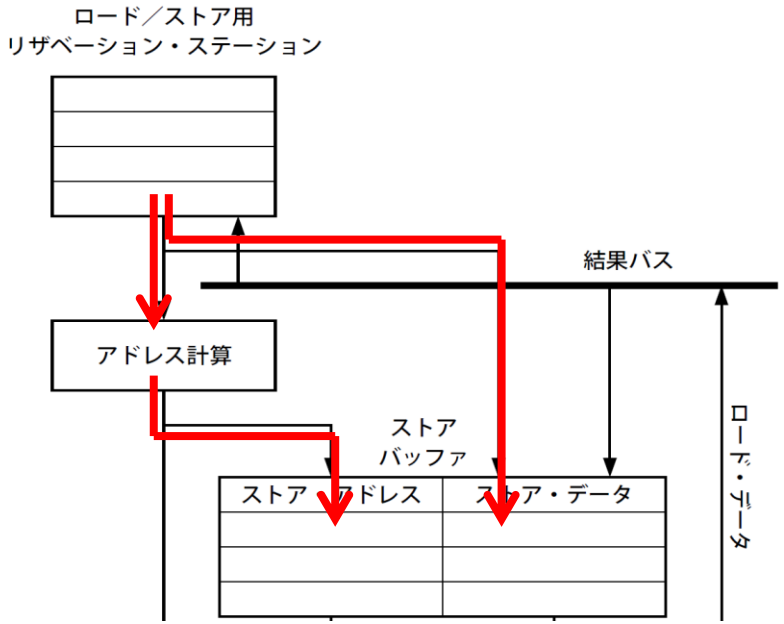
- イン・オーダ実行
  - 性能を大きく制限
  - 全動的命令の1/3はメモリ命令
- ロード命令間でのみアウト・オブ・オーダ実行
  - ロードはメモリ状態を変更しない
  - ストアとロードの順序を変えない→偽の逆依存
  - 性能を制限
    - 一般にロードのレイテンシは長い
- ロードとストアの実行順の変更
  - ストア・バッファ
  - ロード／ストア・キュー

ストア・バッファ


ロード／ストア用
リザベーション・ステーション
結果バス
アドレス計算
ストア
バッファ
ストア・アドレス
ストア・データ
ロード・データ
データ・キャッシュ

動作(1)

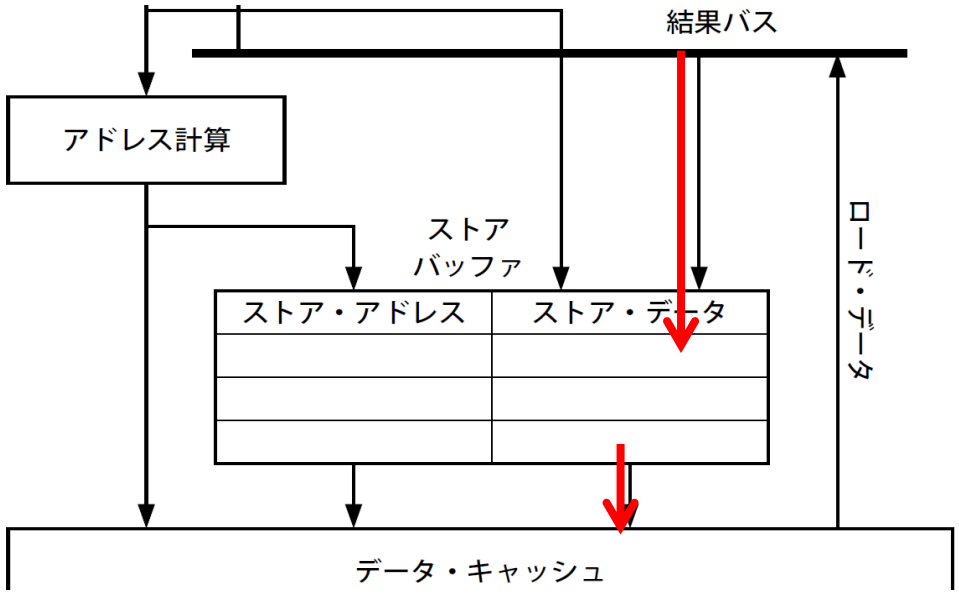
- 発行
  - インオーダ←依存チェックが必要なため（後述）
  - アドレス計算
  - ストア命令について、ストア・バッファのエントリ割り当て
    - データ・アドレス書き込み
    - ストア・データがあれば、書き込み



動作(2)

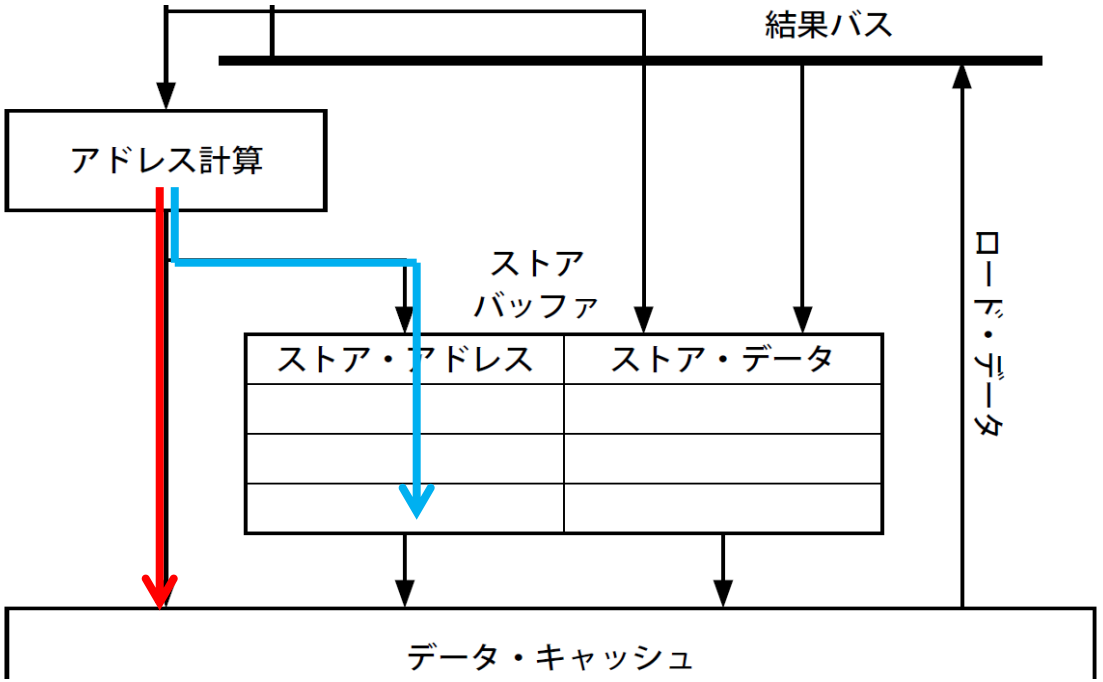
- ストア・バッファ
  - 他の命令の実行結果がストア・データの場合、フォワーディングされるのを待つ
  - コミットされていないストアの待ち合わせ
  - ストアがROBからコミット

    →ストア・バッファからデータ・キャッシュに書き込み



動作(3)

- ロードのストア追い越し
  - アドレスをキャッシュに送出し、読み出し要求
  - ストア・バッファのアドレスをチェック
    - 一致　　:依存あり⇒発行キャンセル
    - 不一致 :依存なし⇒ロード値を得る



メモリ曖昧性除去

- memory disambiguation

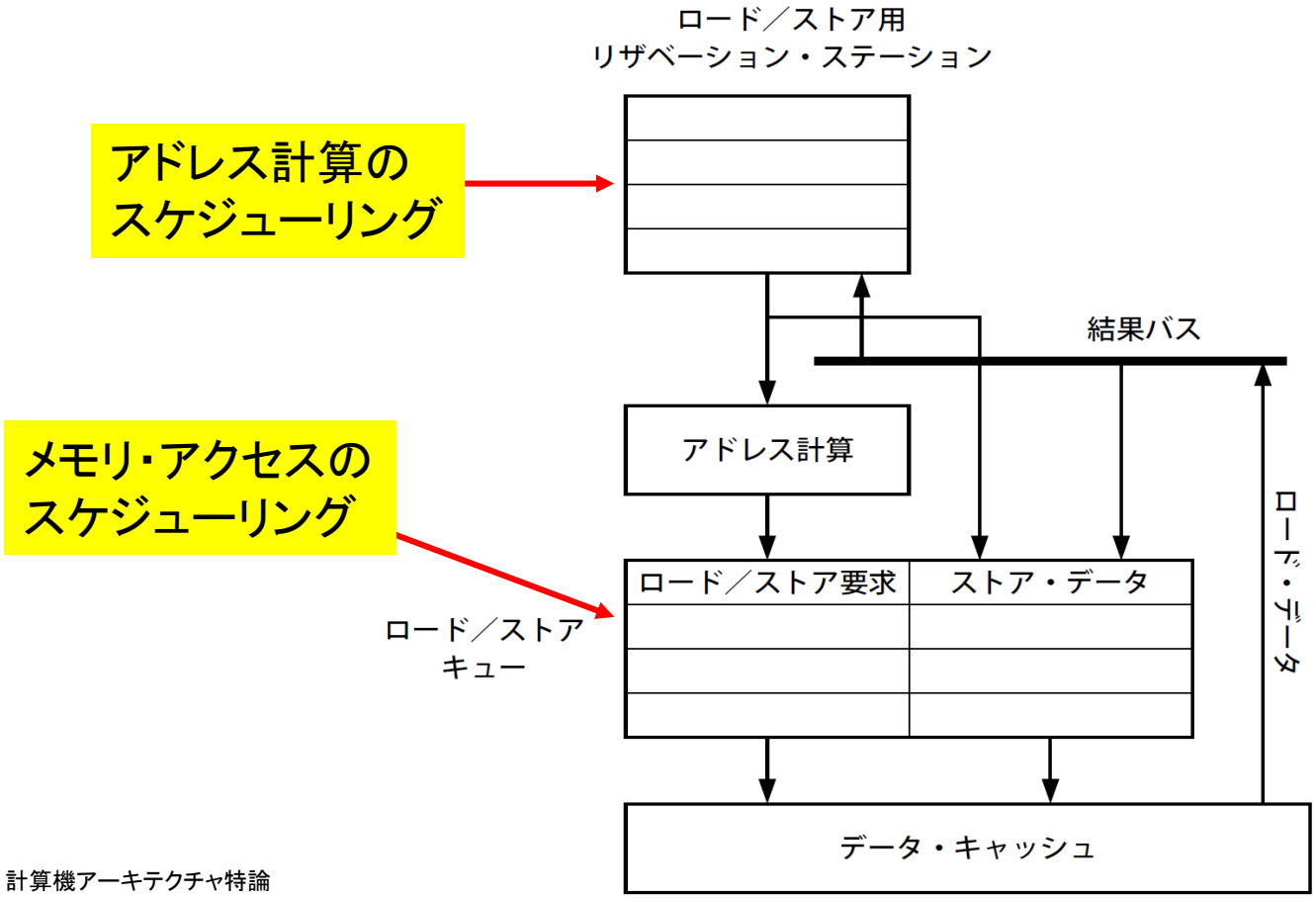
- メモリ依存をチェックすること
  - エイリアス・チェック

ストア・バッファ方式の長所と短所

- 長所
  - 依存のないロードは、ストアを追い越せる
- 短所
  - アドレス計算はインオーダ

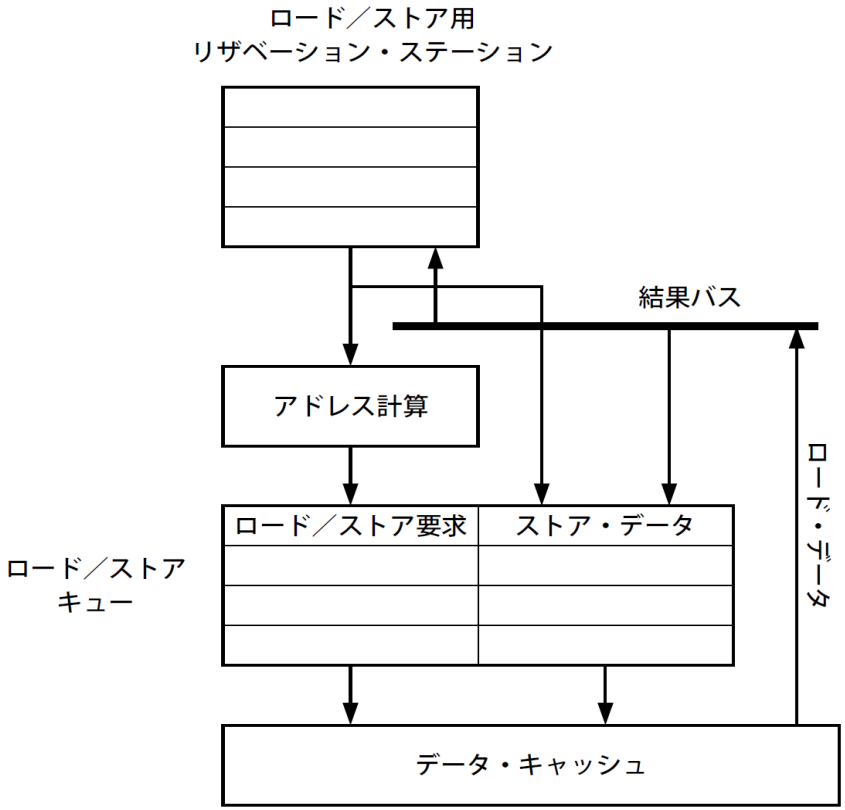
分離ロード／ストア

- アドレス計算とメモリ・アクセスを別々にスケジューリング



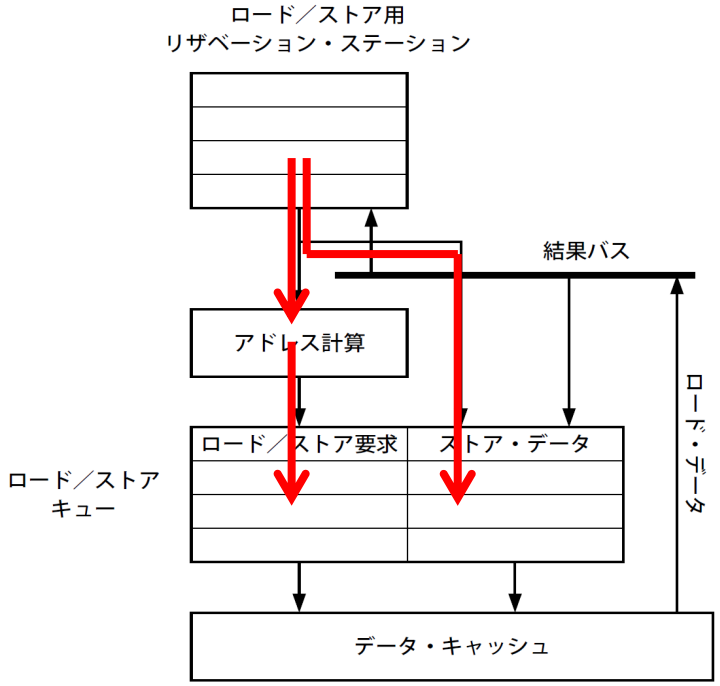
ロード／ストア・キュー(LSQ) (1)

- エントリ割り当て
  - フロントエンドでイン・オーダ割り当て



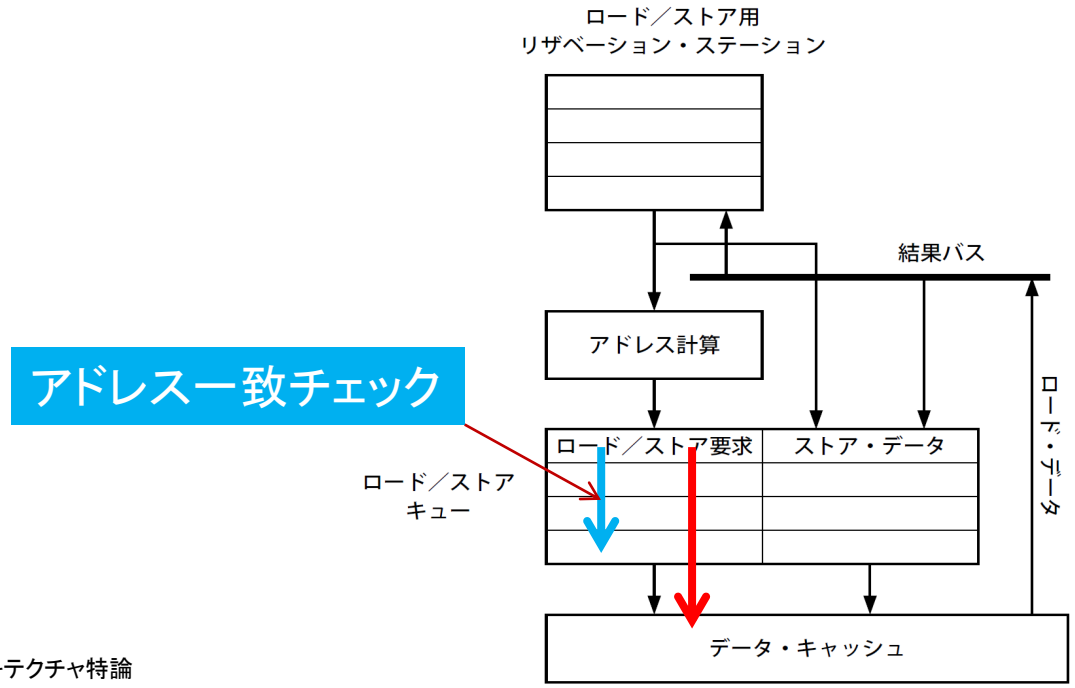
ロード／ストア・キュー(LSQ)(2)

- LSQへの書き込み
  - データ・アドレス
  - （ストアの場合）ストア・データ
  - アウト・オブ・オーダ



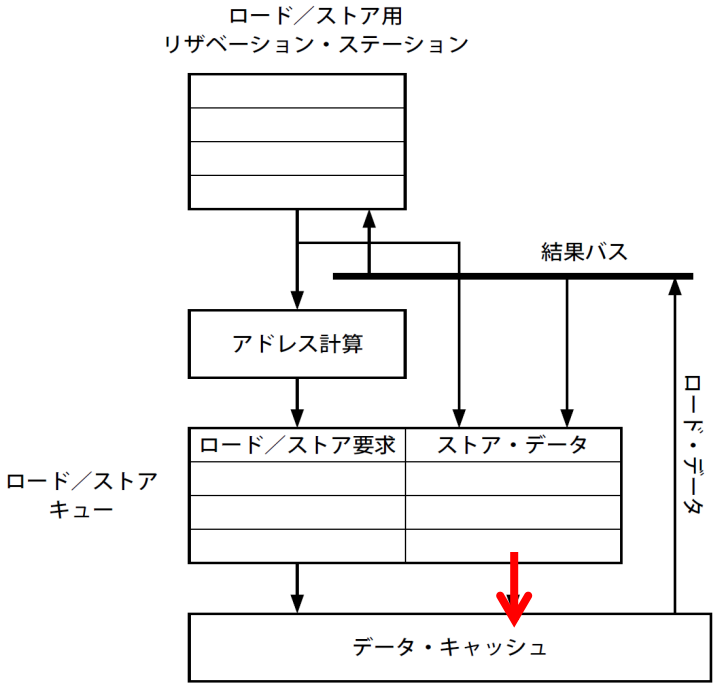
ロード／ストア・キュー(LSQ)(3)

- LSQからデータ・キャッシュへのロード要求
  - 自身のデータ・アドレスが得られている
  - 先行するストアのデータ・アドレスが得られている
  - 先行ストアに依存がない→アドレスの連想検索→複雑



ロード／ストア・キュー(LSQ)(4)

- LSQからデータ・キャッシュへのストア要求
  - 自身のデータ・アドレスとストア・データが得られている
  - ROBからのコミット⇒データ・キャッシュに書き込み



メモリ依存投機

- 先行するストアのデータ・アドレスが得られていなくても、ロードを実行する
  - メモリ命令が多いx86では、特に重要
- 依存予測: ストア・セット予測器
  - G. Z. Chrysos and J. S. Emer, Memory dependence prediction using store sets, In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp.142-153, June 1998.
  - 非常に高い予測精度
  - これが商用プロセッサに使われているかどうかはドキュメントされていないので、不明
- 後でストアに依存があったことがわかると、予測を誤ったロードから再実行
  - 複雑

まとめ

- レジスタ・リネーミング
  - 出力依存と逆依存の除去
  - 論理レジスタを物理レジスタにマッピング
  - 物理レジスタ：ROB＋RFまたはRF
  - マップ表

- メモリ命令のスケジューリング
  - 実効アドレスがわからないと、スケジューリングできない
  - イン・オーダ→大きな性能制限
  - ストア・バッファ
    - ロードのストア追い越し
    - アドレス計算はイン・オーダ
  - ロード／ストア・キュー
    - アドレス計算もアウト・オブ・オーダ
    - 複雑