内容

- 命令レベル並列とは
- 並列実行への制約と命令スケジューリング
- スーパスカラ･プロセッサの基本構成
- 命令発行ポリシー
  – イン･オーダ
  – アウト･オブ･オーダ

命令の並列実行へ(1)

- パイプライン・ストールの原因
  - 構造ハザード
  - データ・ハザード
  - 制御ハザード
- 解消法
  - フォワーディング
  - 遅延ロードや遅延分岐＋命令スケジューリング
  - 分岐予測

命令の並列実行へ(2)

- CPI
  - ET = N × CPI × T
    - ET: 実行時間、T: クロック・サイクル時間
  - CPI (clock cycles per instruction)はほとんど1=下限
  - クロック速度を上げる以外、性能を向上させる方法がない
- 次のステップ
  - CPIを1以下へ
  - 1クロックで複数の命令を並列に実行
  - IPC (instructions per clock cycle)を1以上へ

並列実行

- 複数のパイプライン
  - IPC>1.0
  - 命令間の並列性を利用

命令 i1	IF	ID	EX	MEM	WB		
命令 i2	IF	ID	EX	MEM	WB		
命令 i3		IF	ID	EX	MEM	WB	
命令 i4		IF	ID	EX	MEM	WB	
命令 i5			IF	ID	EX	MEM	WB
命令 i6			IF	ID	EX	MEM	WB

命令レベル並列の定義

- 命令レベル並列
  - ILP: instruction-level parallelism
  - 粒度 : 並列に実行する1つの単位に含まれる仕事の大きさ
  - 粒度が1命令である並列
  - その他の粒度 :
    - ループレベル、関数レベル…
    - マルチコア、マルチプロセッサ : 複数のプロセッサ

並列性の制約

- 理想

命令 i1	IF	ID	EX	MEM	WB		
命令 i2	IF	ID	EX	MEM	WB		
命令 i3		IF	ID	EX	MEM	WB	
命令 i4		IF	ID	EX	MEM	WB	
命令 i5			IF	ID	EX	MEM	WB
命令 i6			IF	ID	EX	MEM	WB

- 3つの制約
  - 資源競合
  - データ依存
  - 制御依存

資源制約

- 資源競合による制約
  - 実行に必要な資源が得られない

- 例

```
i1: r1 = r2 + r3
i2: r4 = r5 + r6
i3: r7 = load 0(r2)
i4: r8 = load 4(r3)
```

  - これらに意味上の関係はない→並列
  - 加算器:2つ必要
  - データ・キャッシュの読み出しポート:2つ必要
  - レジスタの読み出しポート:6つ必要

資源制約の解消

- 要求に対応できるように、たとえば、複数の資源を用意
- トレードオフ
  - コスト対並列性（性能）
    - 加算器を2つにすれば、コストは2倍
    - レジスタ・ファイルのポートを2から8にすれば、コストは16倍
  - 複雑さ対並列性（性能）
    - レジスタ・ファイルのアクセス時間は長くなる

データ依存制約

- 3つの型
  - 真の依存(true dependence)
  - 逆依存(anti-dependence)
  - 出力依存(output dependence)

真の依存

- 書き込み→読み出しの関係
  - レジスタ

```
i1: r1 = r2 + r3
i2: r4 = r1 + 1
```

a = …
… = a + …

  - メモリ

```
i1: store A(r3) = r1
i2: r2 = load A(r3)
```

  - プログラムの意味から生じる→除くことは難しい

逆依存

- 読み出し→書き込みの関係
  - レジスタ

```
i1: r1 = r2 + r3
i2: r2 = r4 + 1
```

... = a + ...
b = ...

    - コンパイラによる変数のレジスタ割り当て
    - レジスタの再利用→一種の資源競合

      レジスタの割り当て方を変えれば解決
    - 人工的な依存
  - メモリ

```
i1: r2 = load A(r1)
i2: store = A(r1)
```

出力依存

- 書き込み→書き込みの関係
  - レジスタ

```
i1: r1 = r2 + r3
i2: r1 = r4 +  2
```

    - 人工的な依存
    - レジスタの割り当てを変えれば解決
  - メモリ

```
i1: store A(r2) = r1
i2: store A(r2) = r3
```

a = …
b = …

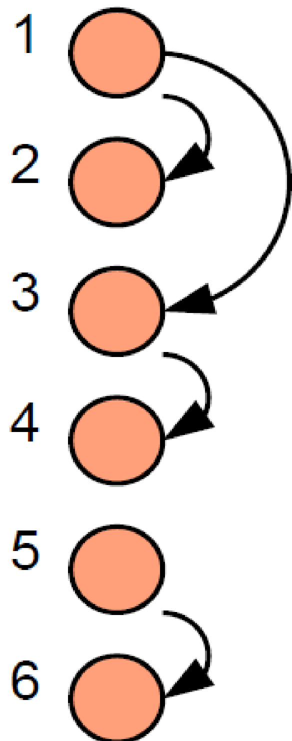
命令スケジューリング

- 目的
  - 制約を守り、最大の並列性を引き出す
- 命令の並べ替え
  - 制約を守る→プログラムの意味を保つ
  - 同時に実行可能な命令を見つける→IPC向上

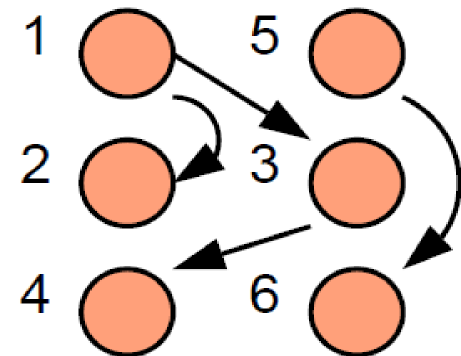
命令スケジューリング(cont'd)

元の順

並列実行可能な順



命令スケジューリング



いつ行うか？

- 実行時
  - 動的スケジューリング
  - スーパスカラ・プロセッサ (superscalar processor)
- コンパイル時
  - 静的スケジューリング
  - VLIW (very long instruction word)

なぜ動的スケジューリングなのか？(1)

- バイナリ互換性
  - ソフトウェアがプロセッサAで動作→プロセッサBでも動作
  - ユーザ
    - 経済的、性能享受
  - ソフトウェア・メーカ
    - 経済的←再コンパイル不要

- 情報産業の資産
  - ソフトウェア
    - 書き換え困難
    - 寿命は長い←数十年
  - ハードウェア
    - 寿命は短い←数年
    - LSI技術の急速な進歩
  - ハードウェア・コスト<<ソフトウェア・コスト

なぜ動的スケジューリングなのか？(2)

- 実行時情報の利用
  - キャッシュ・ミス
  - 動的スケジュールを行うなら、ミスがすなわちストールではない
    - 後ろの命令が実行され、れいてんしは隠蔽される

なぜ静的スケジューリングなのか？

- ハードウェア・コストのコスト制約が非常に厳しいマーケット
  - 単純なハードウェアで小面積
  - バイナリ互換性がなくてもOK
    - 「ハードウェアコスト>>ソフトウェアコスト」な分野
    - 組み込み機器
- 高速クロック
  - 単純なハードウェア
- 低電力

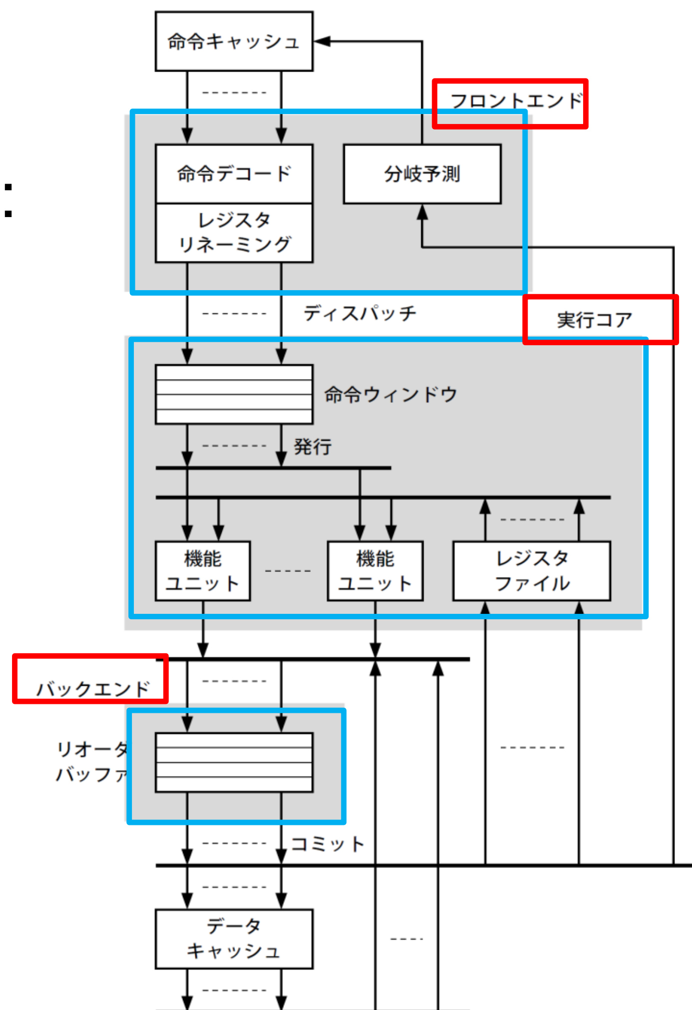
- 動的コンパイル
  - 既存のバイナリをプロセッサに適合したバイナリに変換
  - Transmeta Crusoe Code Morphing

内容

- 命令レベル並列とは
- 並列実行への制約と命令スケジューリング
- スーパスカラ･プロセッサの基本構成
- 発行ポリシー
  - イン･オーダ
  - アウト･オブ･オーダ

基本構成

- デカップル
  - 3つの部分が独立して動作:
    - フロントエンド
    - 実行コア
    - バックエンド



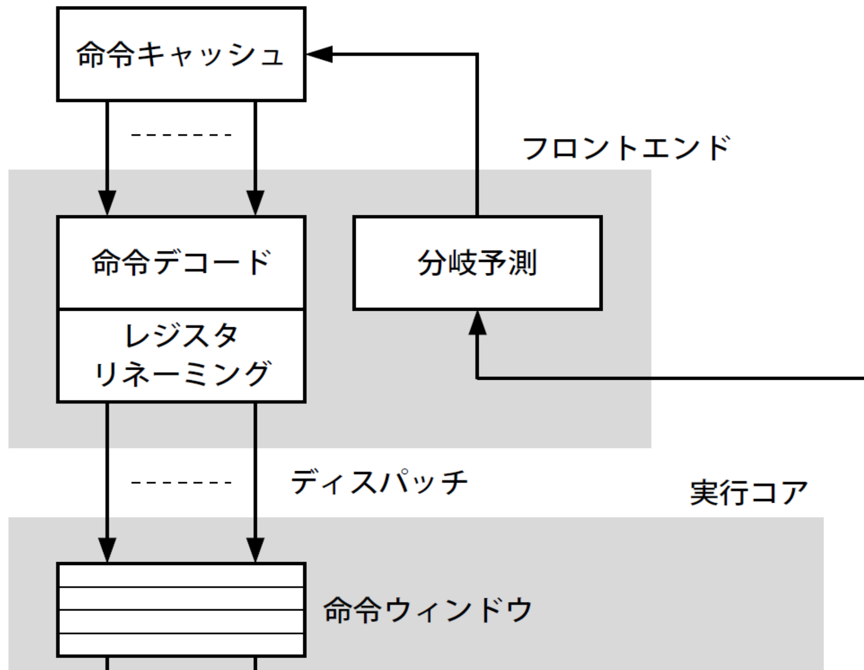
各部の役割

- フロントエンド
  - プログラム順
  - 命令フェッチ
  - レジスタ・リネーミング
  - データ依存関係解析
- 実行コア
  - 命令スケジューリング
  - 並列に実行可能な命令を見つけて実行
    - プログラム順ではない
- バックエンド
  - プログラム順にレジスタとメモリへの書き込み

用語

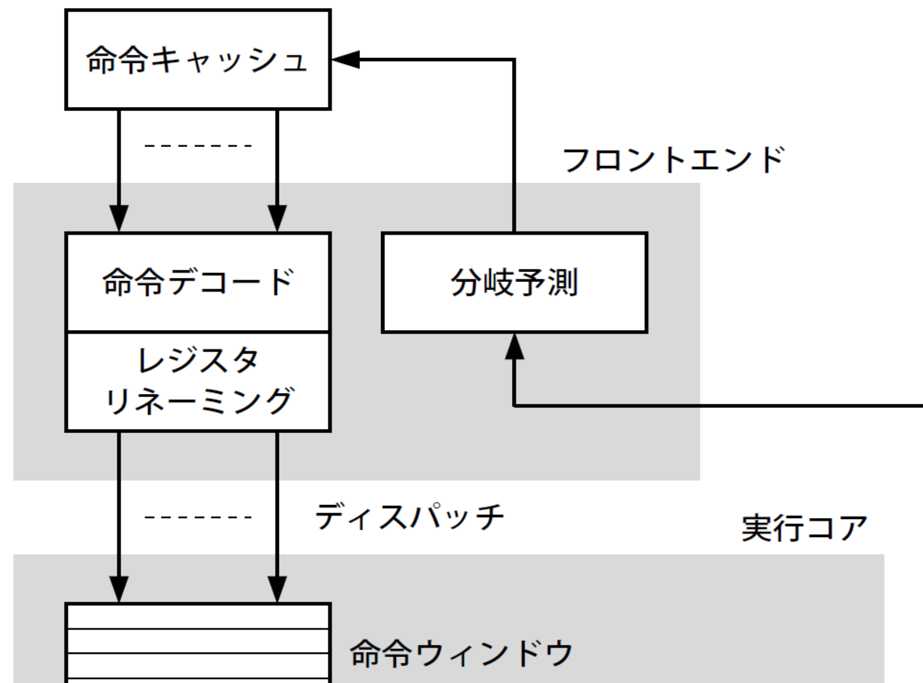
- アウト･オブ･オーダ(out-of-order)
  - プログラム順でない順
- イン･オーダ(in-order)
  - プログラム順

フロントエンド



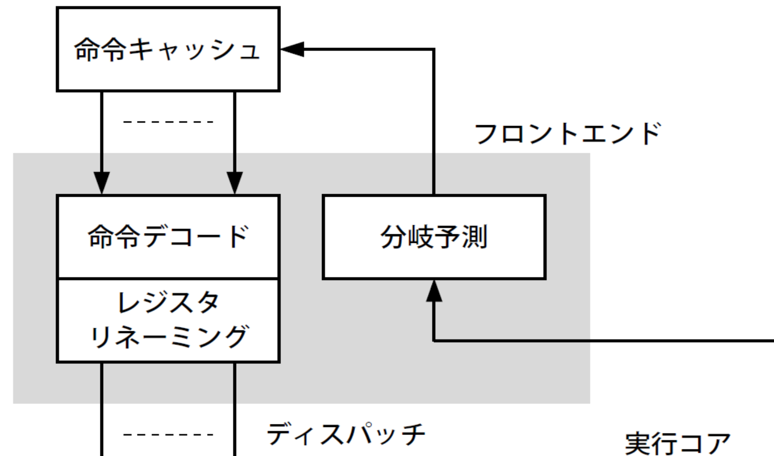
- 複数の命令フェッチ
  - 例：4命令フェッチ：IR←I-Cache[PC..PC+12]
  - PC←PC+16

フロントエンド(cont'd)



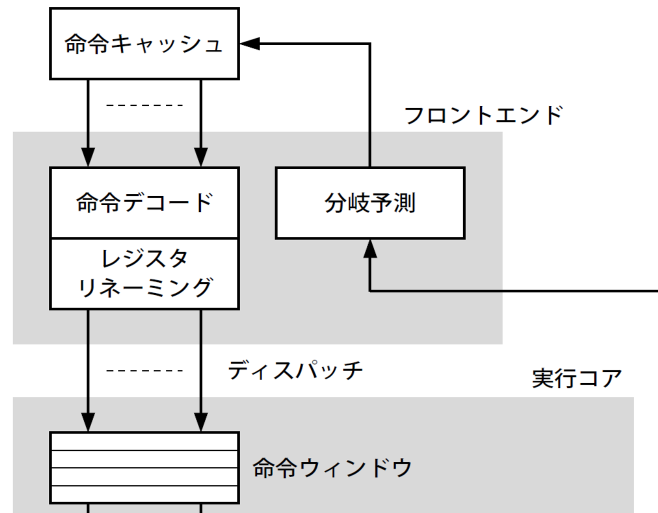
- 分岐予測
  - 分岐によるストール回避
  - 実行コアに切れ目なく、十分な数の命令を供給

フロントエンド(cont'd)



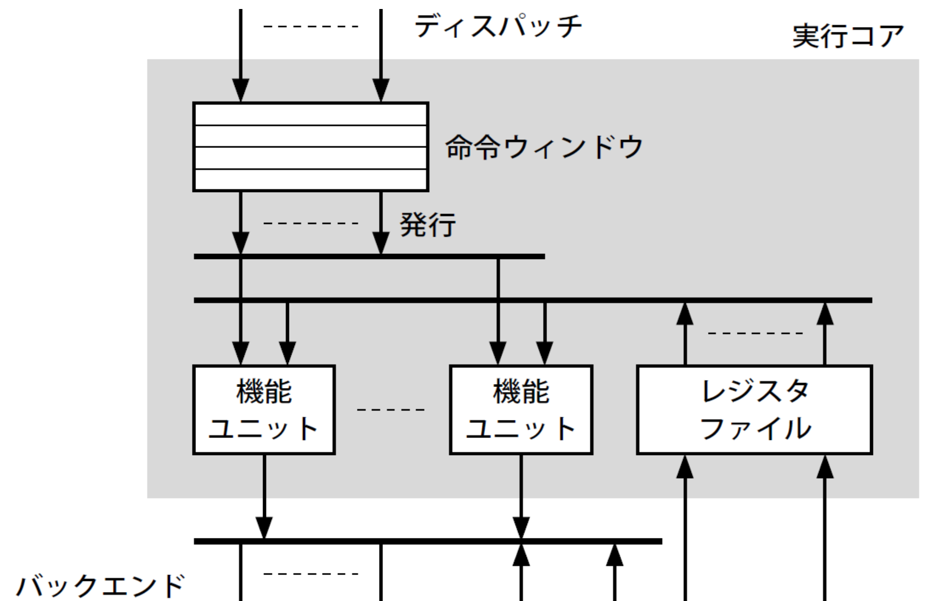
- データ依存解析
  - 真のデータ依存関係のコーディング : タグ
    - Tomasuloのアルゴリズム
    - アウト・オブ・オーダで命令を実行するため
    - 命令スケジューリングに使用
- レジスタ・リネーミング
  - 出力依存と逆依存を除去→並列性の増加
- リオーダ・バッファの割り当て

フロントエンド(cont’d)



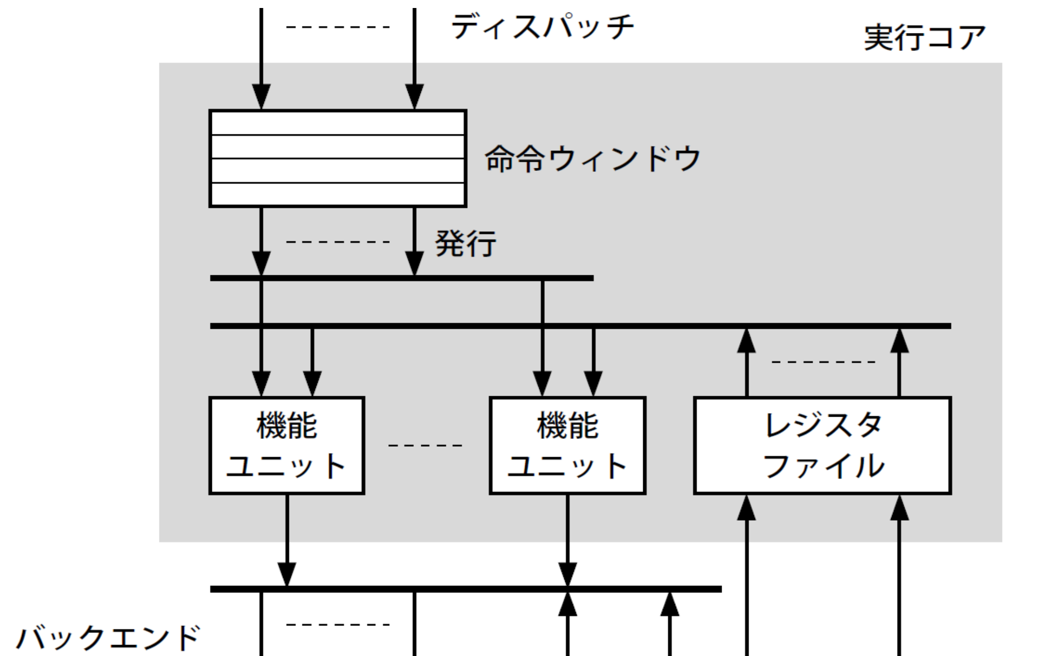
- ディスパッチ
  – 命令ウィンドウに書き込む
    - 命令
    - タグ
    - …

実行コア



- 命令ウィンドウ
  - フロントエンドからの命令を格納
  - 並列に実行可能な命令を見つける→アウト･オブ･オーダ
    - データ依存解決
    - 資源競合なし

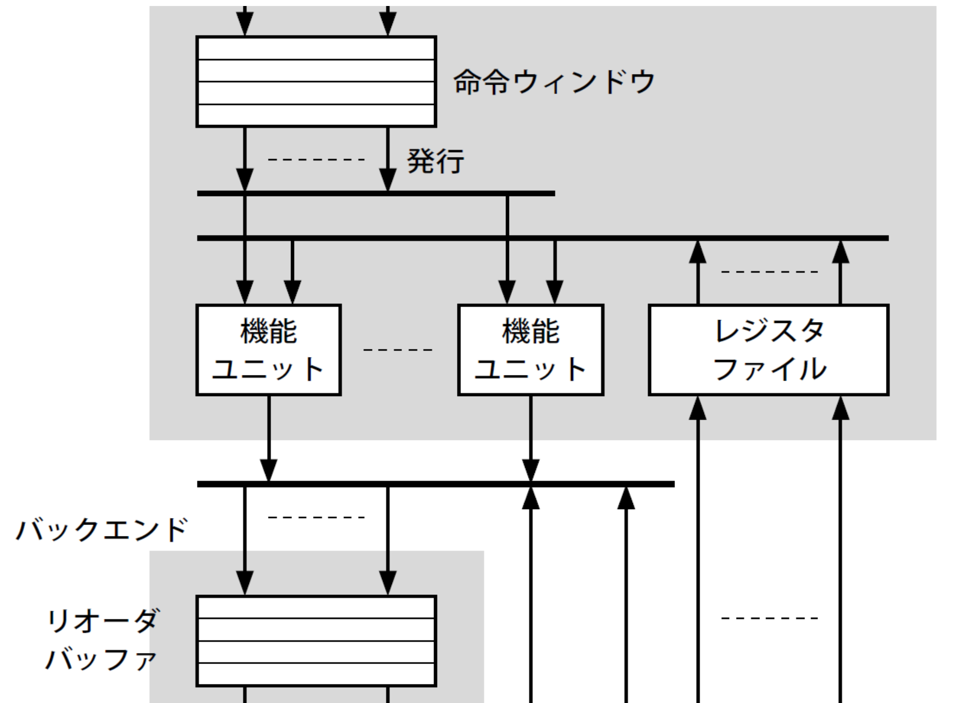
実行コア(cont’d)



r1 = ...
... = r1 ...

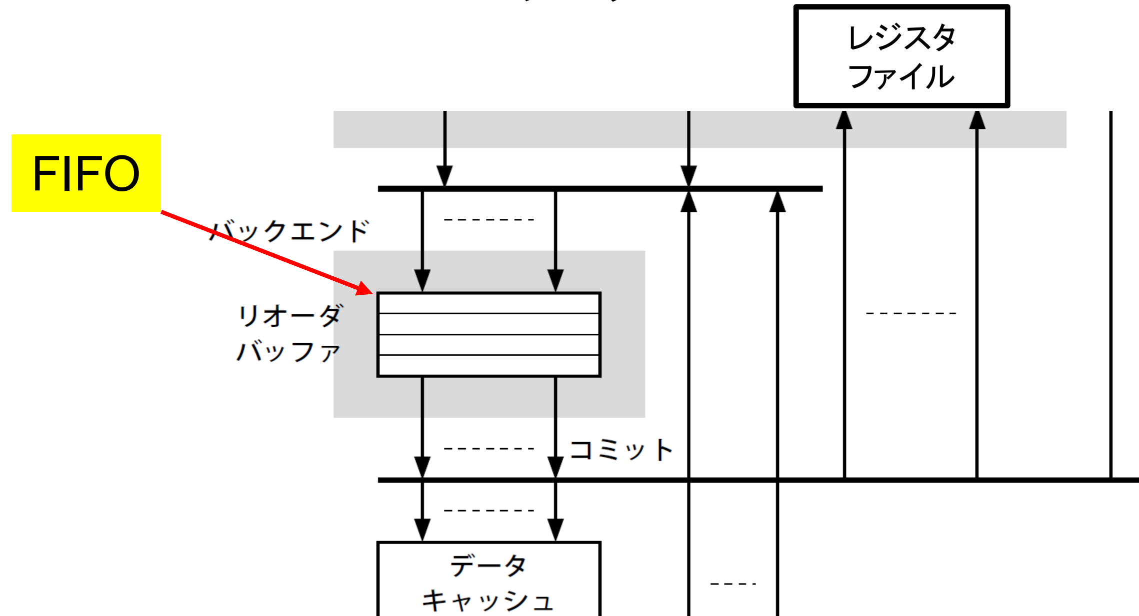
- 命令発行
  - 機能ユニットに命令を送出
  - 命令ウィンドウから削除
  - 次サイクルに利用可能になるオペランドを、命令ウィンドウに通知

実行コア(cont’d)



- 機能ユニットで同時実行
  - （複数の）ALU
  - （複数の）ロード／ストア・ユニット
- 結果をリオーダ・バッファへ書き込む

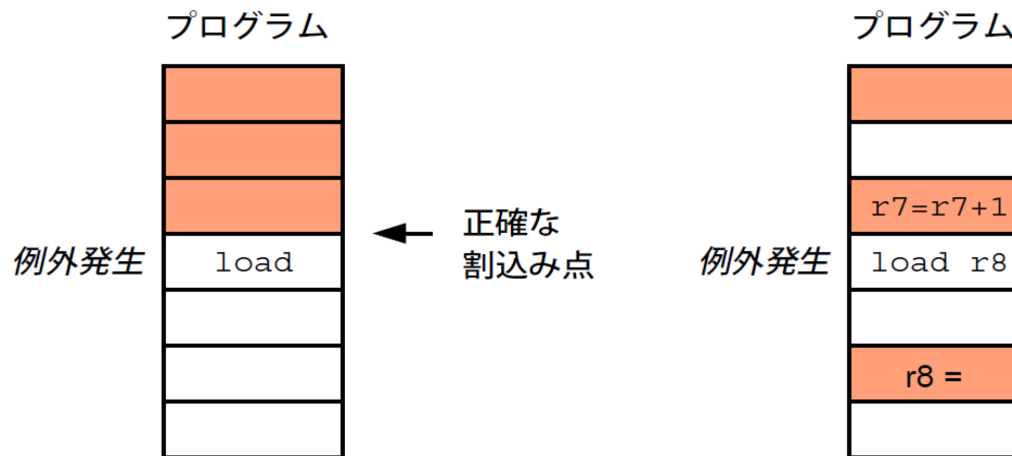
バックエンド



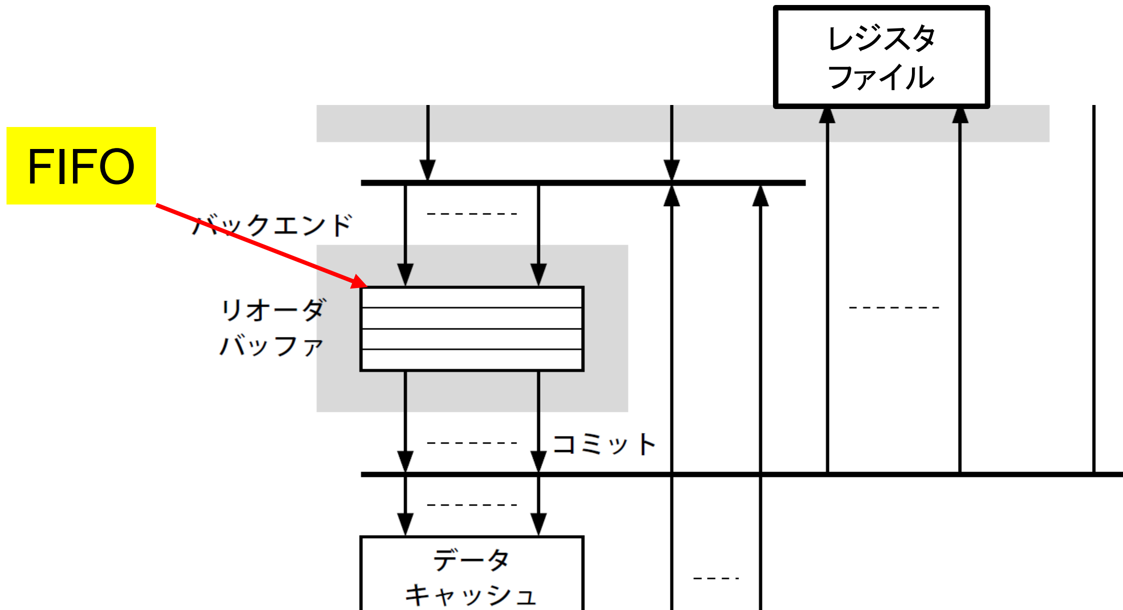
- コミット/リタイア
  - プロセッサ状態の更新
    - 逐次実行モデルにおけるレジスタとメモリの状態
  - プログラム順
    - 逐次実行モデルの維持
    - 正確な例外

正確な例外

- 定義
  - 例外を起こした命令より以前の命令の実行はすべて終了しており、かつ、後の命令はすべて終了していない
- 再開が容易
  - 再開点：例外命令
  - 再実行命令：例外命令以降



プログラム順でのコミット



- アウト･オブ･オーダ⇒プログラム順
  - ソートしなおすわけではない
- フロントエンドで、リオーダ・バッファのエントリを割り当てる→プログラム順に割り当てられる
- アウト･オブ･オーダで実行結果が、割り当てられたエントリに書き込まれる
- FIFOの先頭からコミットされる

内容

- 命令レベル並列とは
- 並列実行への制約と命令スケジューリング
- スーパスカラ･プロセッサの基本構成
- 発行ポリシー
  - イン･オーダ
  - アウト･オブ･オーダ

イン・オーダ命令発行

プログラム

i1	r1=r5
i2	r2=r1+1
i3	r3=r6
i4	r4=r3+1

スケジューリング

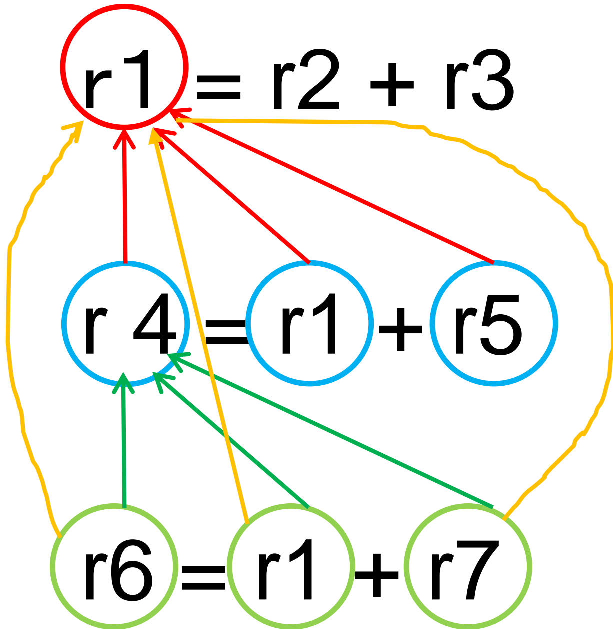
サイクル	機能ユニット0		機能ユニット1	
0	i1	r1=r5		
1	i2	r2=r1+1	i3	r3=r6
2	i4	r4=r3+1		

イン･オーダ命令発行の複雑さ

- 命令ウィンドウ･サイズ＝i命令（i: 同時発行命令数）
  - 前から最大i命令しか発行できない←イン･オーダ

- 依存チェック
  - 高々i命令間チェック
    - 各命令のデスティネーション･レジスタと全後続命令のソース･レジスタを比較ー$O(i^2)$
    - 実際上i=4程度なので、たいして複雑でない
  - 真の依存と出力依存
  - 逆依存満足
    - イン･オーダなので必ず満足
    - 同時に発行されたとしても、読み出しの方が先だから

比較器の数



$$\sum_{k=1}^{i} 3(k-1) = \frac{3}{2} i(i-1)$$

アウト·オブ·オーダ命令発行

プログラム

i1	r1=r5
i2	r2=r1+1
i3	r3=r6
i4	r4=r3+1

スケジューリング

サイクル	機能ユニット0		機能ユニット1	
0	i1	r1=r5	i3	r3=r6
1	i2	r2=r1+1	i4	r4=r3+1

- i3がi2より先に実行されていることに注意
- 高い並列度
  - プログラム順制約がない

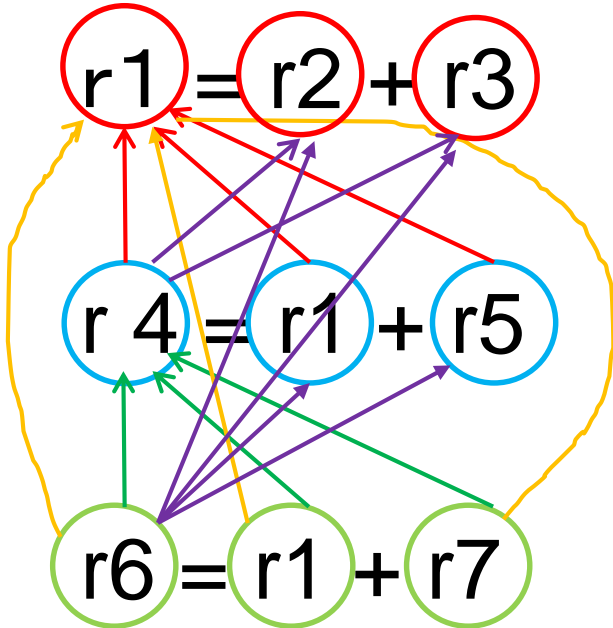
アウト･オブ･オーダ命令発行の複雑さ

- 命令ウィンドウ・サイズ＝n(>>i)命令
  - nが大きいほどIPCは向上する

    ←並列実行可能な命令が見つかる可能性が高まる

- 依存チェック
  - n(>>i)命令間のチェックー$O(n^2)$
    - e.g., n=64→10,080の比較器
    - 配線数も非常に多い
  - 真の依存と出力依存に加え、逆依存もチェック
  - イン･オーダに比べて非常に複雑

比較器の数



$$\sum_{k=1}^{n} 5(k-1) = \frac{5}{2} n(n-1)$$

まとめ

- 命令レベル並列
  - 単一パイプライン処理の限界を打破：IPC>1
  - 並列に対する3つの制約
  - 制約を守った命令スケジューリングが並列性を引き出す
- スーパスカラ・プロセッサの基本構成
  - フロントエンド：命令フェッチ、レジスタ・リネーミング…
  - 実行コア：命令スケジューリング、実行
  - バックエンド：コミット
- 命令スケジューリングの複雑さ
  - イン・オーダ命令発行
    - 単純
  - アウト・オブ・オーダ命令発行
    - 複雑→Tomasuloのアルゴリズム