自己紹介

http://www.mtl.t.u-tokyo.ac.jp/~irie

- 所属
  - 電子情報工学 /電子情報学専攻
- 研究分野
  - コンピュータシステム
    - **コンピュータ・アーキテクチャ**
    - **ヒューマン・コンピュータ・インタラクション**
    - **ディペンダビリティ/セキュリティ**
- 「**アドバンスト・コンピュータアーキテクチャ**」
  - 来年度開講

今日のお話

- **プロセッサ・アーキテクチャ**の動向
  - 現在の主な設計戦略
  - CPU単体の性能向上の重要性
- **CPUマイクロ・アーキテクチャ**による性能向上技術
  - 過去の成長手法
  - 現在の制限と材料
- **今**，一から作るならこんなCPU
  - 電力を削減しながら性能を上げるカギ
- **STRAIGHTプロセッサ**
  - **3倍速い**新型汎用アーキテクチャの設計図（既発表分）

プロセッサ・アーキテクチャの動向

2016/11/28

最近のプロセッサ

	デスクトップ Skylake(x64)	サーバ Haswell-EX(x64)	モバイル Cortex A73(ARM v8)	IoT Cortex A35(ARMv8)
プロセス世代	14nm	22nm	16nm	28nm
大きさ	数百mm^2	662mm^2	1.2mm^2	0.4mm^2
動作周波数	4GHz	2.5GHz	2.5GHz	1-2 GHz
消費電力	90W	数百W	0.8W	6mW
CPU構成	ベクタユニットつき8way oooSS x 4	ベクタユニットつき8way oooSS x 16	ベクタユニットつき8way oooSS x 4	1way 8stage
アクセラレータ	GPU	GPU	パッケージによる	パッケージによる
LLC	32MB L3	45MB L3	8MB L2	8KB-64KB L1 (1MB L2)

プロセッサ・アーキテクチャの汎用性と伸縮性

- 伸縮性
  - 表の例だけでも
    面積で数千倍　W数で数十万倍
- 共通フレームワーク
  - 核となるCPUアーキテクチャ
  - プロセス世代，命令セット
  - 構成する各要素
- 重層的な拡張
  - CPUひとつあたりの規模
  - CPUの数
  - アクセラレータの種類，数
  - キャッシュ

現在のプロセッサの成長戦略

- 地道な**コア数増**，**シングルスレッド処理能力増**
  - 予想に反して緩やかに伸び続けている
- 用途に合わせたプロセッサ設計
  - コアの**プルーニング**
  - 目的にそったコア構成（**テーラリング**）
    - **ワークロードの特徴**，**並列性**，**汎用性**があるほど特に有利
    - Deep Learning/ セキュリティHW：この傾向に合致
- **Cache Coherency Network**
  - ヘテロ構成のコア間の連携強化

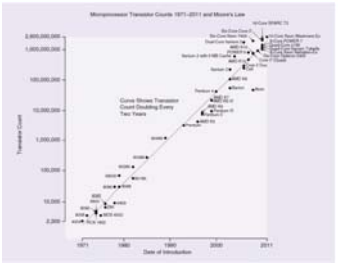
コアの種類と特性

	プログラムの書きやすさ	得意ワークロードの多さ	電力効率	性能
CPU (big)	◎	◎	×	○リソース次第
CPU (little)	◎	△	○	×
GPU	△	△	○	◎
DSP	○	△	◎	○
FPGA	×	○	◎	○
ASIC	ー	×	◎	◎

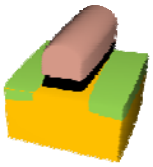
- 一つのプロセッサパッケージ内に
  複数個・複数種類のコアを実装

半導体プロセッサ成長の源泉：半導体微細化

- **ムーアの法則**
  - **「半導体の集積度は3年で4倍になる」**
  - 正確には**経験則/経営目標**
- 使える**リソース量の指数的増加**
- **Dennardスケーリング**
  - CMOSの嬉しい特性
  - 素子が小さくなると性能・効率共に向上



[M.Guarnieri, 2015]

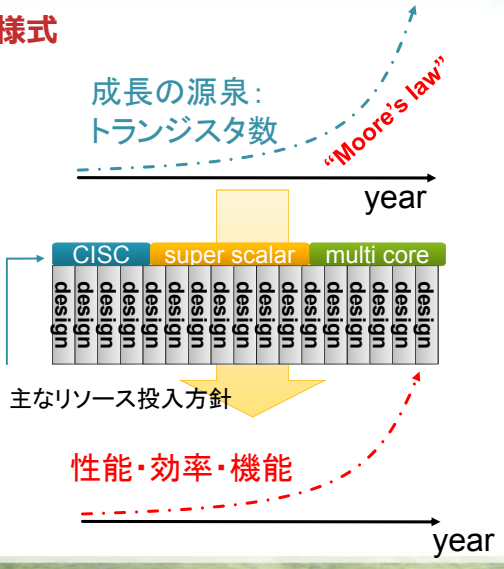


スケーリングファクター*K:*

デバイスの寸法	*1/k*
電圧	***1/k***
遅延	***1/k***
電力消費/デバイス	***$1/k^2$***
電力消費/面積	***1***

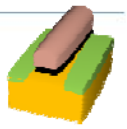
計算機「アーキテクチャ」による性能向上

- コンピュータの**構成法・様式**
  - コンピュータとは「**このように作るもの**」
  - プログラマ・OSから見た**コンピュータ仕様**
  - トランジスタ**リソースの使い方**の指針
    - 量を活かす
    - ワークロードへの合致のさせ方



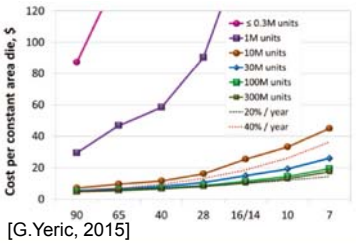
恩恵の減速：微細化と共に出てきた課題達

- **メモリ・ウォール**
  - チップ性能に見合った「内需」が必要
- **配線遅延**
  - RAMのポート数，容量と速度の両立が不可
  - チップ内の地理的影響
- **ホット・スポット**
  - 周波数の制限
- **リーク電力**
  - デナードスケーリングの破綻
- **プロセスばらつき**
  - 速度，電力面でのゲイン減少
- **ダーク・シリコン**
  - Tr稼働率に制限
- **デザイン・製造ハザード**
  - 最新プロセスに見合った効果が必要



デバイスの寸法 *1/k*

電圧	***1/k*** ⬅
遅延	***1/k***
電力消費/デバイス	***$1/k^2$***
電力消費/面積	***1***



[G.Yeric, 2015]

近年の成長源

- △トランジスタ数
  - Moore則の恩恵は後退しつつも**微細化世代進行は継続**
    - **コスト制約**
  - **三次元積層技術**の進展
    - 容量およびI/Oバンドの制約の緩和
    - 幾何学的に配線電力を緩和
- ×電力
  - Trの増加に**電力供給**が追いつかない
  - **電源確保に制限のある用途**の増加

アーキテクチャによる性能の上げ方

- **周波数上昇**の効果は飽和
  - 電力の制限
  - メモリのレイテンシ
- **並列性の利用，処理効率の向上**
  - なるべく多くの処理を同時に行う
  - 同じ仕事なら少ない処理で行う
- **値再利用**
  - 過去にやった計算結果を覚えておいて利用
- **近似計算**
  - 実用上問題ない範囲で精度を落として
    速度と効率を向上

ILP TLP DLP

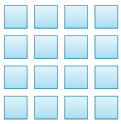
- **Instruction Level Parallelism**
  - CPU一つあたりにリソースをかけることで抽出
    - 時間的並列性：パイプライニング
    - 空間的並列性：スーパスカラ
  - スレッド実行そのものが高速化
- **Thread Level Parallelism**
  - コア数を増やすことで抽出
  - 複数スレッドで構成されたタスク実行が高速化
- **Data Level Parallelism**
  - SIMD, SIMTユニットの実行幅を増やすことで抽出
  - 流儀に添って書かれたプログラム部分の実行が高速化

Pollack's Rule あるいは√Trの法則

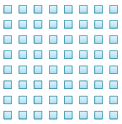
- プロセッサ成長に関する経験則
  - （ある方法論で得られる）
    性能はつぎ込んだリソースの平方根に比例する
    - 価格，トランジスタ，電力
- スーパリニア→リニア→飽和
  - リニア帯までで組み合わせた方がリソース効率は良い
    - **ILP** vs. **多ポートRAMの指数的コスト**
    - **TLP** vs. **Amdahl's Law**
    - **DLP** vs. **アルゴリズムの出現頻度**

法則の帰結： トレードオフの重層化

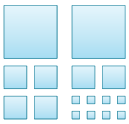
- できることは取り込まれていく
  - 「マルチコアかアクセラレータか？」
    ⇒リソースに合わせて**両方使う**，が基本
    - かつての「SMT vs. CMP」の議論同様



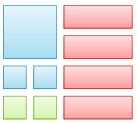
マルチコア



メニーコア



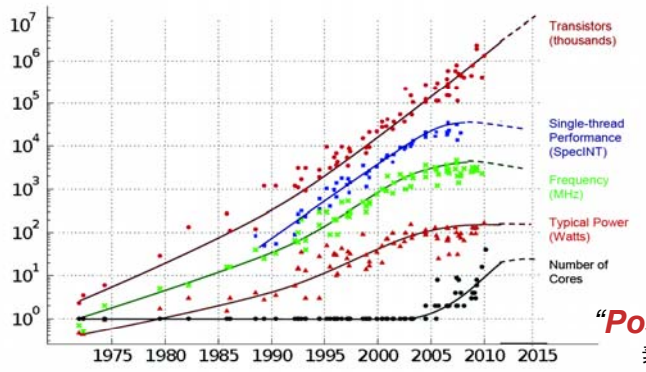
ビッグ・リトル
(ヘテロジニアス・マルチコア)



ヘテロジニアス・ISA

プロセッサ成長の傾向予測

35 YEARS OF MICROPROCESSOR TREND DATA

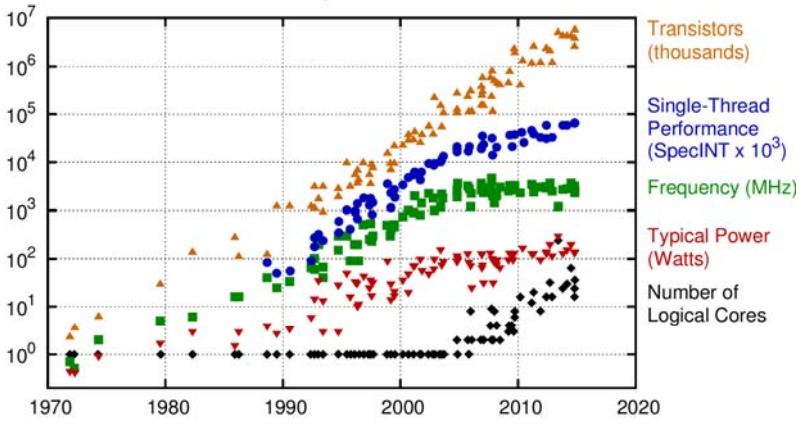


[M. Horowitz+] [C. Moore]

"Post Moore"

素子の進歩が止まるなら
カスタムプロセッサが有利
様々な見通しが
後退しているのは確か

40 Years of Microprocessor Trend Data



[M. Horowitz+] [C. Moore] [K.Rupp]

CPUコア研究の必要性

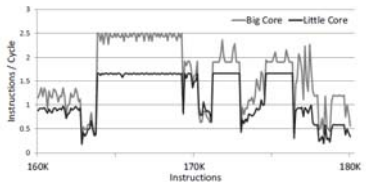
- カスタムプロセッサやアクセラレータの研究と両輪で必要
  - CPU側の基本アーキテクチャが変わらないまま取り残されている
- CPU処理能力向上の利点
  - 用途に対する**面積効率**
  - **クリティカルな処理時間**を縮める唯一の方法
  - **汎用性**
  - **新しいアーキテクチャ**の模索
- 需要：現に，シングルスレッド処理能力は**微増中**
  - DVFSの利用
  - 命令ウィンドウ幅の拡大
  - コンパイラ最適化

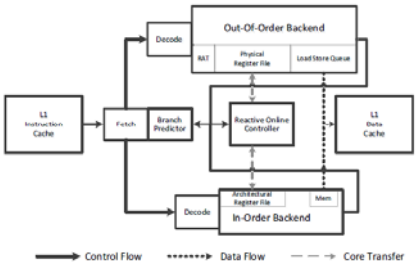
CPUコア最新研究の傾向

- ヘテロジーニアス構造を**1つのコアへ取り込み**動的に切り替え
  - どうせダークシリコンなので1つのコアにユニットを冗長にもって**最適なものを動かす**
- コア内融合の利点
  - **フェーズ**に応じて**こまめに良いとこ取り**が可能
  - **同時に動かさない**ので完全なコア冗長化が不要
    - 微妙に共有して**面積を節約**
- 例
  - **Composite Cores** [Lukefahr+, 2012]　CPU同士
  - **SEED** [Nowatzki+, 2015]　完全に異なるアーキテクチャ

Composite Cores [Lukefahr+, 2012]

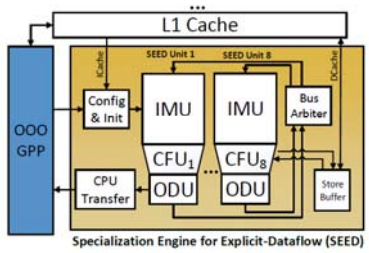
- in-orderとoooの良いとこ取りを狙う
  - 両方の機構を一つのコアに搭載
  - フェーズに合わせて動的に最適な方で実行
  - 同時に使わないので，共有できる部分は共有
- 実行される命令の種類から最適なコアを予測，切り替え
  - oooで性能が上がらないときに電力を節約

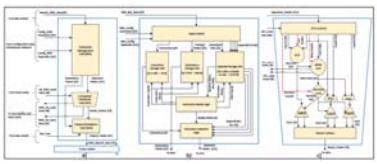




SEED [Nowatzki+, 2015]

- CPU と タイルアーキテクチャの良いとこ取りを狙う
- ループに特化してデータフローユニットを適用
  - コンパイラプリミティブ+ループ継続予測によって動的に切り替え
  - コードは両バージョンを用意
- SEED unit
  - ループのデータフローを FU networkへマップ
    - 制御やメモリ依存はトークンによってデータフロー化
  - 大量の中間値を保持





CPU研究の課題

- 「動的切り替えで**賢くユニットを選択して実行**」
  - 「最大でも既存のスーパスカラの性能を越えない」
    - または
  - 「コード傾向毎に実行ユニットとコードを用意」
- **トレードオフの曲線から抜け出すものではない**

- 組み合わせでなく，
  CPUの実行方式そのものを改良できないか？

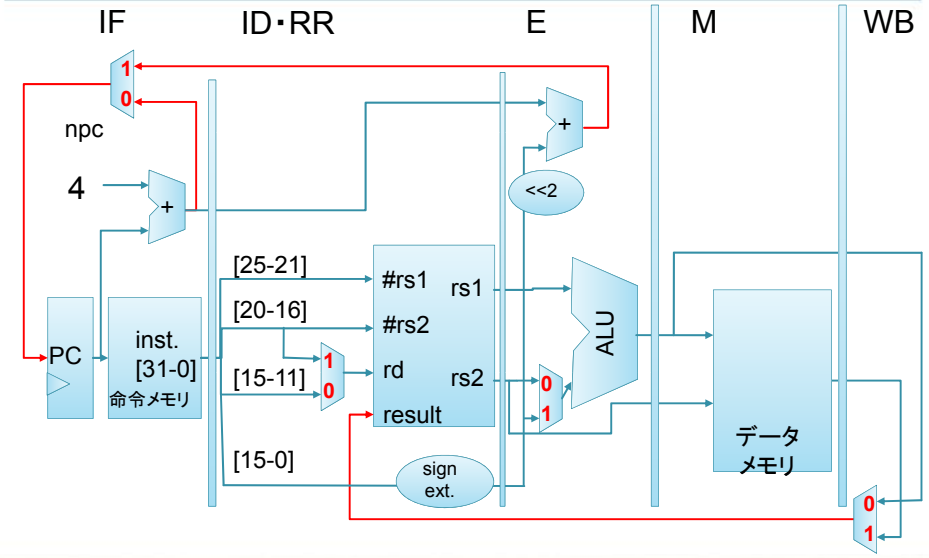
チャレンジ

- 「**CPU**の」
  - ボトルネックを解決しないとコア数・種類を増やせない
  - 汎用性が高くないと新しいプロセスを使えない
- 「**シングルスレッド実行**の」
  - 並列性が高い部分はDLPやTLP技術の方が効率的
- 「**効率と性能双方**を向上できる」
  - 効率だけ上げてもシングルスレッドは速くならない
  - 勿論「電力じゃぶじゃぶ」で性能上げてもダメ
- 「**拡張可能**な**リソースのつぎ込み方**」
  - 同じ戦略でプロセス何世代か分は性能が上がって欲しい

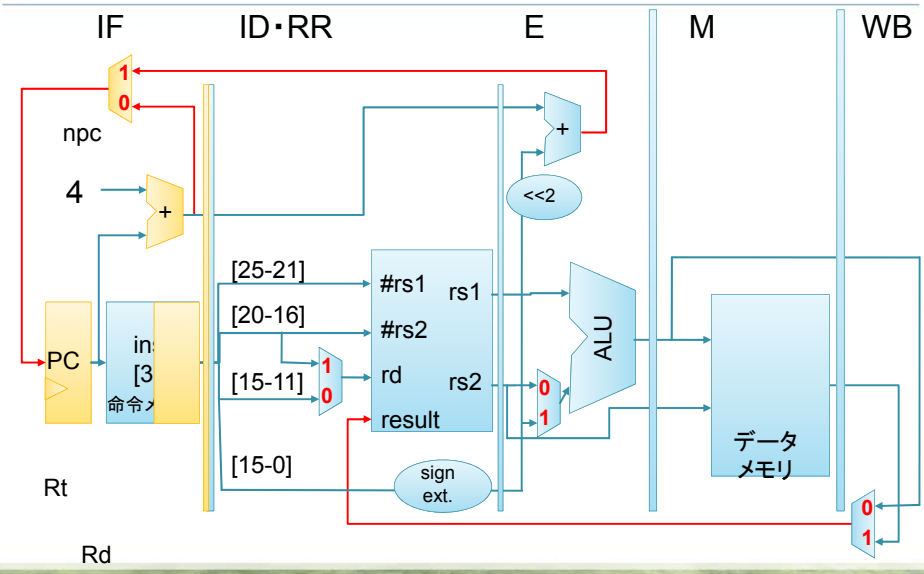
CPUマイクロアーキテクチャによる性能向上技術

2016/11/28

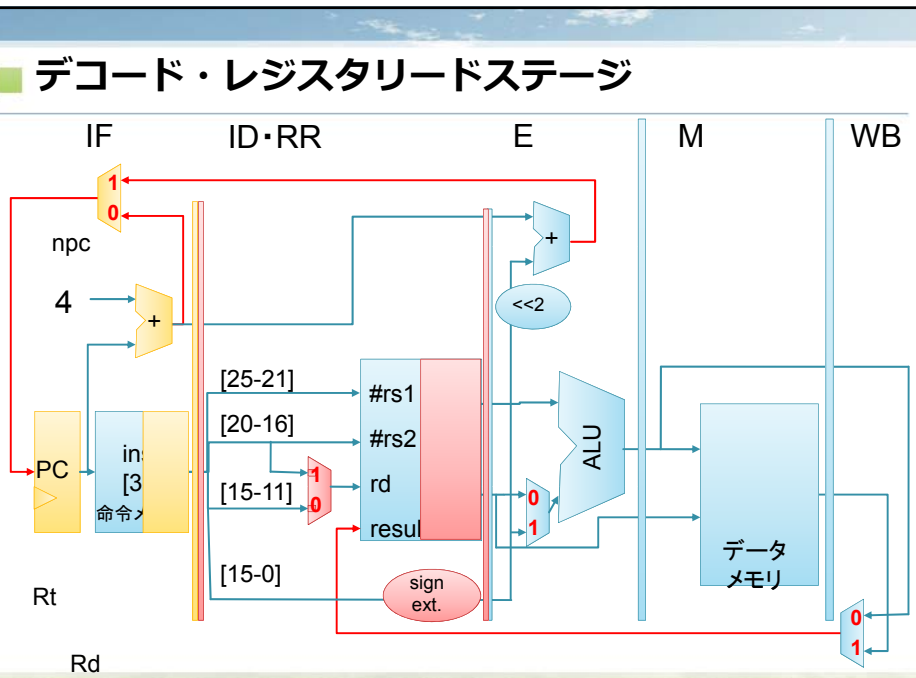
おさらい：スカラ・プロセッサ



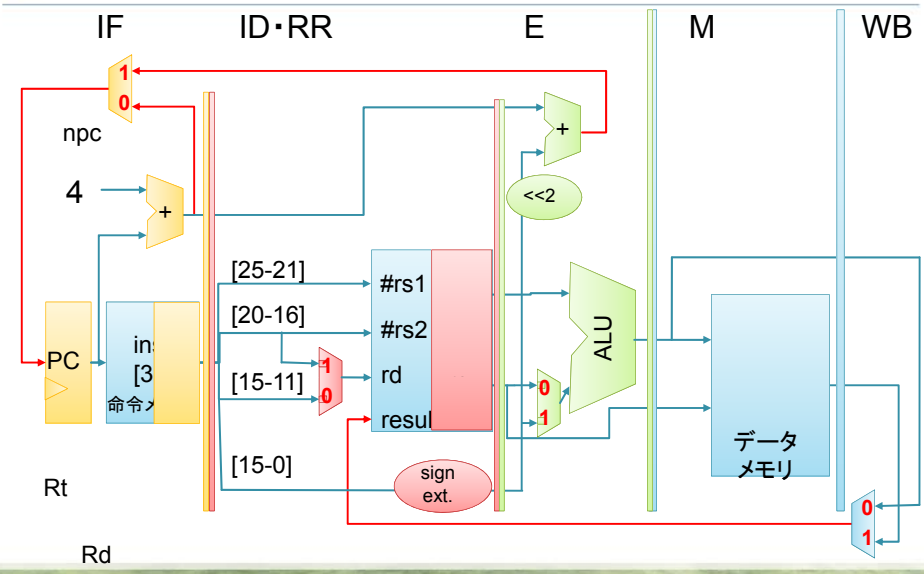
フェッチステージ



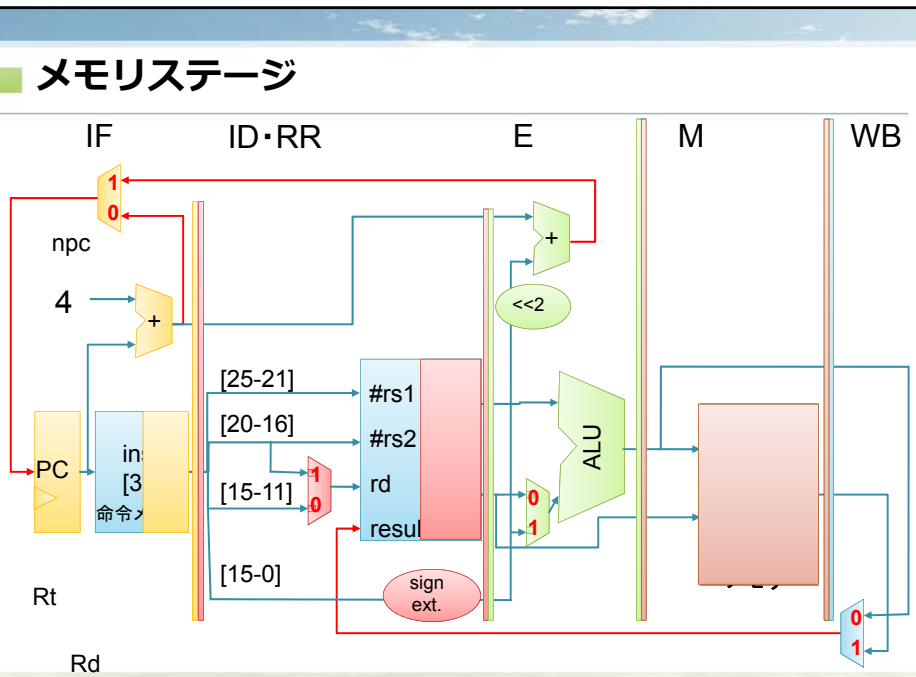
デコード・レジスタリードステージ



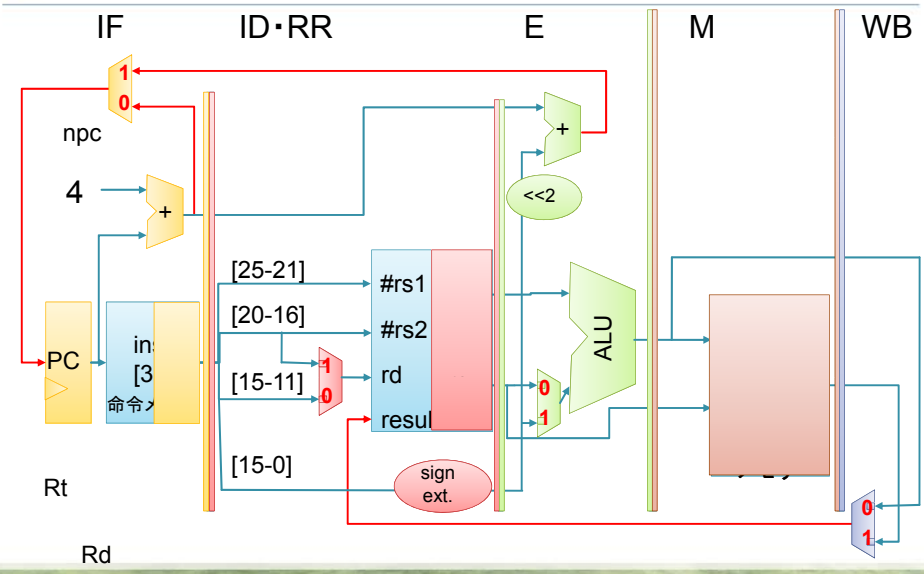
実行ステージ



メモリステージ



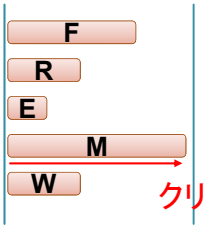
ライトバックステージ



スループットとレイテンシ

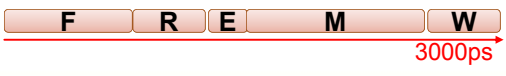
- 例えば各ステージの遅延が以下の場合

フェッチ: 800ps
レジスタリード: 400ps
エクセキュート: 200ps
メモリ : 1200ps
レジスタライト: 400ps



クリティカル・パス

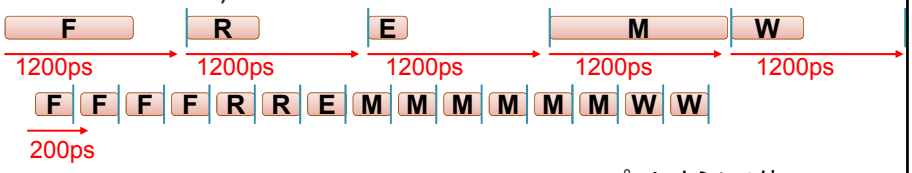
動作クロックはクリティカルパスが決定



オーバヘッドなど考慮しなければ
レイテンシ2倍
スループット2.5倍 : 性能向上

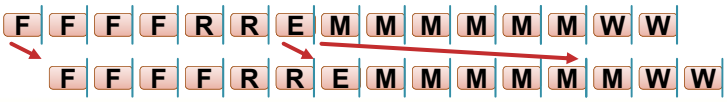
スーパ・パイプラインとクリティカルループ

- 先ほどの例は，各ステージを分割すれば…



スループット さらに6倍

- 「クリティカル・ループ」
  - 1サイクルでできないと命令間で依存があるときに
    バブルの影響が大きい処理
    - ≒クリティカル・パスを定める部分
  - スカラの場合の典型的なポイント



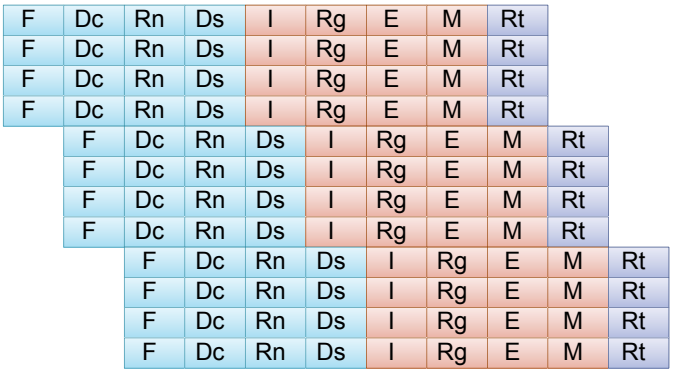
スーパ・パイプライン拡張のオーバヘッド

- **リソース効率**を落としてくる要素
  - **電力**　fV^{2}に比例，＋パイプラインレジスタ増加分
  - **クロッキングオーバヘッド**
    - スキュー，ジッター，ラッチオーバヘッド
  - **バブル**
    - スカラの場合特にメモリ

- 1stageの限界は**6-8FO4** [Hishikesh+, 2002]
  - Prescottの**31段**をピークに**20段**ほどまで減少

スーパ・スカラ

- 1サイクルに複数命令を同時処理
  - 性能ポテンシャルは　深さ×**幅**　で決定
    - **Cycle Per Instruction** から**IPC**へ



イン・オーダとアウト・オブ・オーダ

- **イン・オーダ**実行
  - コードの順番通りに実行
- **アウト・オブ・オーダ**実行(**ooo**)
  - 実行結果が不変の範囲で**処理の順番を変更**
  - 一度**スケジューラ**に入れ，
    実行可能なものをバックエンド・パイプラインへ発行
    - キャッシュやメモリのレイテンシや
      浮動小数点演算の依存による**バブル解消に効果的**
      - **スーパスカラの幅**を活かすためにはほぼ**必須**
    - それなりに複雑な**スケジュール制御**と**ステート保持制御**機構を導入

o-o-oの実現方式

- **Tomasulo**のアルゴリズム
  - レジスタ番号を内部的に読み替えて偽依存を解消
  - 依存が解消するまで待機するスケジューラの導入
- **リオーダ・バッファ**方式(論理レジスタの実体あり)
  - インフライトな値を仮に保存，コミット時にレジスタに書き戻し
  - レジスタ容量を節約できる
  - 検索が複雑だがリカバリは楽
- **物理レジスタ**方式（論理レジスタの実体なし）
  - 論理→物理の対応表のみ保持
  - リカバリにはステートの再構築が必要だが，検索は楽
  - スケジュールとデータのパスを切り離すことができる

実行結果を変えないためには？

- どういう順序換えが許される？

```
LD    $1, 0($2)
ADD   $4, $1, $3
ADD   $5, $6, $7
ADD   $8, $5, $7
ADD   $3, $1, $1
ADD   $5, $4, $3
```

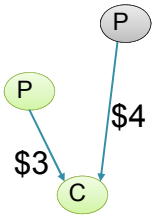
```
FADD $3, $1, $2
FADD $4, $1, $3
FADD $7, $5, $6
FADD $8, $5, $7
FADD $2, $1, $2
```

命令間依存関係

- ある命令（プロデューサ）が値を生成し，後続のある命令（コンシューマ）が値を使う
  - レジスタまたはメモリを介して値の受け渡し
- 命令の実行順を変えるときには依存関係に注意
  - とくに，値の更新に注意

レジスタ依存

P ADD $3, $2, $1
C ADD $5, $4, $3



メモリ依存

P ST $5, 0($2)
C LD $1, 0($2)

依存関係の種類

- 2命令間の関係にブレイクダウン

Read After Read

ADD $2, $1, imm
ADD $3, $1, imm

基本的に問題とならない．
マルチプロセスなどでリードにステート更新の副作用がつくときには問題となる．

Write After Read（逆依存）

ADD $2, $1, imm
ADD $1, $3, imm

前の命令で使用する値を後ろの命令が上書きする．順序を入れ替えると前の命令が無効な値を読み出す．

Read After Write（真の依存）

ADD $2, $1, imm
ADD $3, $2, imm

前の命令の生成値を後ろの命令が使用する．順序を入れ替えるとコンシューマが無効な値を読み出す．

Write After Write（出力依存）

ADD $2, $1, imm
ADD $2, $3, imm

前の命令の生成値を後ろの命令が上書きする．順序を入れ替えると残すべき出力値が上書きされる．

偽依存

- データ生成ではなく資源共有を原因とする依存

Read After Read

```
ADD  $2, $1, imm
ADD  $3, $1, imm
```

基本的に問題とならない.
マルチプロセスなどでリードにステート更新の副作用がつくときには問題となる.

Write After Read（逆依存）

```
ADD  $2, $1, imm
ADD  $1, $3, imm
```

前の命令で使用する値を後ろの命令が上書きする. 順序を入れ替えると前の命令が無効な値を読み出す.

Read After Write（真の依存）

```
ADD  $2, $1, imm
ADD  $3, $2, imm
```

前の命令の生成値を後ろの命令が使用する. 順序を入れ替えるとコンシューマが無効な値を読み出す.

Write After Write（出力依存）

```
ADD  $2, $1, imm
ADD  $2, $3, imm
```

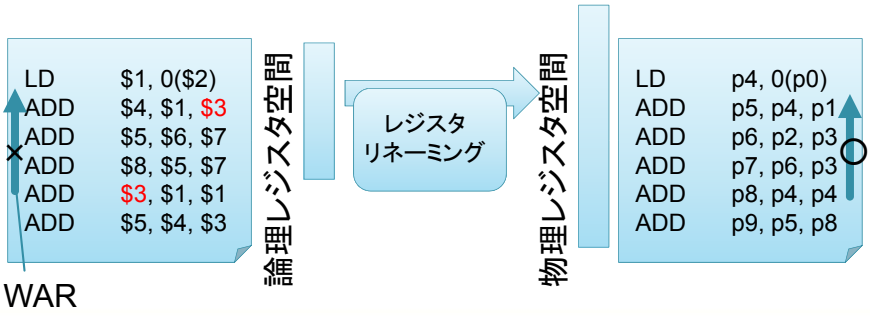
前の命令の生成値を後ろの命令が上書きする. 順序を入れ替えると残すべき出力値が上書きされる.

アウト・オブ・オーダ実行の基本方針

- RAR
  - 対処しないでOK
- RAW
  - 違反しないようにソースがreadyであることを監視
  - スケジューラによる発行(issue)制御
- WAW, WAR
  - 資源を投入して上書きを排すれば解決
  - レジスタ・リネーミング

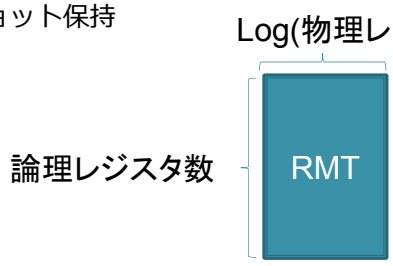
レジスタ・リネーミング

- コードに書かれている論理レジスタ番号を，実際に使われる物理レジスタ番号へ読み替える
  - 上書きのないコードへ変換⇒偽依存を解消
  - 多量のレジスタを実装してもオペランドフィールドを圧迫しない



ソースレジスタの決定ロジック

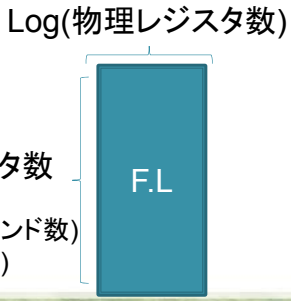
- デコード後に物理レジスタへ読み替え
- Register Map Table
  - 論理レジスタ番号でひくと対応する物理レジスタ番号が得られる
    - RAMで構成
    - 命令毎にスナップショット保持



Rポート数＝フロントエンド幅×(ソースオペランド数)
+フロントエンド幅×(ディスティネーションオペランド数)
Wポート数=フロントエンド幅×(ディスティネーションオペランド数)

ディスティネーションレジスタの決定ロジック

- まだ使ってないレジスタを割り当てる
  - フリーリスト：
    上書きしても大丈夫なレジスタのリスト
- フリーリストから一つ取り出す
  - 該当命令のディスティネーションとして読み替え
  - 該当論理レジスタについてRMTに登録

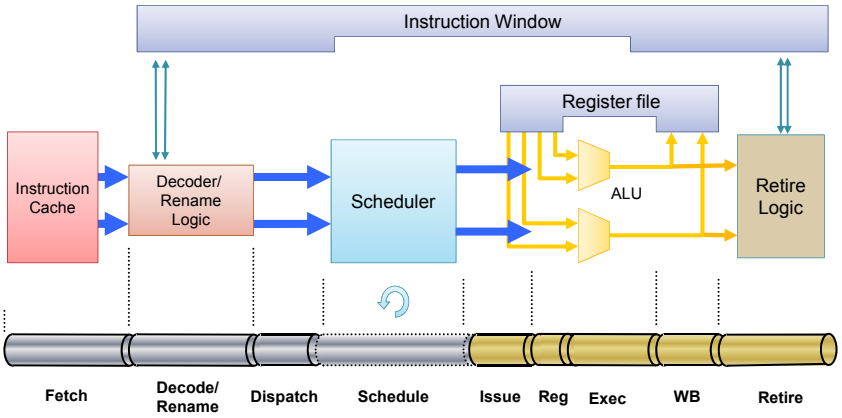


Rポート数＝フロントエンド幅×(ディスティネーションオペランド数)
Wポート数=リタイア幅×(ディスティネーションオペランド数)

フリーリストの管理

- フリーリストに入るレジスタとは？
  - 上書きしても良いレジスタ
    ⇒もうどの命令も参照しないレジスタ
    ⇒コード上で上書きされたレジスタ
- ディスティネーションレジスタでRMTを引くと
  - その命令が上書きする論理レジスタに対応する
    物理レジスタが分かる
  - この命令が実行されれば，その物理レジスタはフリー
- 命令の完了を待ってフリーリストへ追加

アウト・オブ・オーダ・スーパスカラ・プロセッサ

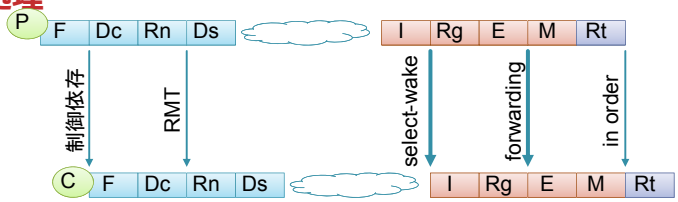


スーパスカラ・プロセッサの利点

- **データ依存**や**制御依存**が**単純でない**プログラムを最も高速(効率的)に実行できるアーキテクチャ
  - 命令単位の制御依存，データ依存に対応
  - データフロー制約をハードウェアが解析して ハードウェア毎に適したILP実行
- 現状ほぼ**唯一のオプション**

スーパスカラのクリティカル・ループ

- 1サイクルで終わらせないといけない処理
- 問題となるのは
  - **制御依存**
  - **レジスタ・リネーミング**
  - **発行ループ**
  - **データフォワーディング**
  - **リタイア処理**



スーパスカラの周辺技術

- **予測**によってパイプラインを埋める
  - 依存を切る，クリティカル・ループを隠蔽する
  - 元々リタイアするまで投機状態なので導入が容易
    - **分岐予測**，**アドレス予測**，**依存予測**，**値予測**，**キャッシュヒット/ミス予測**，**レジスタレイテンシ予測**…
- **クラスタ化**
  - 大きいテーブルを分割して**クリティカル・パス平滑化**
  - 局所性を利用してペナルティ削減
    - レイテンシが動的になるので投機実行と併用
- 導入の目安
  - その**コスト**で**キャッシュ/コア**を増やしたときよりも**効果的か？**

スーパスカラの隘路

- リソースをどう使って性能を上げるか？
- **実行幅**を増やす？（性能ポテンシャルの向上）
  - フォワード，スケジューラ，RMTのクリティカルパスが増加
- **命令ウィンドウ**を増やす？（抽出可能なILPの向上）
  - 管理しなければいけない情報が増加
  - 実行に直接関係ない電力の増加
- **投機**をはげしくする？(リソース稼働率の向上)
  - 投機失敗による無駄なエネルギーの増加
- 一方でRISCバイナリの**ILP上限**
  - プログラムのデータフローには沢山あるはず
  - 「まだ90倍は並列性がある」[NICOLAU+, 1984]
    - だが，実際にパイプラインに入れてみると**IPCは3〜5**など
- **節約は簡単**だが性能向上させるには慎重な設計が必要

スーパスカラを越えようとしたアーキテクチャ

- 制御・依存解消を**ブロック化**して効率化
  - 静的解析の限界
- 演算を組み合わせて**制御あたりの演算量**を増加
  - アクセラレータへ
- 大きいユニットの**分散化**，処理の**局所化**
  - ILPの収穫逓減
- **投機マルチスレッディング**
  - マルチコアへ
- **スーパスカラ**も地道に性能向上
  - リソースの使われ方が洗練

今，一から作るならこんなプロセッサ

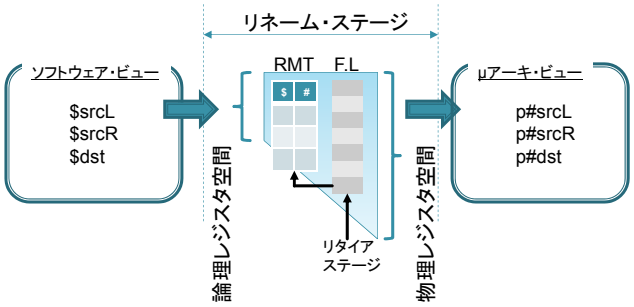
2016/11/28

制約条件と，使える道具

- 制約条件
  - CPUなので…
    - **命令単位**の**データ依存**と**制御依存**
    - **ooo**できなければ性能が出ない
  - **1命令あたりに必要な電力**は極力減らす
- 使えるリソース
  - **Tr容量**　ただしあまりスイッチングしないテーブル等
  - **コンパイラ**，**命令セット**による支援
    - ヘテロISAが受け入れられるなど，以前より制約緩和
- 方針
  - **アクセス頻度の低いテーブル容量で制御軽減を買う**
  - **最大実行幅ではなくコンスタントなILP抽出**

管理削減のターゲット

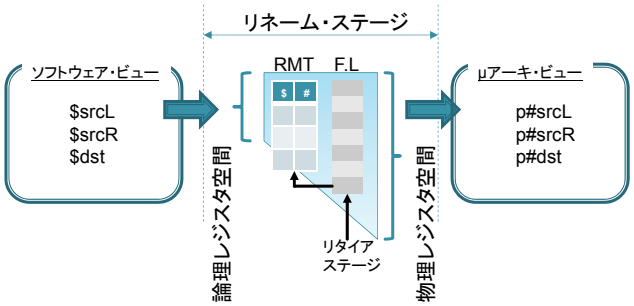
[従来の物理レジスタ方式によるo-o-o準備]



○パイプライン実行から**偽依存**(レジスタ上書き)を排除 ⇒ **o-o-o性能向上**
○物理レジスタをソフトウェアから隠蔽して**互換性保持**
○**高価な物理レジスタ**の利用状況を**細かく管理して節約**
×実行に直接関与しない**電力増**

効率化の鍵

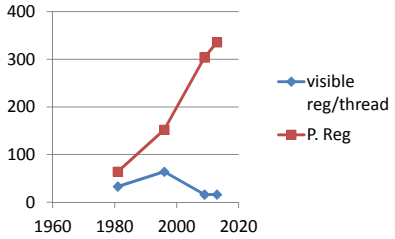
[従来の物理レジスタ方式によるo-o-o準備]



○パイプライン実行から**偽依存**(レジスタ上書き)を排除 ⇒ **o-o-o性能向上**
○物理レジスタをソフトウェアから隠蔽して**互換性保持**
○**高価な物理レジスタ**の利用状況を**細かく管理して節約**
×実行に直接関与しない**電力増**

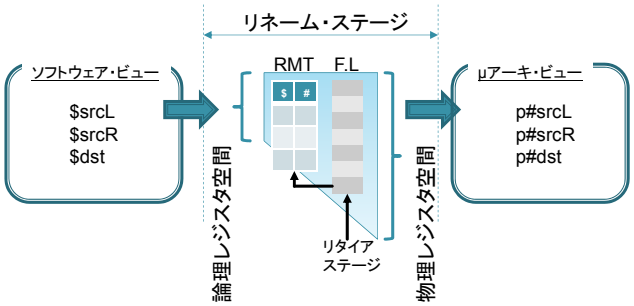
ソフトウェア・ビジブルなレジスタ数

- CPUの物理レジスタ増に対し，論理レジスタ数は不変のまま
  - GPUだと数万がソフトウェア・ビジブル
- ソフトウェアからレジスタ数を隠蔽しておくことが最良か？
  - 解放すれば使いようがあるはず



効率化の鍵

[従来の物理レジスタ方式によるo-o-o準備]



○パイプライン実行から**偽依存**（レジスタ上書き）を排除 ⇒ **o-o-o性能向上**
×**物理レジスタをソフトウェアから隠蔽**して△互換性保持
○**高価な物理レジスタ**の利用状況を**細かく管理して節約**
×実行に直接関与しない**電力増**

「高価なレジスタ」

- レジスタは増やせないのか？
  - ◎容量
    - ポート数大だから高いが使えるTrも増えている
  - ○レイテンシ
    - 大きくすると遅くなるが，レジスタはクリティカルループ外
  - ○熱
    - 負荷が分散する分冷える
      - レジスタを倍にしても性能は倍にならないはず
  - △電力
    - アクティブ電力は熱と同様
      - リークはデバイスとゲーティングで対策
- 問題は管理

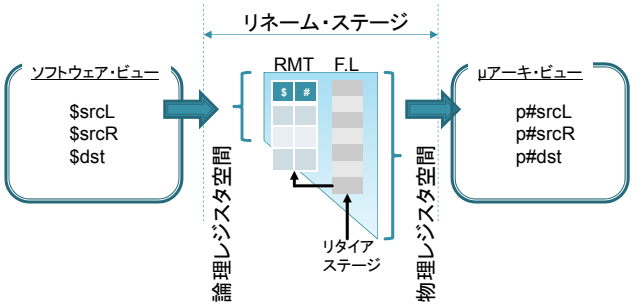
レジスタの管理コスト

- RMT(クリティカルループ内)
  - [**論理レジスタ数**] x [log(**物理レジスタ数**)] bit の**RAM**
  - ↑を毎サイクル[3r1w] x [フロントエンド幅]回アクセス
- フリー・リスト(クリティカルループ内)
  - [**物理レジスタ数**] x [log(**物理レジスタ数**)] bit の**FIFO**
  - ↑を毎サイクル[フロントエンド幅]だけpop
    [リタイア幅]だけpush
- 命令ウィンドウ(ストール判定ループ内)
  - [log(**物理レジスタ数**)]をキーとしてアクセスできる
    [命令ウィンドウ幅] x [log(**物理レジスタ**)] bit の**CAM**
  - ↑を毎サイクル[リタイア幅]r[フロントエンド幅+リタイア幅]w回アクセス

レジスタ**節約のための管理機構**:
複数の多ポートテーブルを**クリティカルループ**に挿入し，**拡張性を阻害**

効率化の鍵

[従来の物理レジスタ方式によるo-o-o準備]



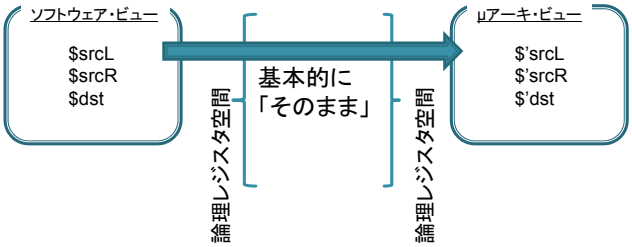
○パイプライン実行から偽依存(レジスタ上書き)を排除 ⇒ o-o-o性能向上
×物理レジスタをソフトウェアから隠蔽して△互換性保持
×管理コストが高価な物理レジスタを細かく管理して節約
×実行に直接関与しない電力増

Tr増，電力制限の下では，レジスタ節約管理は却って逆効果なのでは？

新実行方式の構想

[リネームレス・アウト・オブ・オーダ実行の構想]

•物理レジスタを潤沢に用意，ソフトウェア側にも広大なレジスタ空間を見せる
•レジスタは単なるKVSとして振る舞い，一切の管理をしない



？レジスタ数を利用して偽依存のないコードを準備
○レジスタ数をソフトウェアに見せて新しい最適化を導入
○レジスタは管理不要，コードの通りに読んで書きっぱなしでOK?
○レジスタ管理に関する電力削除，代わりに稼働率の下がったレジスタを並べる

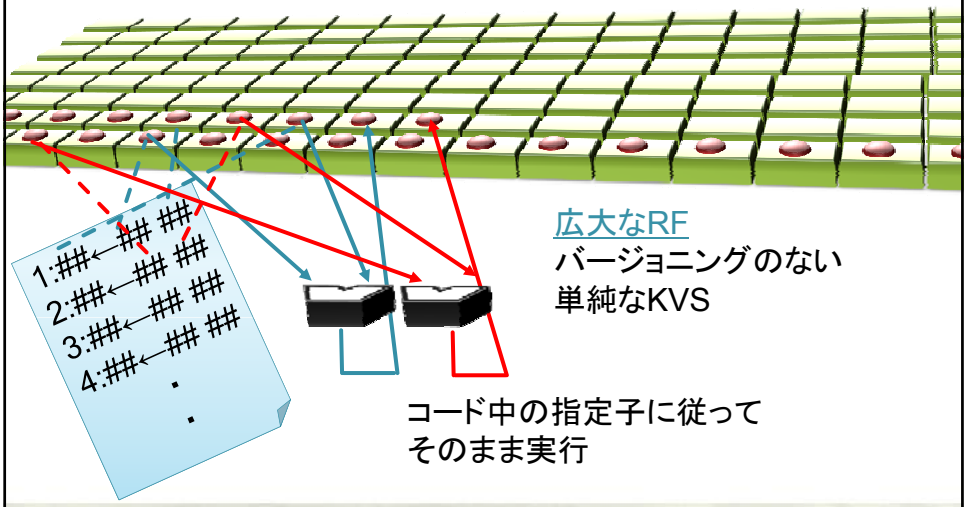
課題

容量と制御を交換するアーキテクチャの構想

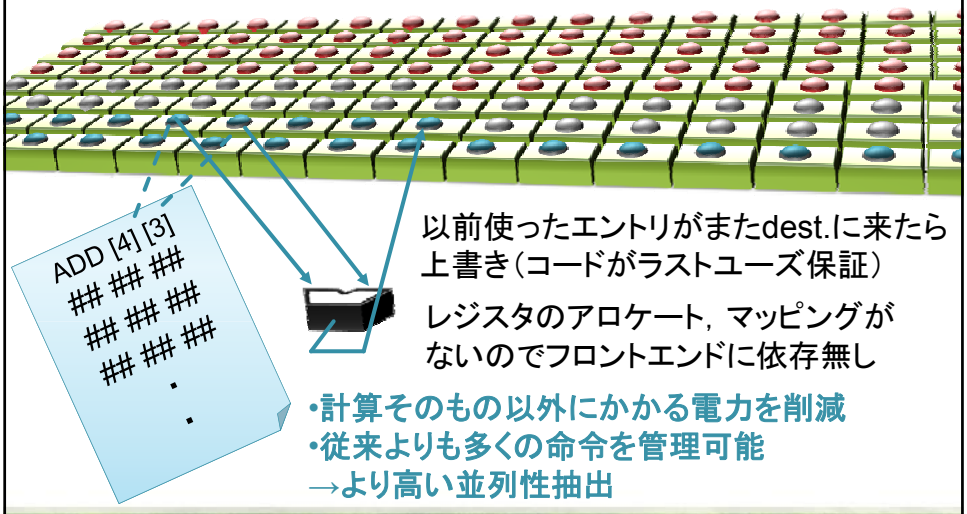
1. 広大なレジスタ空間でライトワンスコードを実行
   - ⇨ 偽依存がないのでリネーミング不要
     コードのレジスタ番号でそのままo-o-o並列実行
2. 豊富なレジスタ数を背景にコード側で寿命を保証
   - ⇨ 開放管理不要　コードのままに上書きして再利用
3. ↑の軽量大容量RFで大きな命令ウィンドウ実現
   - ⇨ レジスタマッピング不要，フリーリスト不要
     （従来の困難点排除，スケジューラはMat）
   - ⇨ 発行幅を増やさずにIPC向上

提案アーキテクチャのイメージ

トランジスタ増をレジスタ本数増に使う



提案アーキテクチャのイメージ



以前使ったエントリがまたdest.に来たら
上書き（コードがラストユーズ保証）

レジスタのアロケート，マッピングが
ないのでフロントエンドに依存無し

- 計算そのもの以外にかかる電力を削減
- 従来よりも多くの命令を管理可能

→より高い並列性抽出

STRAIGHTプロセッサ

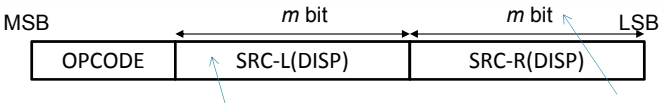
2016/11/28

STRAIGHTプロセッサ

- SSのリソース管理を見直したCPUアーキテクチャ
  - 高性能，高効率，拡張性
    - **ライト・ワンスコード**を静的に用意
    - **リネームレス・アウト・オブ・オーダ**実行
    - ソフトウェアから見えるレジスタの拡張性
- 以下順番に紹介
  - **命令セットアーキテクチャ**
  - **コンパイラ**
  - **マイクロアーキテクチャ**

命令形式とコード

命令形式



「k命令前の実行結果」2^m命令前まで参照可能
＝「以降参照されない」

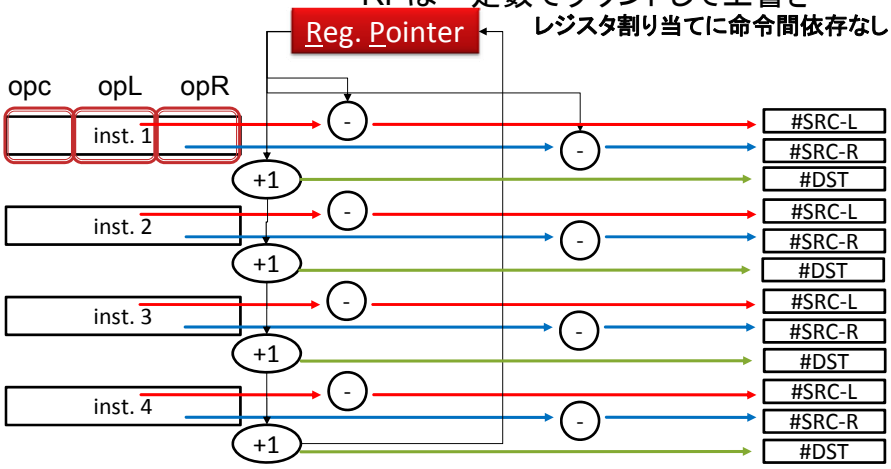
コード例

```
s = a + b + c + d;
```

```
LD [10]      #ld a
LD [10]      #ld b
LD [10]      #ld c
LD [10]      #ld d
ADD [4] [3]  # s = a + b
ADD [1] [3]  # s = s + c
ADD [1] [3]  # s = s + d
ST [1] [10]  #st s
```

並列化が容易なフロントエンド

「RP」を基準にレジスタ番号決定
RPは一定数でラウンドして上書き
レジスタ割り当てに命令間依存なし



STRAIGHT命令セット ver1.0

制御命令

J, JR, JAL, BEZ, BNZ, FPBEZ, FPBNZ, RPINC

転送命令

SPADD(i), FPADD(i), LD, ST, FPLD, FPST, SPLD, SPST,

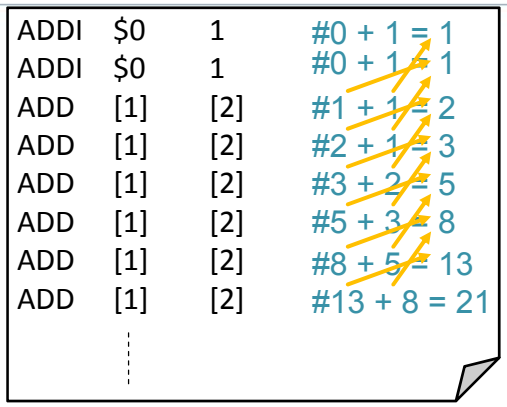
演算命令

(F)ADD(i), (F)SUB(i), (F)MUL(i), (F)DIV(i), FTOI, ITOF, SHL(i), SHR(i), AND(i), OR(i), XOR(i), SLT, RMOV

システム命令

SYS, HALT

このコードは？



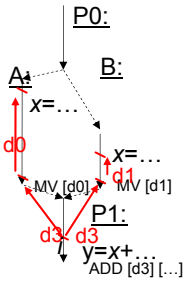
- 新しい最適化の工夫の余地も
  - ソフトウェア・ビジブルな大量の論理レジスタ
  - レジスタでもインデックス的なアクセスが可能

STRAIGHTコンパイラの特徴

- **ソースレジスタを命令距離で指定**するようなコードを出力する
  - 他の専用プロセッサ向けコンパイラとは異なる新しいチャレンジ
- レジスタ・カラーリング前の**Single Static Assignment** 形式の中間言語を利用
  - ちょうどLLVMがこの形式を利用
  - **LLVM-IR**からのポスト・コンパイラの形で実装

コンパイラのチャレンジ

- **制御の合流**時に調整が必要
  - 逆デュアルフロー表現の課題
  - NOPでパス長統一
  - 同じ順番で「**積む**」
- レジスタ寿命
  - 参照距離の上限を越える場合には中継
    - あるいはメモリ
- セーフネット：
  上書き可能なスタック・ポインタ・レジスタ
  - どんなプログラムでも記述できることを保証



コンパイルの流れ

- 中間表現の読み込み
- OPCODEをSTRAIGHTに
- CFGの頭からコンパイル
  - 分岐と合流の情報が含まれている
- 合流するBBにfixed領域を生成
  - 末尾にfixed領域を生成
    - 順番に必要な変数を並べる
- ソース・レジスタの生成
  - 参照される行 - 代入される行

サンプルコード

```
int add(int x, int y){
        if (x > 0)
        {
                x++;
        } else {
                x = x + 2;
        }
        int a = x + y;
        return a;
}
```

```
int main(int argc, char *argv[]) {
        int x = (int)argv[0];
        int y = (int)argv[1];
        add(x, y);
        return 0;
}
```


LLVM-IR

```
(前略)
define i32 @add(i32 %x, i32 %y) nounwind {
entry:
        %cmp = icmp sgt i32 %x, 0
        br i1 %cmp, label %if.then, label %if.else
if.then:                                          ; preds = %entry
        %inc = add nsw i32 %x, 1
        br label %if.end
if.else:                                          ; preds = %entry
        %add = add nsw i32 %x, 2
        br label %if.end
if.end:                                           ; preds = %if.else, %if.then
        %x.addr.0 = phi i32 [ %inc, %if.then ], [ %add, %if.else ]
        %add1 = add nsw i32 %x.addr.0, %y
        ret i32 %add1
}
（後略）
```

OPCODEの変換とCFGの把握

```
Function : @add
BasicBlock : entry fixed : 0 *linked
0       ADDi    $0          0                   S:()        %cmp.i
1       SLT     [%cmp.i]    [%x]                S:(i32)     %cmp
2       BEZ     [%cmp]      if.else             S:()

BasicBlock : if.then fixed : 1
3       ADDi    [%x]        1                   S:(i32)     %inc
4       J       if.end                          S:()

BasicBlock : if.else fixed : 0
5       ADDi    [%x]        2                   S:(i32)     %add
        BR
BasicBlock : if.end fixed : 0
6       PHI     %inc        if.then             S:(i32)     %x.addr.0
7       PHI     %add        if.else             S:(i32)     %x.addr.0
8       ADD     [%x.addr.0]         [%y]                    S:(i32)     %add1
9       ADD     $0          [%add1]             S:()
10      JR      [%%]                            S:(i32)
```


パスによって定義の異なる変数の位置調整

```
Function : @add
BasicBlock : entry fixed : 0 *linked
0       ADDi    $0          0                   S:()        %cmp.i
1       SLT     [%cmp.i]    [%x]                S:(i32)     %cmp
2       BEZ     [%cmp]      if.else             S:()

BasicBlock : if.then fixed : 2
3       ADDi    [%x]        1                   S:(i32)     %inc
4       J       if.end                          S:()

BasicBlock : if.else fixed : 2
5       ADDi    [%x]        2                   S:(i32)     %add
6       UNDEF                                   S:()

BasicBlock : if.end fixed : 0
7       PHID    2                               S:(i32)     %x.addr.0
8       ADD     [%x.addr.0]         [%y]                    S:(i32)     %add1
9       ADD     $0          [%add1]             S:()
10      JR      [%%]                            S:(i32)
```

+1してmaxで均す

等距離

スタックによる値の退避と復帰

```
BasicBlock : entry fixed : 0 *linked
0       SPADDi -16                                      S:()
1       SPST   [%%]    0                                S:()
2       SPST   [%y]    8                                S:()
3       ADDi   $0      0                                S:()        %cmp.i
4       SLT    [%cmp.i][%x]                             S:(i32)     %cmp
5       BEZ    [%cmp]  if.else                          S:()
BasicBlock : if.then fixed : 2
6       ADDi   [%x]    1                                S:(i32)     %inc
7       J      if.end                                   S:()
BasicBlock : if.else fixed : 2
8       ADDi   [%x]    2                                S:(i32)     %add
9       UNDEF                                           S:()
BasicBlock : if.end fixed : 0
10      PHID   2                                        S:(i32)     %x.addr.0
11      SPLD   8                                        S:()        %y
12      ADD    [%x.addr.0]      [%y]                    S:(i32)     %add1
13      SPLD   0                                        S:()        %%
14      SPADDi 16                                       S:()
15      ADD    $0      [%add1]                          S:()
16      JR     [%%]                                     S:(i32)
```

分岐前に定義され
合流後に使いたい値
(工夫すれば最適化可能)

オペランドを距離に変換

```
BasicBlock : entry fixed : 0 *linked
0       SPADDi -16                                      S:()
1       SPST   [2]     0                                S:()
2       SPST   [4]     8                                S:()
3       ADDi   [0]     0                                S:()        %cmp.i
4       SLT    [1]     [7]                              S:(i32)     %cmp
5       BEZ    [1]     2                                S:()
BasicBlock : if.then fixed : 2
6       ADDi   [9]     1                                S:(i32)     %inc
7       J      3                                        S:()
BasicBlock : if.else fixed : 2
8       ADDi   [9]     2                                S:(i32)     %add
9       UNDEF                                           S:()
BasicBlock : if.end fixed : 0
10      SPLD   8                                        S:()        %y
11      ADD    [3]     [1]                              S:(i32)     %add1
12      SPLD   0                                        S:()        %%
13      SPADDi 16                                       S:()
12      ADD    [0]     [3]                              S:()
14      JR     [3]                                      S:(i32)
```

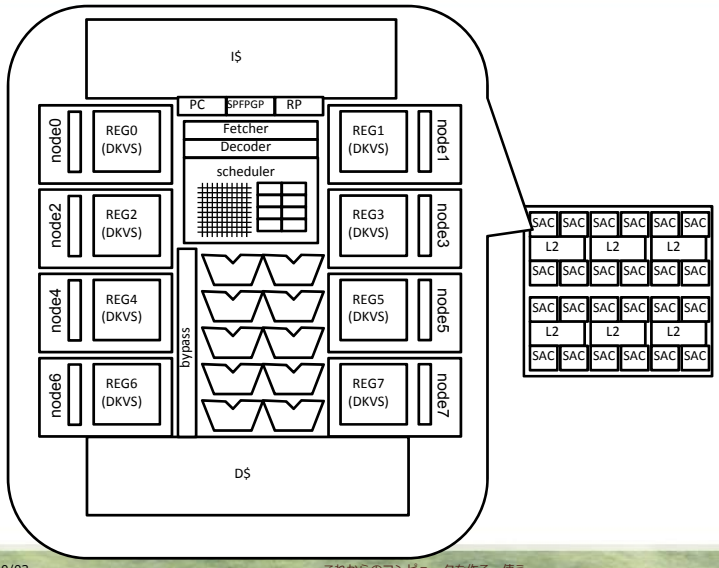
得られる命令コードの特性

- ライト・ワンス制約，寿命制約による
  メモリ・アクセス増は限定的

- メモリ・アクセスを減らす最適化が可能に
  - 広いレジスタ空間を直接コンパイラから操作可能
    - **インデックス的なアクセス**が可能
  - **スピル・イン**，**スピル・アウト**の削減
  - 関数呼び出し時の**save量の削減**
  - **データ構造**のレジスタ・プロモーション
  - 論理レジスタ64本程度でも効果
    - 巨大なレジスタ空間を保証する必要はなし

STRAIGHTマイクロアーキテクチャ

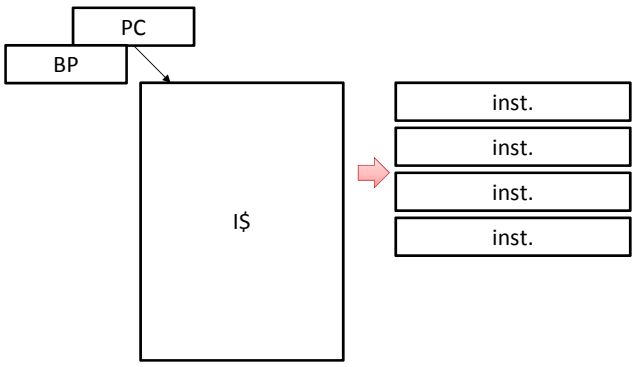
[CMPP,2012][ARC206,2013]



パイプライン構成と実行イメージ

F D S RR E M RW

フェッチ　従来アーキテクチャと同様

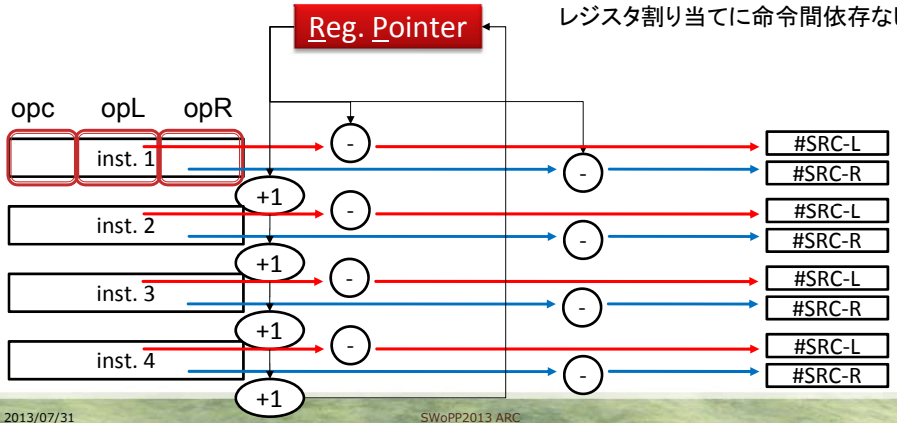


パイプライン構成と実行イメージ

F D S RR E M RW

デコード　「RP」を基準にレジスタ番号決定

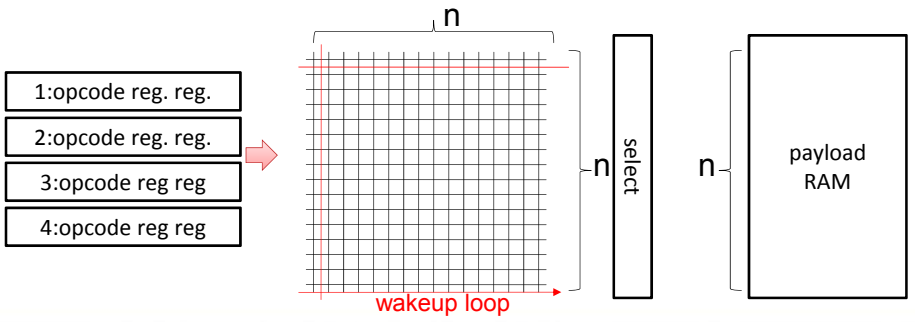
レジスタ割り当てに命令間依存なし



パイプライン構成と実行イメージ

F D S RR E M RW

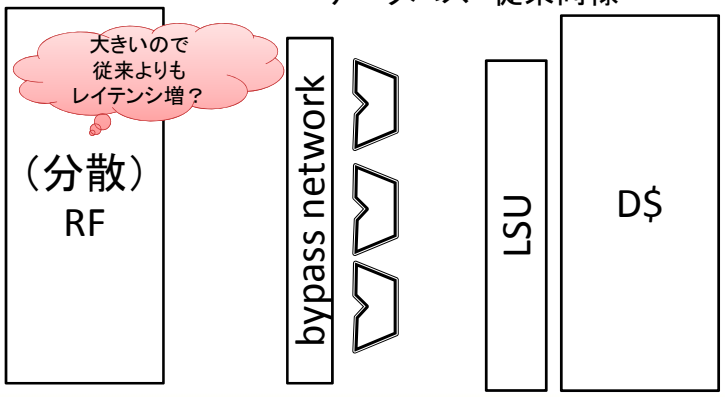
スケジュール　従来同様RAW依存を解決



パイプライン構成と実行イメージ

F D S RR E M RW

データパス　従来同様



パイプライン構成と実行イメージ

F D S RR E M RW

レジスタ書き込み：バックエンド後そのまま書き込み

リタイア処理：インオーダ
例外判定後
RP, **PC**, **SP**のコミットステート更新

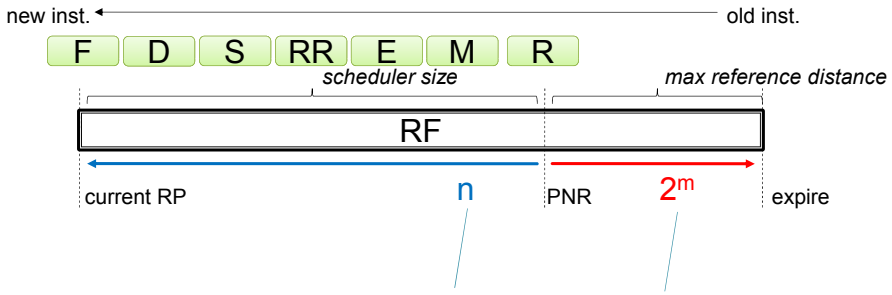
PC RP SP
数命令で数W
フロントエンドに依存なし

従来： 論理RF or RMTのコミットステート更新
フリーリスト更新

cRMT FL
毎命令数十W
フロントエンドに依存あり

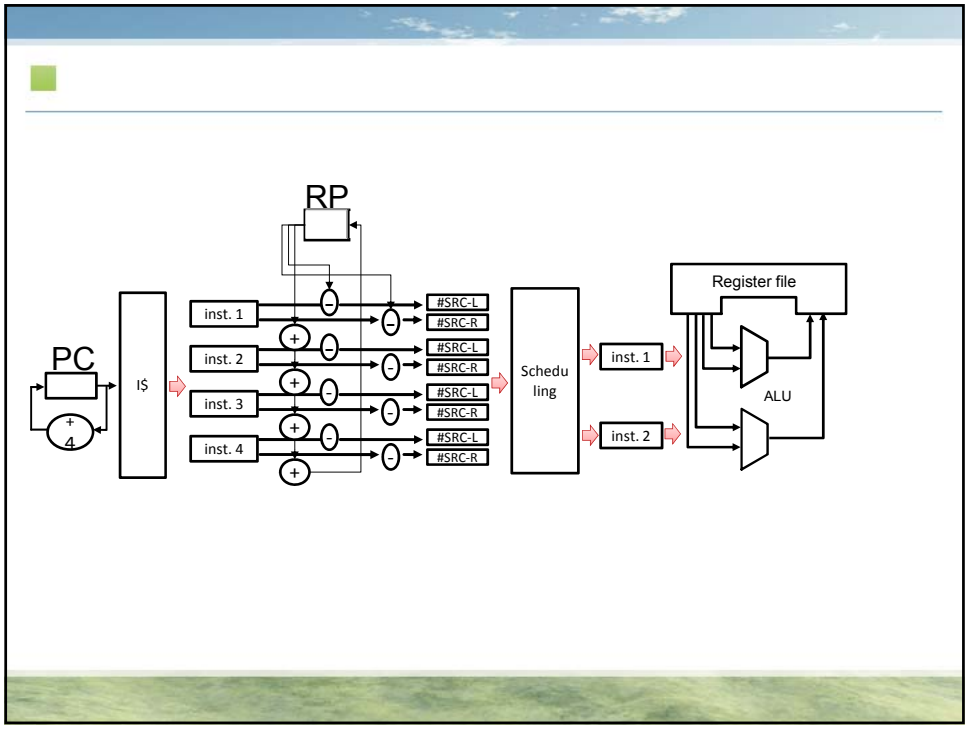
スケールパラメタ

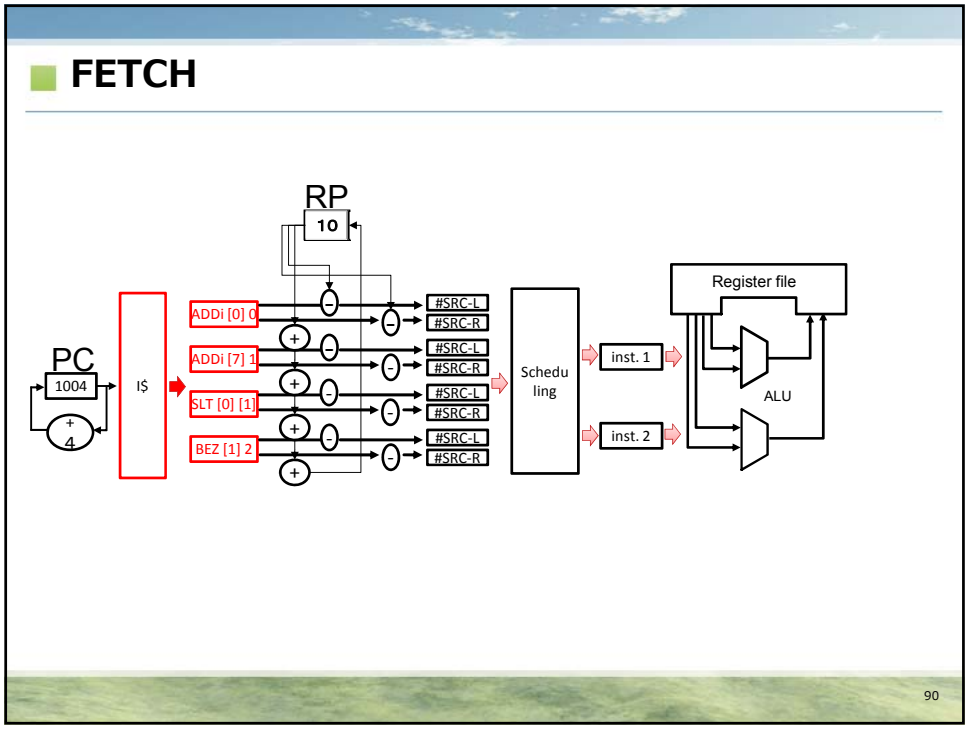
- 破綻しないためのRP空間を考える



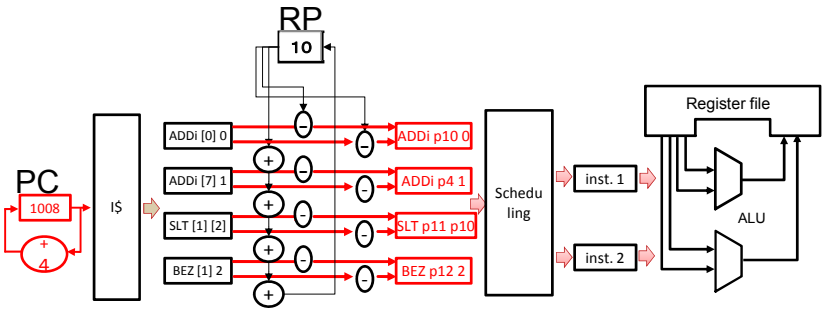
スケジューラエントリ　　最大参照距離

2^m+nのRP空間があればよい


RP
PC
4
I$
inst. 1
inst. 2
inst. 3
inst. 4
#SRC-L
#SRC-R
Schedu
ling
inst. 1
inst. 2
Register file
ALU

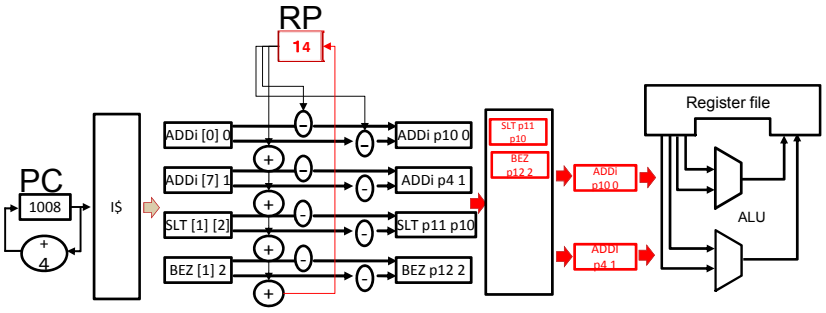

FETCH
RP
10
PC
1004
4
I$
ADDi [0] 0
ADDi [7] 1
SLT [0] [1]
BEZ [1] 2
#SRC-L
#SRC-R
Schedu
ling
inst. 1
inst. 2
Register file
ALU
90

DECODE and RENAME



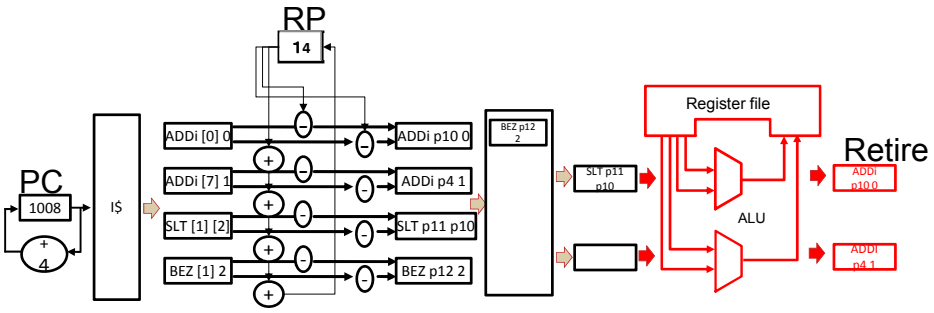
91

SCHEDULING



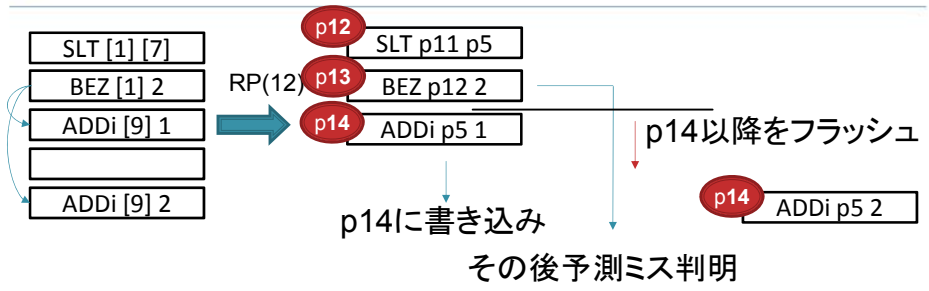
92

EXECUTE and Write Back



93

分岐ミスからの復帰



- フラッシュ処理は**大幅に簡略化**
  - **RP, PC, SP**をフラッシュ先頭の値へ
  - スケジューラの**スコアボード**をフラッシュ先頭以降初期化
    - あとは**自然に正しい状態へ上書きされる**
- SSではRMTとフリーリストの復帰が必要

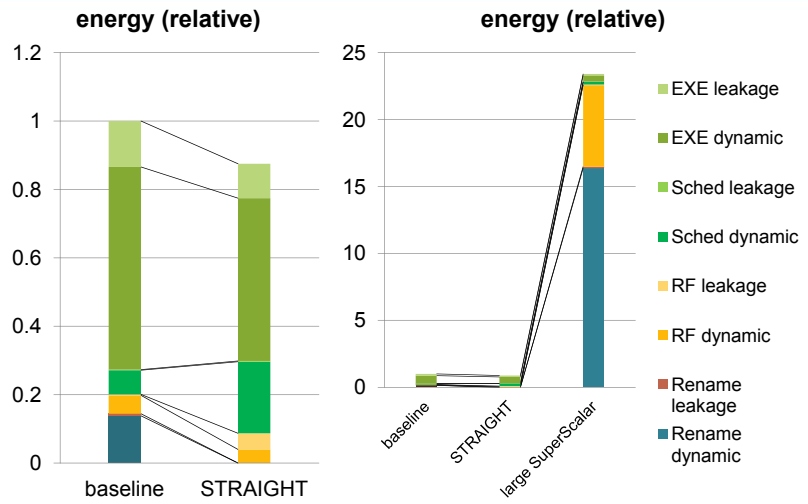
2016/11/28 2016年度 コンピュータシステム 第8回 94

スーパスカラの隘路の解消

- リネーム
  - 並列の足し算
- リタイア
  - Fire-and-Forget
- 分岐予測ミス時
  - テーブル回復不要

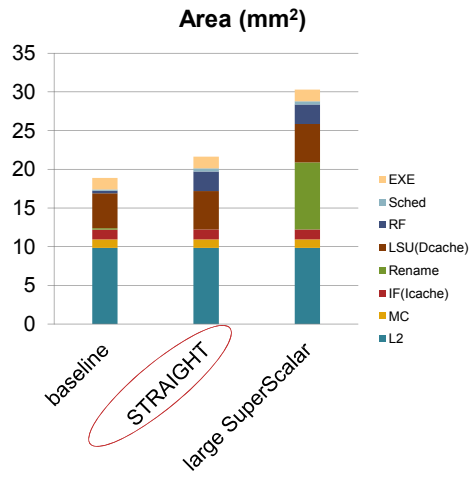
- 不要な制御電力の大幅削減
- スケジューラとデータパスの許す限り拡張可能

コア消費電力評価



IPC増と IPC/power増の双方を達成

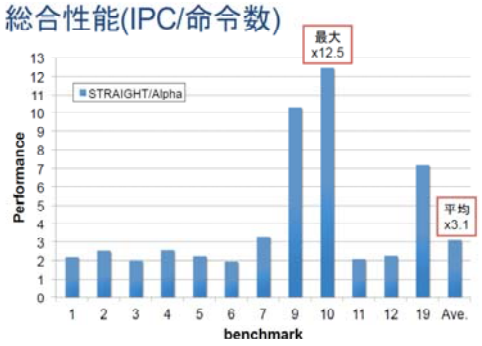
面積評価

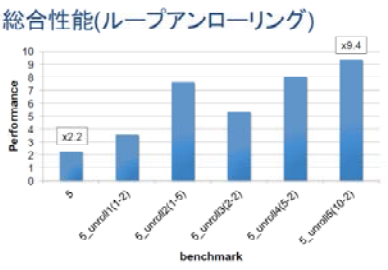


- RF増とRMT減の交換
  - 全体ではやや増加
- 従来アーキでは
  リネーム面積が爆発

性能評価

- livermoreループによる評価
- パイプライン幅が同じAlphaと比較して数倍～十数倍の性能
  - ただしハンドアセンブルコードvs. Gcc（今後詳細化）





今後の展開

- **汎用ベンチマーク**の性能評価
  - 拡張性，省電力，コンパイラ最適化により3倍の性能を目指す
- **配列計算，ループ計算**の最適化
  - 従来はメモリに格納するしかなかったデータ構造をレジスタに格納可能
  - ソフトウェアプリフェッチの効果拡大
  - ストア命令からの長距離フォワーディング
- **Deep Learning**向けアクセラレータとしての検討
  - 製造を視野に入れて企業との共同研究

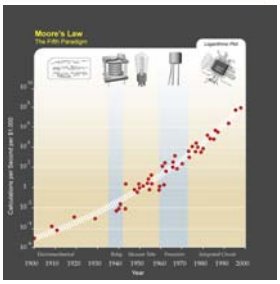
$$b_i = f(\Sigma W_{ij} a_j)$$

まとめ

2016/11/28

コンピュータ・システムという分野の魅力

- 持続する進歩
  - 常に異なる明日
    「同じ値段なら30年で
    1,000,000倍の計算力」



© Ray Kurzweil

- 自分・社会・人類への影響
  - 「個人の能力」も拡大してくれる

どう作っていくか＆
どう使っていくか
共に影響力大

70年代

今

「コンピュータ」の多様化・遍在化

- あらゆる物が急速にコンピュータ化
  - 従来は「**コントローラ**」
- 背景：**プロセッサ**の進歩
  コンピューティング能力付与のコスト低下
  - **1mm²**，**100mW**あれば
    それなりの**CPU**が**1GHz**で動く
  - **10W**も用意すれば
    マルチコアと**ベクタユニット**がつく
- **CPUが入ってくる意味**：プログラマブル
  - **ユーザの創意**をプログラムにして
    様々な要求に応えることができる
  - **プロセッサ性能向上**が
    ユーザの**能力獲得・回復**に
    さらに直接結びつくように

プロセッサ性能向上の話

- 最近のプロセッサ設計傾向
  - 使える向上要素をバランス良く，多角的に
- CPUコア性能を向上させる技術
  - 高性能コアのマイクロアーキテクチャ
- 研究中のアーキテクチャの紹介
  - コンパイラ支援によりリネームレスooo実行を実現

Questions?

```
ADDi 0 10
SPST 1 -8
ADDi 0 2
ADDi 0 1600
ST 2 1
SPST 2 -16
ADDi 0 3
ADDi 0 2400
ST 2 1
SPST 2 -24
ADDi 0 20
ADDi 0 0
UNDEF
SLT 3 2
BNZ 1 11
ADDi 13 0
ADDi 13 32
ST 2 1
UNDEF
ADDi 13 0
ADDi 13
```

```
32RMOV 13
ST 2 1
UNDEF
ADDi 13 1
J -12
ADDi 0 10
SPST 1 -32
SPLD -8
UNDEF
ADDi 0 1
UNDEF
SLT 4 2
BNZ 1 31
SPST 4 -40
SPLD -16
SPLD -24
LD 1 0
ST 1 3
ADDi 0 1
MULi 1 32
RMOV 6
```

```
RMOV 6
SPLD -32
ADD 4 3
LD 1 -32
UNDEF
SLT 8 4
BEZ 1 12
ADD 7 9
LD 1 0
ADD 6 1
ST 1 8
ADDi 14 1
MULi 1 32
RMOV 14
RMOV 14
RMOV 14
ADD 4 3
RMOV 8
J -13
SPLD -8
SPLD -40
```

```
ADDi 1 1
J -32
ADDi 0 0
UNDEF
```

Livermore Loop 2