平成２６年度修士論文

STRAIGHT アーキテクチャへの
レジスタキャッシュ適用による電力削減効果

大学院情報システム学研究科
情報ネットワークシステム学専攻

学 籍 番 号 :　1352027
氏 名:　山中 崇弘
主任指導教員: 入江 英嗣 准教授
指 導 教 員 :　長岡 浩司 教授
指 導 教 員 :　森田 啓義 教授
提出年月日 :　平成 27 年 2 月 27 日

(表紙裏)

目次

第1章　序論 **1**

第2章　関連研究 **3**

2.1 アーキテクチャにおける性能/電力比の改善 3
2.1.1 レジスタリネーミング 3
2.1.2 逆 Dualflow アーキテクチャ 4
2.1.3 Front-end Excutin Architecture 4
2.2 マイクロアーキテクチャの最適化による電力削減 5
2.2.1 スケジューラの行列化 5
2.2.2 Matrix Scheduler Reloaded 6
2.3 レジスタキャッシュ 7
2.3.1 Caching Processor General Registers 7
2.3.2 Non-Latency-Oriented Register Cache 7

第3章　STRAIGHT アーキテクチャ **9**

3.1 概要 9
3.2 STRAIGHT の命令セットアーキテクチャ 9
3.3 レジスタファイルとスケジューラの設計 10
3.4 STRAIGHT のパイプラインステージ 11

第4章　予備評価:STRAIGHT の消費電力評価 **13**

4.1 評価環境 13
4.2 評価結果 14

第5章　STRAGHT のマトリクススケジューラ適用 **19**

5.1 マトリクススケジューラにおける削減効果 19

第6章　STRAIGHT におけるレジスタキャッシュ **22**

6.1 STRAIGHT 用レジスタキャッシュの概要 22
6.2 STRAIGHT 用レジスタキャッシュの実装 22
6.3 STRAIGHT 用レジスタキャッシュの置き換えアルゴリズム 22
6.3.1 LRU(Least Recently Used) 形式の動作 22
6.3.2 FIFO 形式の動作 24
6.4 レジスタキャッシュを採用した際のパイプラインステージ 25

第 7 章　レジスタキャッシュを適用した際の評価 **26**
7.1 評価環境 26
7.2 評価結果 26
7.2.1 ヒット率 26
7.2.2 レジスタファイルの電力削減効果 28

第 8 章　議論 **30**

第 9 章　結論 **32**

謝辞 **33**

参考文献 **35**

図 目 次

2.1.1 コードサンプル . . . 3
2.1.2 レジスタリネーミング . . . 4
2.2.1 dependency matrix . . . 5
2.2.2 age matrix . . . 6
2.3.1 レジスタキャッシュ . . . 7
2.3.2 NORC のパイプラインステージ . . . 8

3.3.1 RP によるレジスタ番号の決定 . . . 10
3.3.2 RP のターンラウンド値とレジスタファイルサイズの関係 . . . 11
3.3.3 分岐予測ミスからの復帰 . . . 11
3.4.1 STRAIGHT パイプライン . . . 12

4.2.1 STRAIGHT と Alpha の電力比較 . . . 14
4.2.2 STRAIGHT と Alpha の消費電力比較 . . . 15
4.2.3 Alpha のスケールを 4 倍にした際の電力比較 . . . 16

5.1.1 スケジューラマトリクス . . . 19
5.1.2 マトリクススケジューラによるエントリ数とクリティカルパスの削減効果の関係 [11] 20
5.1.3 スケジューラマトリクス適用による削減効果 . . . 21
5.1.4 スケジューラマトリクス適用による電力削減効果の比較 . . . 21

6.2.1 STRAIGHT アーキテクチャのブロック図 . . . 23
6.3.1 レジスタキャッシュの置き換え . . . 24
6.4.1 レジスタキャッシュ適用ときの STRAIGHT パイプライン . . . 25

7.2.1 レジスタキャッシュのヒット率 . . . 27
7.2.2 LRU 形式によるレジスタファイルの削減効果 . . . 28
7.2.3 レジスタキャッシュを適用した電力評価 . . . 29

8.0.1 提案手法の適用による削減効果 . . . 30

表目次

4.1 アーキテクチャパラメタ 13
4.2 McPat 設定項目 1 17
4.3 McPat 設定項目 2 18

7.1 電力削減効果策定用アーキテクチャパラメタ 26

第1章　序論

情報社会を支えるマイクロプロセッサは，半導体技術の成長に伴い成長を続けている．チップ上のトランジスタ数は集積技術の発展に従い増加しており，アーキテクチャ技術はこのトランジスタ資源の利用により，成長を続けている．2000 年頃よりプロセッサは電力，配線遅延などの問題 [1] が指摘され始め，トランジスタ資源をパイプライン段数や発行幅，もしくは実行幅に活用していく手法は消費電力の制約を受けることになった．この様に発行実行幅の拡張のみによる性能向上を行うとスーパスカラアーキテクチャにおいてパスが増大してしまう点が妨げとなる．加えて，仮にパイプライン幅を拡張したとしても，そのパイプライン幅を有効活用する様な並列に実行するワークロードが存在していないために，従来のアーキテクチャ設計の転換を促すことになった．

このアーキテクチャの方向性の転換により，トランジスタ資源の新な利用先としてコア数やメモリ帯域に挙げられる様に，プロセッサ自身の実行部分に活用する以外の面でのアプローチが行われていた [2]．しかしコア数増大のアプローチによるメニーコアプロセッサにも，並列化が行われたワークロードに対しても各コアにおいてスレッド同期のオーバヘッドや割り当てるスレッド長に偏りが生じ, 性能向上面において限界が訪れようとしていることが指摘されている [3]．加えてダークシリコン問題では，チップ上に集積されるトランジスタを全て駆動することが出来ない，メニーコアプロセッサの様にコア数増大のアプローチを続けると，22nm 世代のチップにおいては 21%が同時駆動されず，更に 8nm まで集積技術が進むとチップ上のトランジスタの内過半数が同時駆動されないことを主張している [4]．加えて，従来のアーキテクチャを根本的に転換し，プロセッサの成長戦略を続けていくためには各コアのシングルスレッド実行する性能を上げつつも電力は抑えることが求められている．

そこで我々の研究室ではこのダークシリコン問題を打破する性能/電力比を得るために，制御を軽量化しつつも高いシングルスレッド能力を有する STRAIGHT アーキテクチャを提案している [5]．STRAIGHT アーキテクチャはライトワンスマナーに従う十分な論理レジスタ空間を持つことで，従来のプロセッサにおいて主要な制御電力であるレジスタリネーミングを排除する．更に豊富な物理レジスタを持たせることで，電力のオーバヘッドであるフリーリスト管理を省略している．また，トランジスタ資源をレジスタ容量に充てることで従来のプロセッサにおいて困難であったフロントエンド幅と命令ウィンドウ幅の拡張を容易とする．そこで命令ウィンドウ幅の拡張による演算器の稼動率向上を狙いシングルスレッド性能の向上を狙う．この STRAIGHT アーキテクチャを，既存のプロセッサシミュレータに対して STRAIGHT としてのアーキテクチャパラメタを与えることで近似した初期評価においては，性能指標である IPC(Instruction Per Cycle) は 30%，性能/電力は 18%の向上であった．しかし既存のプロセッサを STRAIGHT アーキテクチャに見立てて行った評価となるため，STRAIGHT アーキテクチャの命令セットアーキテクチャが反映されておらず正確な評価ではない．

そこで本研究では STRAIGHT 専用のシミュレータによる詳細な評価の中で，STRAIGHT アーキテクチャにおいて電力の評価を行い，実行部分のオーバヘッドとなるレジスタファイルやスケジューラに関して最適化とその評価を行う．レジスタファイルに対しては，バイパスネットワー

クによるレジスタ値の再利用を補助するレジスタキャッシュの導入により，レジスタファイルへと直接アクセスする回数を減らすことで電力削減を狙う．一方スケジューラに対しては，大きな命令ウィンドウに対して行列化を適用してクリティカルパスの削減による電力削減を狙う．以上の 2 つの手法を STRAIGHT アーキテクチャに導入することによる削減効果の評価を本論文で行う．

本論文の構成として，第 2 章では, 関連研究と題して性能/電力比の改善を目指している物を取り扱った後に，最適化に利用するレジスタキャッシュとマトリクススケジューラに関する論文を紹介する．第 3 章では，STRAIGHT アーキテクチャの詳細な説明を行い，第 4 章では，STRAIGHT 専用のシミュレータを用いた電力評価を行う．第 5 章では，マトリクススケジューラによる STRAIGHT アーキテクチャの電力削減効果を見積る．第 6 章では，STRAIGHT 用のレジスタキャッシュの仕様とアルゴリズムについて説明し，第 7 章では，レジスタキャッシュとマトリクススケジューラを導入した評価をする．そして第 8 章で結論を述べる．

第2章　関連研究

2.1　アーキテクチャにおける性能/電力比の改善

2.1.1　レジスタリネーミング

まず図 2.1.1 のようなコードを仮定したとき，既にオペランドが揃って実行できる命令 3 が待機していた場合に記述された順番に実行していく in-order 実行では，この命令の待機時間がボトルネックとなってしまう．

```
命令1： LD    r1,   [a];         ←メモリ上の変数aをレジスタr1に読み込む
命令2： ADD   r2,   r1,   r5;    ←r1とr5を加算してr2に格納
命令3： SUB   r1,   r5,   r4;    ←r5からr4を除算してr1に格納
```

図 2.1.1: コードサンプル

そこで従来のスーパスカラでは性能向上のために実行できる命令から実行していく Out-of-Order 実行がある．しかし図 2.1.1 を実際に Out-of-Order 実行した場合に命令 3 が先に必要なオペランドが揃ったことを仮定すると，Out-of-Order 実行により命令 2 の実行結果の内容がコードを記述した者との意図とは外れた結果となる．このような依存関係による問題の解決としてコードの記述上使用する論理レジスタを実際には別のレジスタに適用するレジスタリネーミングがある．このレジスタリネーミングは論理レジスタと物理レジスタを関連付ける RMT(RMT:Register Mapping Table) に対して読み書きを行う．また利用する物理レジスタから利用可能な物理レジスタを管理しているフリーリストを介することで依存関係の生じる論理レジスタを割り当てる．(図 2.1.2) このレジスタリネーミングにより Out-of-Order 実行が偽依存に縛られずに行うことが出来る．

従来のスーパスカラプロセッサにおいて，発行幅や実行幅を増やしているのにも関らず演算器の稼動率が低い．すなわち命令ウィンドウサイズが不足しているために，発行幅や実行幅を増やしても演算器の稼動率は向上しない．しかし命令ウィンドウサイズの増大は物理レジスタの増加を必要としており，この物理レジスタの増加は多ポート RAM である RMT の容量が増え，読み書きする際の電力も増大する．加えて大規模の命令ウィンドウを持つためには自身の充填率を高めなければ，その大規模なウィンドウサイズを有効活用できない．そこでフロントエンド幅を拡張して命令ウィンドウの充填率を上げていく．その一方でフロントエンド幅の拡張はリネーム処理を複雑化させるため，電力が増大する．

このように out-of-Order スーパスカラプロセッサにおける主要な制御であるレジスタリネーミングの複雑化から，単純にフロントエンド幅や命令ウィンドウを拡張させる手法により性能向上をさせることは難しい．そこで次の節からは制御電力を改善していくことで，性能/電力比を向上させる研究を紹介する．

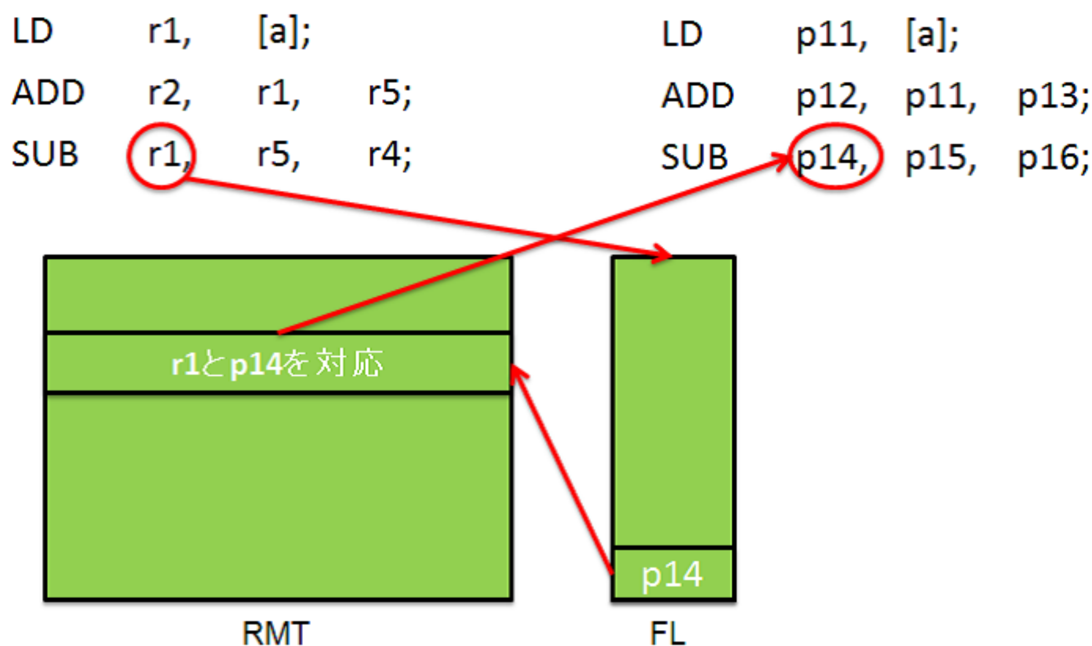

図 2.1.2: レジスタリネーミング

2.1.2 逆 Dualflow アーキテクチャ

Out-fo-Order スーパースカラプロセッサにおいてレジスタリネーミングを省略する研究として，dualflow アーキテクチャ[6] を基にした逆 Dualflow アーキテクチャが提案されている [7].「n 命令前」の実行結果ををトレースキャッシュに保持，再利用することによってレジスタリネーミングを省略している．

その結果レジスタリネーミングに必要なハードウェアの電力の削減効果とリネームステージの簡略化による分岐予測ミスのペナルティの減少による電力削減効果と性能向上が得られるが，その一方トレースキャッシュでトレースする命令の種類が Dualflow 方式への変換結果のトレースをパスによる区別を行うことによるトレースの種類の増加により，トレースキャッシュのミス率が増加してしまう．しかしこのミス率の増加はパイプラインが短縮される効果により保てることを主張している．

2.1.3 Front-end Excutin Architecture

Front-end Excution Architecture(FXA)[8] では，Out-of-Order に実行するユニット (OXU) と in-order に実行するユニット (IXU) の 2 つを組み合わせることで性能/電力比を向上させる．OXU については Out-of-Order のスーパースカラプロセッサと共通の実行コアを持つが，IXU は functional unit とバイパスネットワークのみに実行コアを持ち，フロントエンドに位置する．また IXU はスケジューリングなしで命令を実行するため，Out-of-Order のスーパースカラプロセッサにおける動的スケジューリングを行わないために電力が小さくなる．FXA では OXU と IXU の使用比率として IXU が 50%以上を占めており，到達点として従来型のスーパースカラプロセッサと比べると性能/電力

比は 25%の向上，従来型の in-order のスーパースカラプロセッサと比べると性能/電力比は 27%の向上を示した.

2.2　マイクロアーキテクチャの最適化による電力削減

2.2.1　スケジューラの行列化

ディスパッチされた命令を溜めて，その命令が発行許諾された命令を溜めて，その命令が必要とするソースオペランドが揃った際に命令の発行をする処理を行う機構をスケジューラと呼ぶ．このスケジューラでは発行された命令を実行可能な状態に起こす機構である wakeup ロジックと，どの命令を行うかを選択する picker(select) ロジックがある.

wakeup ロジックや select ロジックについて，チップ上の集積率の増加により，配線遅延が指摘されている [9]. そこで wakeup ロジックや select ロジックの行列化が行われている [10]. wakeup ロジックでは dependency matrix と言う手法が用いられ，スケジューラ内の命令に対して行と列を一つずつ持ち, 各セルがステートを表すビットを持つ. このビットは該当する行に割り当てられた命令が，該当する列に割り当てられた命令からの依存関係で待機しているかどうかを示す．図 2.2.1 のようにビットがセットされている場合を例にあげると，A のブロードキャストチャンネルは双方共に 0 を示しており ready 信号を出す準備が出来ている. 一方 B のブロードキャストチャンネルは命令 A に対してのみ依存関係があることを示している.

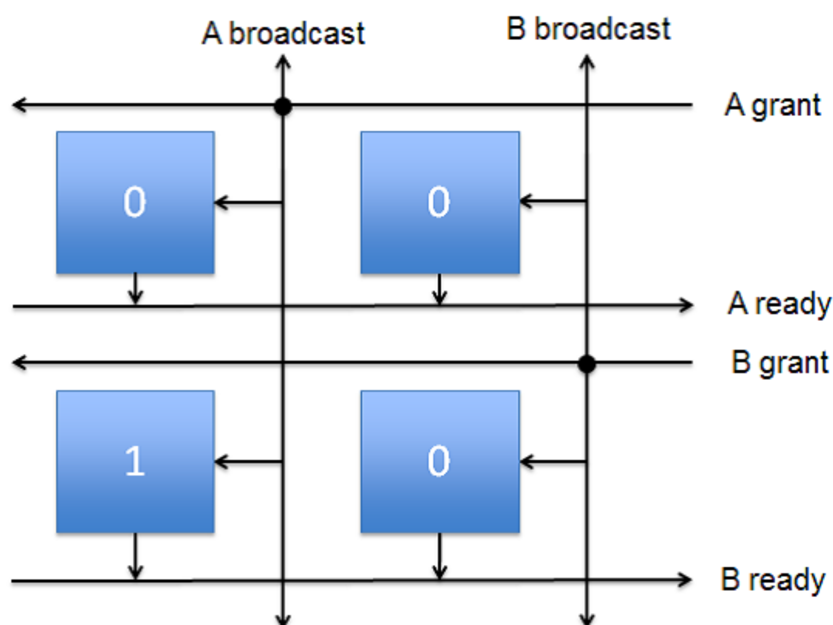

図 2.2.1: dependency matrix

この行列化により，従来の wakeup ロジックと比較すれば高速かつ低電力を実現できることが指摘されている. しかし多数の列を持たせることにより，列の数に比例してブロードキャストチャンネルが増加し行では命令の発行したとき，点を管理する ready generation チャンネルが増加する. これはスケジューラが大きくなれば大きくなる程より顕著となる.

一方 picker ロジックでは age matrix と言う手法があり，これはスケジューラ内の命令単位に行と列を一つずつ持つ．各セルは age conflict bit と呼ばれるビットを一つ持ち，命令全てにこのビットがセットされる．このビットは 1 の場合にはその列に対応する命令よりも古い命令であることを示し，0 である場合にはその行に対応する命令が若いことを示す．

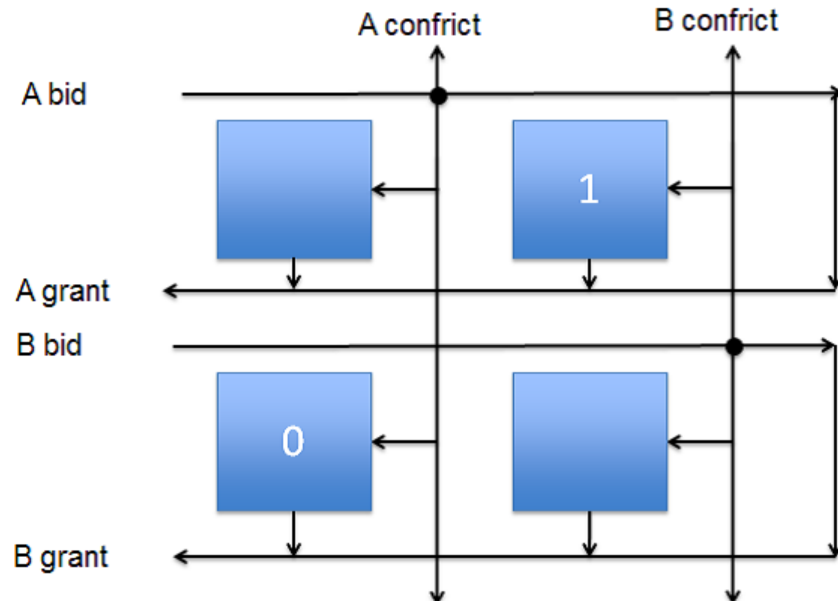

図 2.2.2: age matrix

図 2.2.2 は age matrix の例を示している．命令 A と命令 B が ready 信号を発することにより picker ロジックに入る．すると命令 A が B confrict の列に対して 1 が立つ．このようなときに，age-matrix は命令 B に grant 信号を出す．

2.2.2 Matrix Scheduler Reloaded

Peter G.Sassone らによる Matrix Scheduler Reloaded を用いることで，大規模なスケジューラの行列化において複雑化するクリティカルパスを削減することが可能と言うことを主張している [11].

Wakeup ロジックについては，列に与える結果であるソースオペランドの管理を行う小型のテーブルである WAT(Wakeup Allocated Table) を使うことで，全ての列のブロードキャストを管理する必要がなくなる. 結果として Wakeup ロジックのクリティカルパスを低減することが可能となることを主張している．Picker ロジックについても行の命令 ID を PAT(Picker Allocated Table) と言う小型のテーブルが管理する．Picker ロジックでは，命令 ID に対応した Wakeup ロジックから ready 信号が出ている命令を age-matrix の列を割り当てる．

このスケジューラマトリクスを用いることで，16 エントリでは 33%削減でき，そのままエントリ数の増加従い 128 エントリのスケジューラになると 58%削減できることを示している．すなわち，スケジューラサイズが増大するに従って，スケジューラマトリクスはより大きな効果を発揮することを主張している．

2.3 レジスタキャッシュ

2.3.1 Caching Processor General Registers

プロセッサがパイプライン処理を行う際に, ある命令 B が一つ前の命令 A の結果を使用して行う場合に A と B の命令間隔が短い場合, 命令 B の実行は命令 A がレジスタに書き込むのを待機する. そこで命令 A が実行結果をレジスタに書き込むのを待たずに利用する技術としてバイパシングがある. この技術を利用することでパイプライン処理が滞ることなく実行することが出来る.

このバイパシングを活用するレジスタキャッシュと呼ばれる小容量かつ高速なバッファが提案されている [12]. レジスタキャッシュはレジスタファイルへ直接アクセスする訳ではなく, 図 2.3.1 に示すようにレジスタファイルとパイプラインの間に位置しており, バイパスネットワークの補助を行うことを狙いとしている.

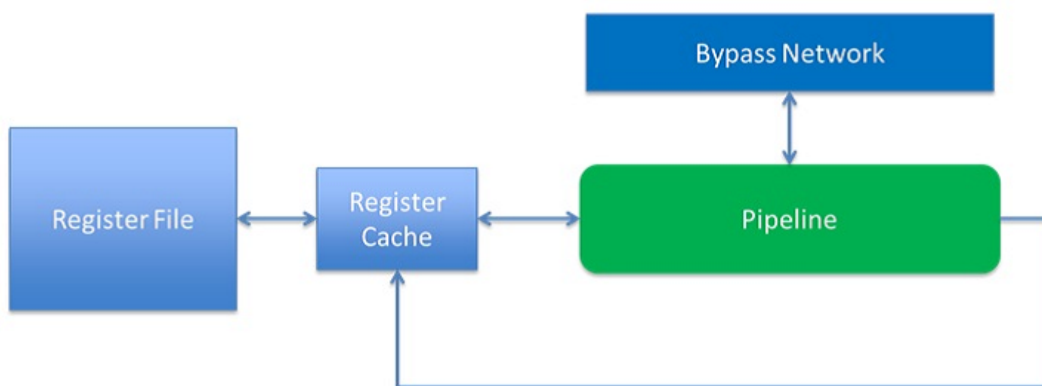

図 2.3.1: レジスタキャッシュ

命令処理を行う際に使用するレジスタ値は直近の命令のみが利用することも示されており, 最初のレジスタ値の使用から現在計算している命令の距離について spec92 の Int を 6 本使用して検証した結果, 全ての計算に用いられた各々のレジスタ値の 50%は二命令先までで使用されている. 加えてパイプラインがバイパスネットワークから供給されるレジスタ値も含めて, レジスタキャッシュで補助をすることによりほぼ全てのレジスタ値を供給することが可能であることも主張している.

レジスタキャッシュはレジスタ値をキャッシュするエントリを所持しており, 同論文では 16 エントリ程度で 96%がレジスタキャッシュにより賄え, 32 エントリでは 99%が賄えることを示した. すなわち 32 エントリのレジスタキャッシュの導入により，パイプライン処理において要求するレジスタ値は 99%はレジスタファイルへのアクセスを必要としないことを示している.

2.3.2 Non-Latency-Oriented Register Cache

レジスタキャッシュを導入することによるレジスタファイルの負荷の低減に着目した研究として NORC(Non-Latency-Oriented Register Cache) がある [13]．NORC は従来のレジスタキャッシュと同様のシステムを持つが，従来のレジスタキャッシュと違い図 2.3.2 のようにレジスタキャッシュが

ヒットするかミスするかに関係なくレジスタリードのステージを持ち，レジスタキャッシュリード (CR) とレジスタキャッシュライト (CW) を持つ．

すなわち，従来のレジスタキャッシュはヒットすることを踏まえたパイプライン構成をしているが，NORC はミスすることを踏まえたパイプライン構成となる．

図 2.3.2: NORC のパイプラインステージ

この NORC によりレジスタキャッシュ自体の電力を除けば，4-way のスーパスカラプロセッサのレジスタファイル自体の電力を 68.1%削減することを主張している．

第3章　STRAIGHTアーキテクチャ

3.1　概要

前章までで紹介したように，チップ上のトランジスタを活用しきれるするには性能を向上させるだけではなく，電力を削減させるアーキテクチャが求められており，性能/電力比を改善する提案がなされている. そこで我々の研究室では，豊富なトランジスタ資源をレジスタ容量に充てる事で制御を軽量化するアーキテクチャ「STRAIGHT」を提案する．STRAIGHT は以下の 3 点を主なコンセプトとしている.

1. ライト · ワンス · コードを広大なレジスタ空間にマッピング
2. 豊富な物理レジスタにより命令のフェッチ順に割り当て
3. 軽量なレジスタ制御による命令ウィンドウの拡張

次にこの 3 点による電力の削減効果と性能向上について第 2 章の議論を踏まえて詳しく述べる.

まず十分な論理レジスタ空間を仮定して，一度使用したレジスタを再び使用することのないコードを記述する．するとコード上で後の命令が前の命令に対して依存関係である逆依存が生じなくなるため，レジスタリネーミングをせずとも Out-of-Order 実行が可能となる．すなわち従来のプロセッサにおいて高価な処理であるレジスタリネーミングを不要とすることができる．その為十分な論理レジスタ空間を与えることで，制御に割り当てられていた電力を削減することに繋がる.

次に豊富な物理レジスタを背景に，物理ディスティネーションレジスタ番号を命令フェッチの順番そのままに割り振ることができる．そうすることで，レジスタの解放を上書きにより行う．すなわち従来のプロセッサにおいて物理レジスタの空きを管理していたフリーリスト管理を不要とすることができる．以上より，従来のプロセッサにおける制御であるフリーリスト管理を不要とする事で電力削減に繋がり，デコード処理とディスパッチ処理を命令フェッチ順に行うことが出来る.

前の章で述べた様にフロントエンド幅はリネーム処理の複雑化による制約を受けていたが，リネーム処理を必要としない STRAIGHT アーキテクチャにおいては拡張が容易となる．そうすることで，命令ウィンドウの拡張を行いつつ充填率を高める事が容易に行うことができる．このように，命令ウィンドウの拡張を行う事で，演算器の稼動率が向上し，性能の向上に繋る．加えて大容量のレジスタを持たせる事による副次的な効果として，本来であればメモリアクセスを行う様な命令をレジスタファイルが保持する事で性能の向上が見込められる.

3.2　STRAIGHT の命令セットアーキテクチャ

STRAIGHT アーキテクチャの命令形式は一般的な RISC 命令と同様に演算，転送，制御などのオペレーションを行う．命令は固定長で，典型的な 2 入力 1 出力の場合，命令の種類を示すオペコード，ソースオペランドレジスタ L，ソースオペランドレジスタ R を指定する.

その際にソースオペランドの値は Dualflow 形式の様な命令の距離により指定する．例えばソースオペランド L が 2 を指定した場合には現在の命令から 2 命令前の実行結果をソースオペランド L の値として利用する．また，ソースレジスタ R は即値として利用可能である．また，ディスティネーションレジスタの値は命令のフェッチ順に割り当てる為，ディスティネーションレジスタの指定を省略する．

3.3 レジスタファイルとスケジューラの設計

この節ではレジスタファイルと命令ウィンドウの設計について述べる．まず始めにレジスタファイルの設計について，図 3.3.1 の様に STRAIGHT ではレジスタ番号の決定の為に特殊なアーキテクチャレジスタ RP を導入する．RP はディスティネーションレジスタ番号の割り当てに用い，命令毎にインクリメントされる．加えて RP とソースオペランドの差分により，ソースレジスタ番号の指定を行う．このようにレジスタ番号を RP のみに依存させる事で命令同士の依存関係を解消し，並列化した処理を容易に行う事が可能となる．

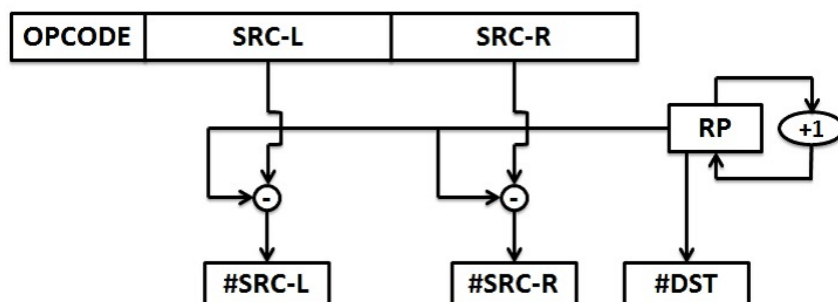

図 3.3.1: RP によるレジスタ番号の決定

STRAIGHT のソースオペランドのフィールドに 10 ビット与える．その為命令距離による記述で 2^{10} 個前の命令の実行結果まで参照することが可能となる．すなわち 2^{10} よりも前のレジスタ番号は，それ以降参照されないことを保証できるため，新しく割り当てることが可能となる．次にデコードステージから命令のリタイアの間に挟まる最大の命令数を n とすると，2^{10} 前の命令に加えて，さらに n 命令前まで参照することで命令の距離による記述を行う．この n はデコードステージから命令のリタイアの間に挟まる命令数なので命令ウィンドウに充填できる最大値である．以上から，レジスタファイル上に 2^{10} 個前までの全ての命令の実行結果と，命令ウィンドウに充填されている n 個の命令を載せることで，命令距離による記述を実現する. すなわち RP を 2^{10}+n でターンラウンドして 0 に戻るように設計すればよい．図 3.3.2 はパイプラインステージと命令ウィンドウサイズ，そしてレジスタファイルの関係を示した図である．

以上のように RP を導入することで，分岐予測ミスにより命令の巻き戻しをする場合に RP を戻すだけで分岐予測ミスから復帰することを可能とする．図 3.3.3 の (a) のように分岐予測ミスにより，分岐前まで戻る場合には RP を分岐命令の時の値に戻す．そうすると図 3.3.3 の (b) のように再び RP を振り分けることができる．

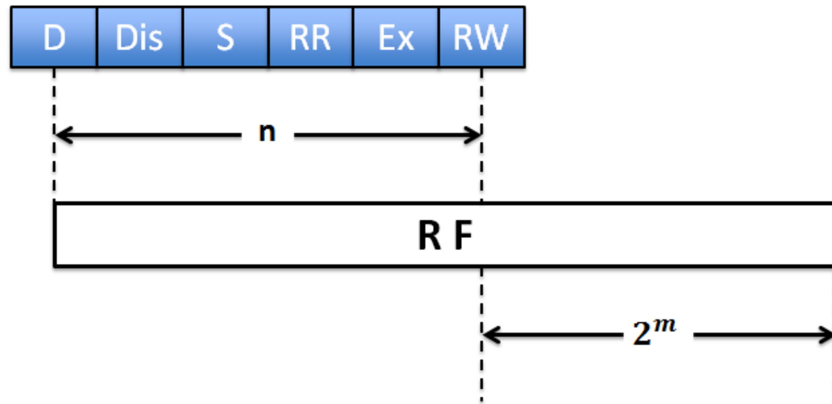

図 3.3.2: RP のターンラウンド値とレジスタファイルサイズの関係

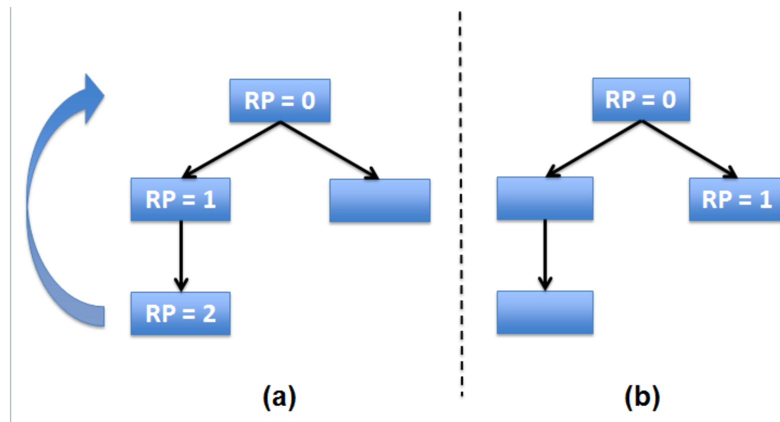

図 3.3.3: 分岐予測ミスからの復帰

3.4　STRAIGHT のパイプラインステージ

STRAIGHT アーキテクチャは図 3.4.1 の様にリネームステージが省略されている事を除けば，通常のスーパスカラと同様に，フェッチ (F)，デコード (D)，ディスパッチ (Dis)，スケジューリング (S)，レジスタリード (RR)，エクスキュート (Ex)，ライトバック (RW) のステージを持つ．

従来のスーパスカラプロセッサにおいては，ディスパッチをステージの前にリネーム処理が挟まるが，STRAIGHT アーキテクチャはリネーム処理が省略されているため，フェッチステージで命令を持ってきた後，デコードステージ，ディスパッチステージをそのまま行う事が出来る．以降のステージではスーパスカラと同様にスケジューリングでは命令の発行選択を制御をする．レジスタリードステージではソースオペランドが必要とするレジスタを読み，エクスキュートステージで命令の実行をする．レジスタライトステージでは命令の結果をレジスタファイルへと書き込む．すなわち通常のスーパスカラプロセッサに比べてリネームのレイテンシを削減する事により，パイプライン幅を短縮する効果がある．その一方でレジスタファイルの大容量化に伴い，レジスタリードステージは伸びる．

従来のスーパスカラプロセッサのパイプライン

STRAIGHTアーキテクチャのパイプライン

図 3.4.1: STRAIGHT パイプライン

第4章　予備評価:STRAIGHTの消費電力評価

4.1　評価環境

この章ではSTRAIGHTアーキテクチャにおいて開発したシミュレータとハンドコーディングにより作成されたベンチマークを用いた消費電力の評価を行う．またこの時のアーキテクチャパラメタは表4.1になる．STRAIGHTのアーキテクチャパラメタについては, 初期見積りで最も性能効率よく高めた物を採用しており，フロントエンドレイテンシはリネームやディスパッチの削減を反映させた物となっている．パイプラインシミュレータ鬼斬[14]を用いて得られた出力と与えたパラメータから入力を作成し，電力シミュレータであるMcPat[15]を用いる．このとき，MacPatの入力用パラメータの設定項目を表4.2と表4.3に示す．この2つの表は，parame name or stat nameの列にはMcPatの入力項目名を示しており，valueの項目には鬼斬による実行結果パラメタの名前を示している．なおvalueの項目で数値のみ記入されているものは，Alphaアーキテクチャと STRAIGHTアーキテクチャ向けに設定した項目である．また，この表に書き加えられていないパラメタに対してはMcPatのdefault値を利用した．

ベンチマークはAlphaとSTRAIGHT双方の命令数の差とサイクル数の差が平均値を示したLivermoreKernel5を用い，同じ処理における評価を行う．これは，AlphaとSTRAIGHTにおいて命令セットアーキテクチャが違うため単純な性能評価が出来無い為である．

表4.1: アーキテクチャパラメタ

	ベースライン / STRAIGHT
フロントエンド幅	4 / 8
リタイア幅	6 / 12
スケジューラサイズ	int32+fp16 / int128+fp64
レジスタファイル	int128+fp128 / int512+fp512
フロントエンドレイテンシ	7cycle / 5cycle
発行幅	int2,fp2,mem2
D1キャッシュ	64KB,8way,64Bline,3cycle hit latency
I1キャッシュ	64KB,8way,64Bline,3cycle hit latency
D1キャッシュ	64KB,8way,64Bline,3cycle hit latency
メインメモリ	200 cycle

4.2　評価結果

我々の提案した STRAIGHT アーキテクチャにおいて命令ウィンドウやレジスタ容量を 4 倍としつつもフロントエンド幅を 2 倍としたモデルと，既存のスーパースカラである Alpha のモジュール毎の電力の評価を図 4.2.1 に，消費電力の評価を図 4.2.2 に示す.

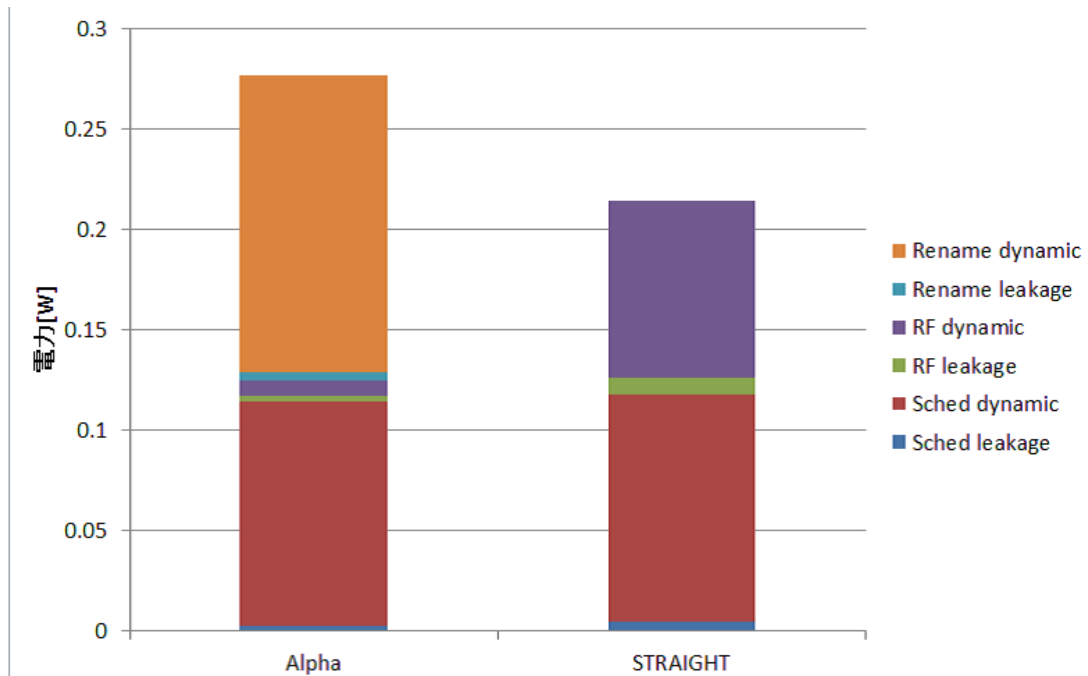

図 4.2.1: STRAIGHT と Alpha の電力比較

これらの図はリネーム処理による RMT の読み書きにより生じる電力を示す Rename dynamic と Rename leakage，レジスタファイルの読み書きによる電力を示す RF dynamic と RF leakage，スケジューラの電力を示す Sched dynamic と Sched leakage を Alpha と STRAIGHT の各々の結果を表示している.

図 4.2.1 を見ると，Alpha においてはスーパースカラにおける電力が高い事が知られるレジスタリネームのロジックがキャッシュを除いた実行部分全体の約 55%を示している．その一方で STRAIGHT においてはレジスタリネーミングを不要としているので，リネーム処理にかかる電力は削減されている．しかしながらレジスタファイルの読み書きに生じる電力は Alpha と比較して 11.2 倍の増加，スケジューラにおいては約 2%の増大を示している．STRAIGHT アーキテクチャのコンセプトの通りに，従来のスーパスカラプロセッサにおいて高価な処理であるリネームロジックを軽減することは出来たが，レジスタファイルやスケジューラロジックが実行部分のオーバヘッドとなる事が示された．このように，レジスタファイルの読み書きやスケジューラの活用による電力の増加を踏まえても，実行部分自体の電力は STRAIGHT アーキテクチャの方が同じサイクル数では低電力で行うことが可能である.

一方図 4.2.2 の様に同じ命令数を実行する事にかかるサイクル数を反映させた消費電力を見ると，STRAIGHT アーキテクチャの消費電力は Alpha から 56.4%削減可能である事が示された．加えて Alpha において STRAIGHT アーキテクチャと同じスケールを実現する事を試みた場合，その消費

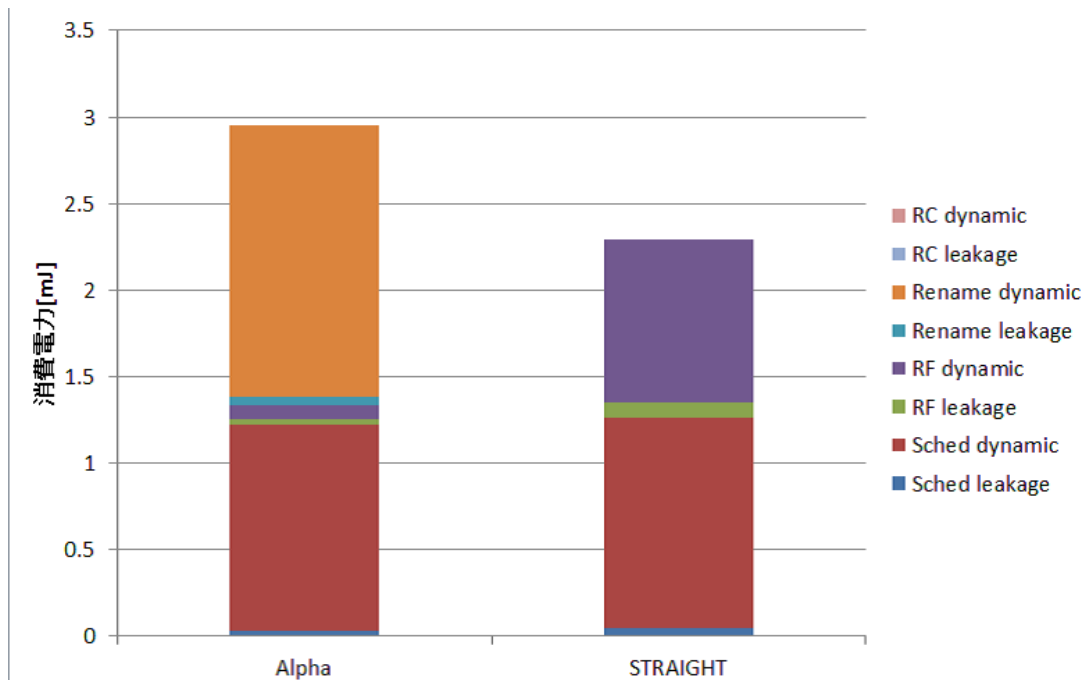

図 4.2.2: STRAIGHT と Alpha の消費電力比較

電力は図 4.2.3 に示す結果となる．

図が示す様に既存のアーキテクチャで STRAIGHT と同規模のアーキテクチャを実現するには，前述した様にフロントエンド幅の拡張によるレジスタリネーミングの複雑化，加えてレジスタファイルの大容量化による RMT の稼動率向上により非現実的なリネーム電力が掛かる．すなわち，性能向上させるためにレジスタファイルや命令ウィンドウを確保しても，リネーム電力が増大してしまうため性能/電力比の向上には結び付かないことが示された．

以上の結果から STRAIGHT アーキテクチャは制御を軽量化した為消費電力を削減し，性能/電力比を向上させる事が示された．一方で図 4.2.1 が示す様に，STRAIGHT アーキテクチャは従来のスーパスカラアーキテクチャと比較するとレジスタファイルやスケジューラの電力が実行部分のオーバヘッドとなる．この大規模なレジスタファイルとスケジューラの電力を削減する手法は既に存在しており，第 2 章で紹介してきた．そこで次の章からはこの実行部分のオーバヘッドを最適化する事により STRAIGHT アーキテクチャの電力を改善して，更なる性能/電力比向上の手法について議論する．

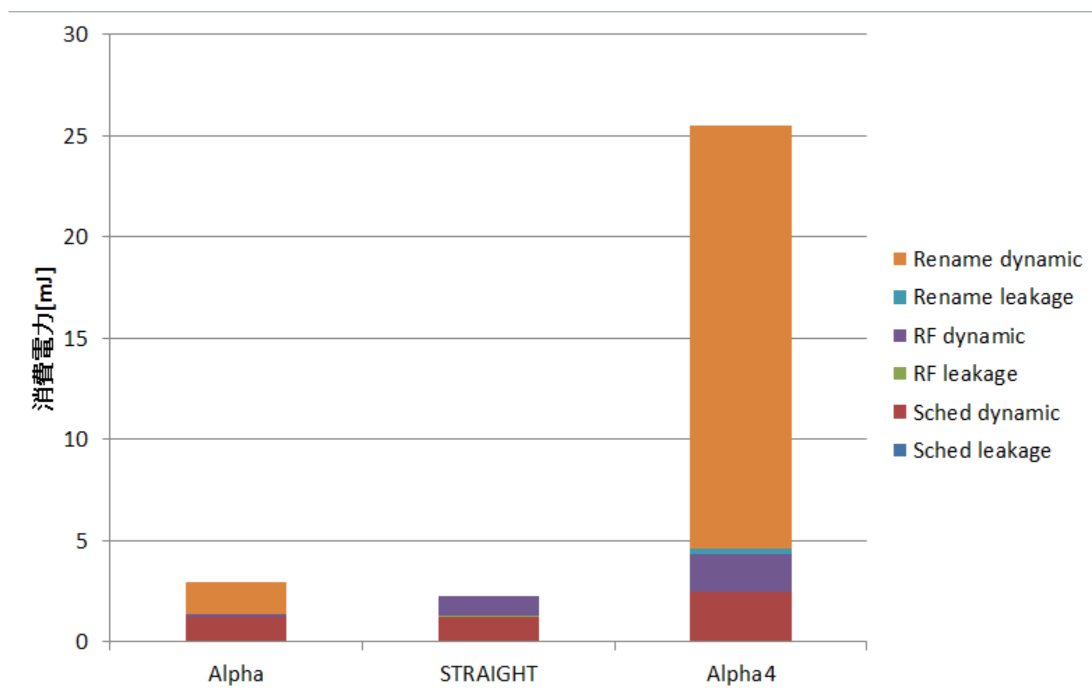

図 4.2.3: Alpha のスケールを 4 倍にした際の電力比較

表 4.2: McPat 設定項目 1

No.	param name or stat name	value
1	number_of_L2Directories	0
2	core_tech_node	22
3	target_core_clockrate	3000
4	temperature	353
5	total_cycles	System の ExcutedCycles
6	busy_cycles	System の ExcutedCycles
7	clock_rate	3000
8	fetch_width	Fetcher の FetchWidth
9	decode_width	Fetcher の FetchWidth
10	commit_width	Retirer の Width
11	pipeline_depth	Fetcher と Renamer と dispatcher と Scheduler[0] の issue の Latency の合計+1，左の値の+3
12	MUL_per_core	2
13	FPU_per_core	2
14	instruction_buffer_size	Scheduler[0] と [1] と [2] の WindowCapacity の合計
15	instruction_window_size	Scheduler[0] と [2] の WindowCapacity の合計
16	fp_window_size	Scheduler[1] の WindowCapacity
17	ROB_size	InorderList の Capacity
18	archi_Regs_IRF_size	RegisterFile の Capacity の第一項
19	archi_Regs_FRF_size	RegisterFile の Capacity の第二項
20	phy_Regs_IRF_size	RegisterFile の Capacity の第一項
21	phy_Regs_FRF_size	RegisterFile の Capacity の第二項
22	load_buffer_size	MemOrderManager の Capacity
23	RAS_size	RAS の StackSize
24	total_instructions	Fetcher の NumFetchedOp
25	int_instructions	PipelinedExecUnit の Name=”ex∗” の NumTotalUsed の合計 ∗ = IntBC，IntALU，IntMUL, Addr
26	fp_instructions	PipelinedExecUnit の Name=”ex∗” の NumTotalUsed の合計 ∗ = FloatBC，FPAdd，FPMUL
27	branch_instructions	Bpred の Statistics の NumMiss + NumHit
28	branch_misprediction	Bpred の Statistics の NumMiss
29	load_instructions	CacheSystem の Load の CacheAccessDispersion の合計
30	store_instructions	CacheSystem の Store の CacheAccessDispersion の合計

表 4.3: McPat 設定項目 2

No.	param name or stat name	value
31	committed_instructions	System の ExecutedInsns
32	committed_int_instructions	No.31 を No.25 と No.26 の比で分配
33	committed_fp_instructions	No.31 を No.25 と No.26 の比で分配
34	ROB_reads	System の ExecutedInsns
35	ROB_writes	System の ExecutedInsns
36	rename_reads	Fetcher の NumFetchedOp の二倍
37	rename_writes	Fetcher の NumFetchedOp
38	fp_rename_reads	No.33 の value の二倍
39	fp_rename_writes	No.33 の value
40	inst_window_reads	System の ExecutedInsns
41	inst_window_writes	System の ExecutedInsns
42	inst_window_wakeup_accessed	System の ExecutedInsns の 2 倍
43	fp_window_reads	No.33 の value
44	fp_window_writes	No.33 の value
45	fp_window_wakeup_accessed	No.33 の value の 2 倍
46	int_regfile_reads	System の ExecutedInsns の 1.5 倍
47	float_regfile_reads	No.33 の value の 1.5 倍
48	int_regfile_writes	System の ExecutedInsns の 0.75 倍
49	float_regfile_writes	No.33 の value の 0.75 倍
50	function_calls	0
51	context_switches	0
52	ialu_accesses	No.25 の value
53	fpu_accesses	No.26 の value
54	mul_accesses	PipelinedExecUnit Name="exIntMUL"の NumTotalUsed
55	cdb_alu_accesses	No.52 の value
56	cdb_mul_accesses	No.54 の value
57	cdb_fpu_accesses	No.53 の value

第5章　STRAGHTのマトリクススケジューラ適用

5.1　マトリクススケジューラにおける削減効果

2章ではスケジューラのクリティカルパスの削減効果について述べてきた．ここで図5.1.1の様なロジックを例にあげる．

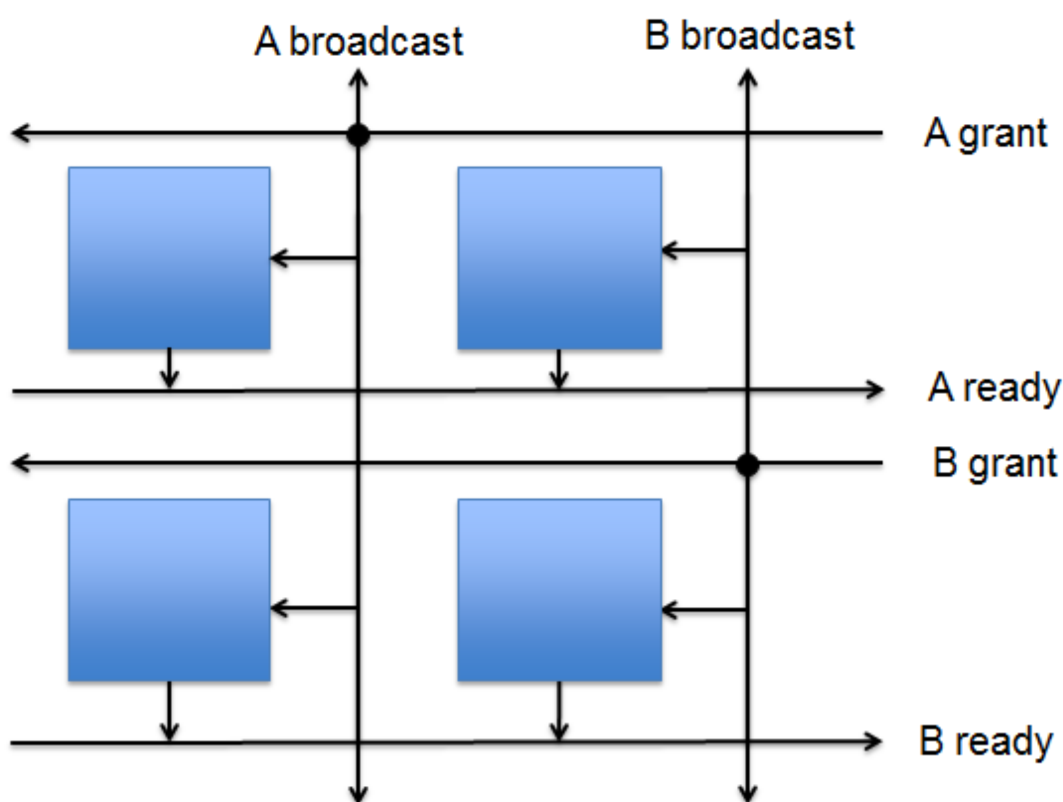

図5.1.1: スケジューラマトリクス

STRAIGHTにマトリクススケジューラを適用する場合にも，ブロードキャストつまり列においてはソースオペランドIDが対応し，ready信号が出される．このready信号を出している命令の中からselectロジックが次に発行する命令を選択して，対象となる命令にgrant信号出す．このような手順を取ることでマトリクススケジューラでは命令の発行を行う．

マトリクススケジューラの適用により，スケジューラのクリティカルパスは図5.1.2のように削減できる．

STRAIGHTのスケジューラサイズはintもfloatも128エントリであるため，マトリクススケジューラの適用により，クリテカルパスは58%削減できる．そこでMcPatの評価においてwakeup

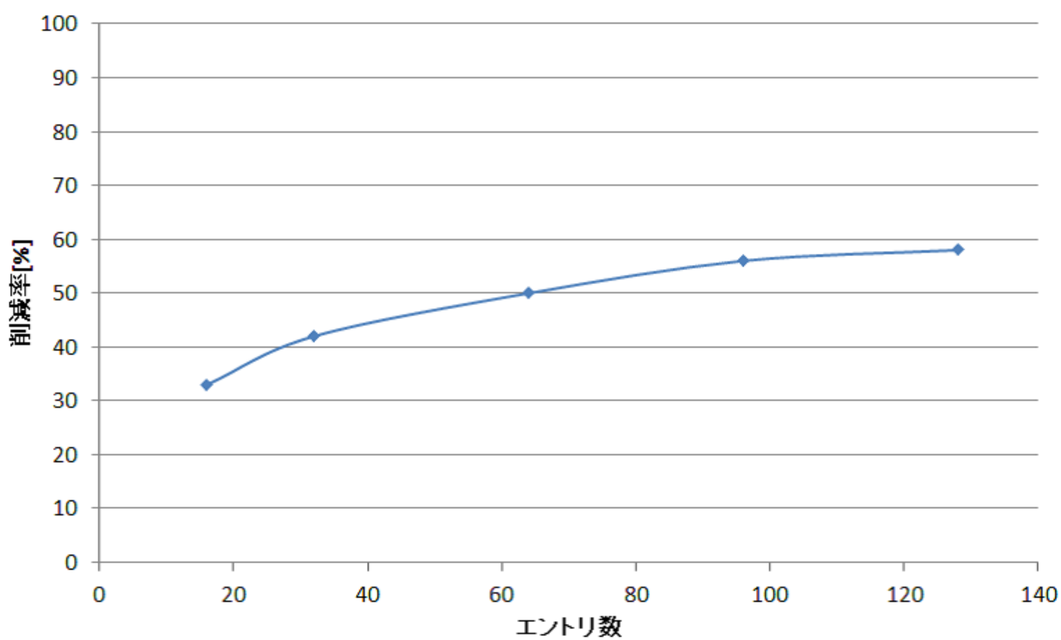

図 5.1.2: マトリクススケジューラによるエントリ数とクリティカルパスの削減効果の関係 [11]

ロジックのアクセス回数と Select ロジックのアクセス回数は等価として扱い，wakeup アクセスを 58%抑えた物として行った．また，今回利用した LivermoreKernel5 は int 演算のみを行っている．すなわち表 4.2 において，No.42 の value に対して 0.42 をかけることでマトリクススケジューラ適用時の McPat の入力パラメタとし，マトリクススケジューラ適用時の電力を測定する．

その結果，STRAIGHT アーキテクチャの 1 サイクル辺りにおけるスケジューラの電力は図 5.1.3 のように見積もれる．

wakeup アクセスの削減により，マトリクススケジューラを適用していない STRAIGHT アーキテクチャと比べて，スケジューラ自体の電力は 41.2%削減することを示した．

次にアーキテクチャ全体としての削減効果を，Alpha とマトリクススケジューラを適用させていない STRAIGHT と適用させた STRAIGHT を比較し，同じ処理を行うときの消費電力を図 5.1.4 に示す．

その結果マトリクススケジューラを適用していない STRAIGHT アーキテクチャに比べて，マトリクススケジューラを適用することで，実行部分の 23.0%削減することが見積られる．またこの図 5.1.4 において，Alpha のスケジューラ電力の削減効果は薄い．なぜならば Alpha のスケジューラのエントリ数は 32 エントリと小さく図 5.1.2 に従い 42%の削減効果である．

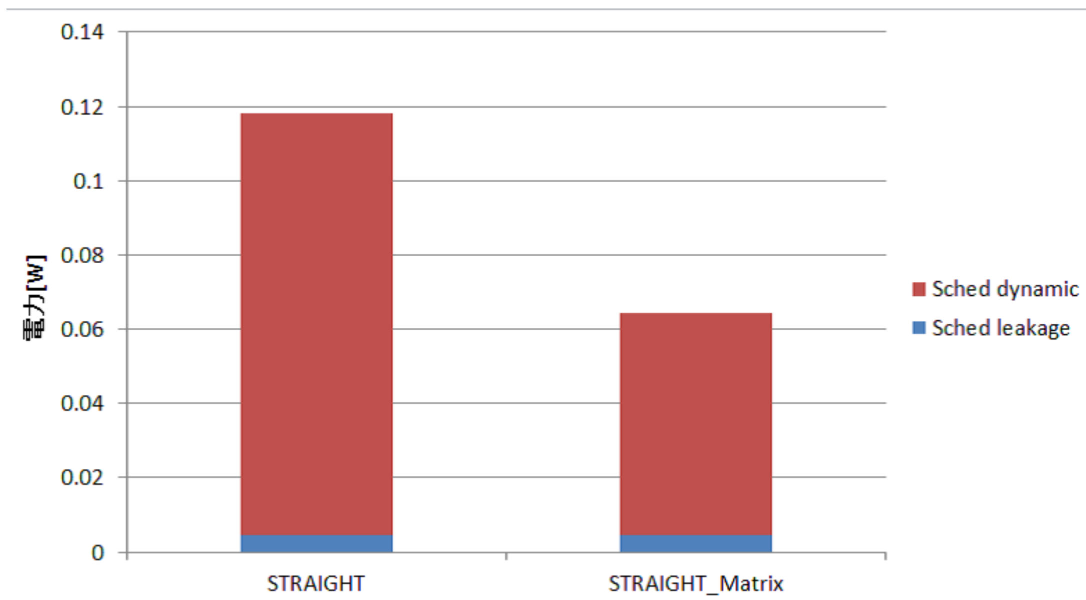

図 5.1.3: スケジューラマトリクス適用による削減効果

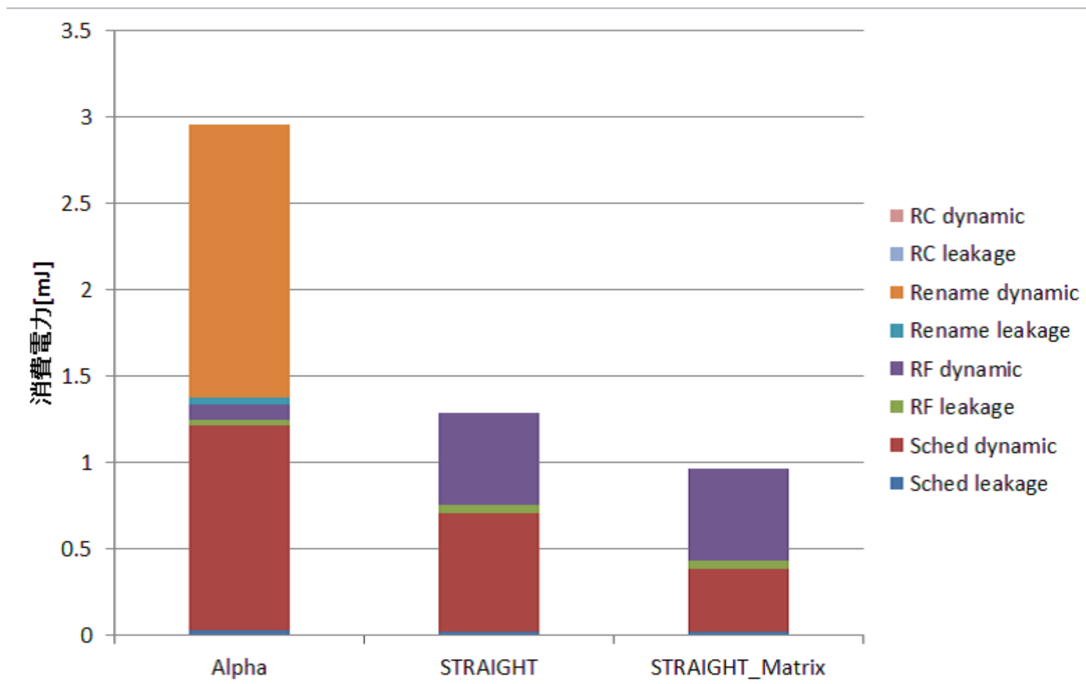

図 5.1.4: スケジューラマトリクス適用による電力削減効果の比較

第6章　STRAIGHTにおけるレジスタキャッシュ

6.1　STRAIGHT用レジスタキャッシュの概要

3章で述べたように，STRAIGHTでは命令の距離による記述を行うため，一度使用した論理レジスタを再度使用する事は無い．この条件からレジスタキャッシュのような小容量のバッファに一度書きこまれたレジスタ番号に対して，再び書きこむ事は無くそのままレジスタキャッシュを追い出されてレジスタファイルへとライトバックされる．すなわち，STRAIGHTアーキテクチャにおいてレジスタキャッシュがリードヒットするときはソースオペランドを指定するときのみである．

また，通常のAlphaのスケールにおいて32エントリのレジスタキャッシュのヒット率は95%程度と知られているがSTRAIGHTアーキテクチャは大きなレジスタファイルを持つため，STRAIGHT用のレジスタキャッシュではAlphaにレジスタキャッシュを実装したときと比べると，必要なレジスタ値を取得しにくくなることが予想される．その結果として従来のレジスタキャッシュに比べてヒット率が下がることが懸念される．そこでSTRAIGHT用のレジスタキャッシュのヒット率を検証する．このヒット率を元にした, レジスタファイルへの削減効果を次の章から議論していく．

6.2　STRAIGHT用レジスタキャッシュの実装

図6.2.1はSTRAIGHTアーキテクチャのコアと，コアを拡大したときのブロック図を示している．

STRAIGHTアーキテクチャはレジスタを大量に保持しているため，そのレジスタの管理のために分散キーバリューストア方式を採用しレジスタを分散ノードとして管理する．

STRAIGHT用のレジスタキャッシュはレジスタ番号とレジスタ値を管理する．またこのときSTRAIGHT用のレジスタキャッシュは各分散ノードであるレジスタファイルに一つずつ実装はせず，一つのレジスタキャッシュにより，すべての分散ノードへとアクセスすることを考慮にいれて実装した. すなわちSTRAIGHTアーキテクチャにおいてレジスタファイルへアクセスする場合には，レジスタキャッシュへアクセスした後に分散キーバリューストア方式に則り分散ノードへとアクセスして，実行している命令が求めるレジスタ値を取得する．次節ではレジスタキャッシュに採用するアルゴリズムについてLRU方式とFIFO方式の2種類を述べる．

6.3　STRAIGHT用レジスタキャッシュの置き換えアルゴリズム

6.3.1　LRU(Least Recently Used)形式の動作

パイプライン処理がレジスタリードを要求する前に，まずはレジスタキャッシュにアクセスする．ここで図6.3.1のような4エントリのレジスタキャッシュに対して，レジスタ値としてA，B，C，Dの4つが保持されている例を上げて動作を説明する．

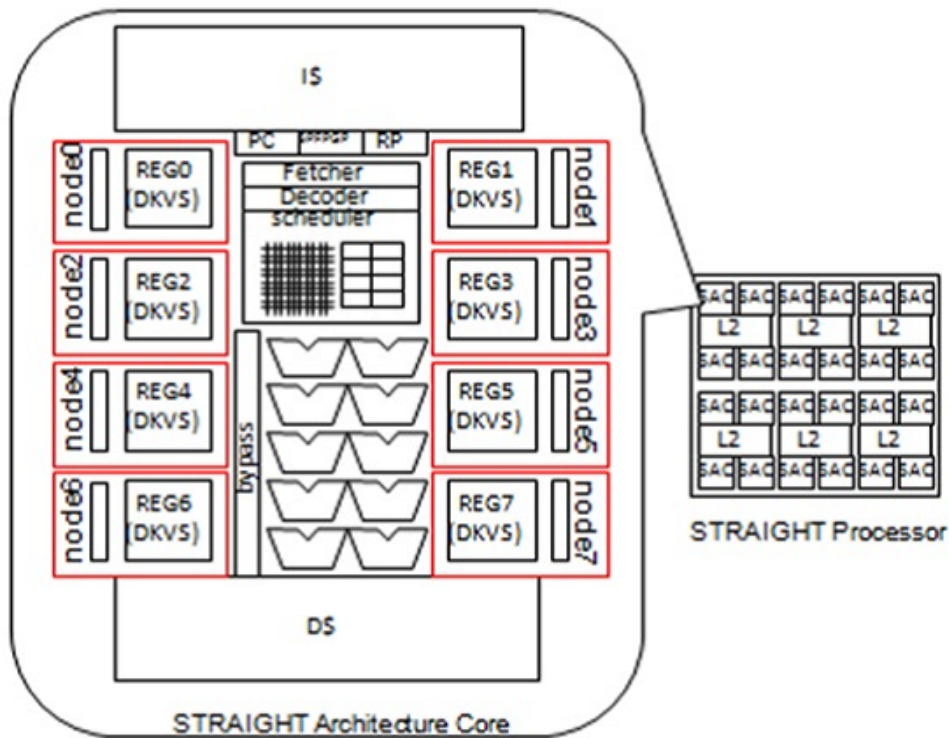

図 6.2.1: STRAIGHT アーキテクチャのブロック図

まずレジスタキャッシュの先頭 (図で言う左側) から，目的のレジスタ値を 1 エントリずつ探していく，例えばレジスタ値が図 6.3.1 の状態 1 において，パイプライン処理がレジスタ値 C を必要としていた場合の動作には

1. 先頭のエントリから順番にレジスタ値 C を探す
2. C を見つけたら先頭のエントリにレジスタ値を移動させる
3. A は 2 番目のエントリに, B は 3 番目のエントリに押し出される

以上のような手順を踏みレジスタ値 C を先頭に移動させて状態 2 のような形にする．このように (i) レジスタキャッシュを読み込んだ際にヒットした場合には，そのヒットしたレジスタ値を先頭のエントリに移動させる．また，一度利用したレジスタに対して再び要求をすることが多い．そのためレジスタキャッシュの先頭に最近利用したレジスタ値を置くことで，レジスタキャッシュに保持させたレジスタ値の中でも新しいレジスタ値が追い出される期間を遅らせる．その結果として，レジスタキャッシュをリードした際のヒット率を向上させることを狙う．仮に状態 2 のときにレジスタキャッシュ上に存在しないレジスタ値 E を要求した場合には状態 3 のようにレジスタ値 E を先頭に置く．これも利用するレジスタ値は直近の命令が利用することが多い性質を利用しており，先頭のエントリに配置する．このとき，最後のエントリに格納されていたレジスタ値はレジスタキャッシュからレジスタファイルへとライトバックされる．すなわち (ii) レジスタリードときにレジスタキャッシュミスをした場合には，レジスタファイルにアクセスをして目的のレジスタ値を利用してレジスタキャッシュの先頭へと書き込み，最後のエントリにレジスタ値がある場合にはレジスタファイルへとライトバックする．

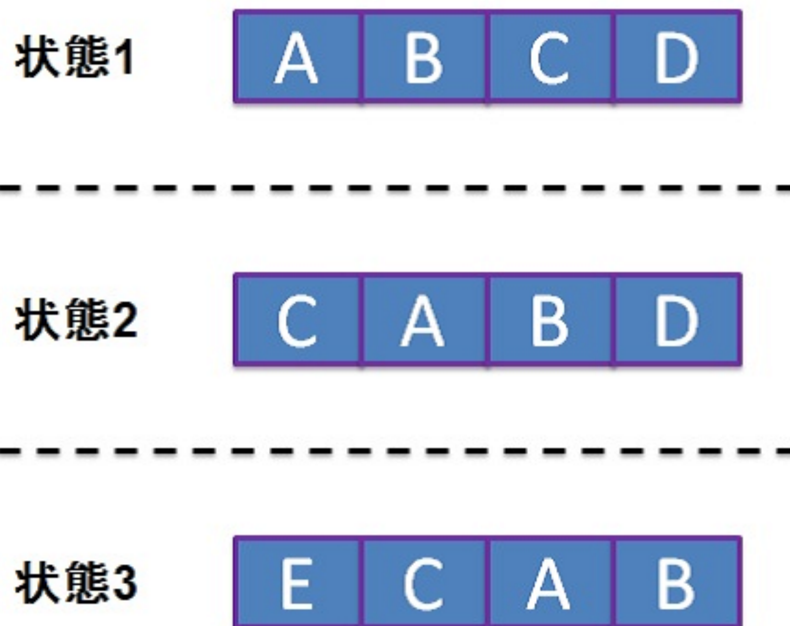

図 6.3.1: レジスタキャッシュの置き換え

パイプライン処理が実行が終わった命令をレジスタに書き戻す処理であるリタイアの際にも，まずはレジスタキャッシュにアクセスする．しかし STRAIGHT はライトワンスマナーに従っているため，レジスタキャッシュに書き込まれているレジスタ番号に対して上書きを行わない．この理由から，レジスタキャッシュに書き込む処理をする場合にはレジスタキャッシュ内を探索せずにそのまま書き込む．すなわち (iii) レジスタライトする場合にはレジスタキャッシュの先頭にレジスタ値を書き込み，最後のエントリにレジスタ値がある場合にはレジスタファイルへとライトバックする．以上より LRU 形式の場合は，先頭のエントリに近いレジスタ値程，命令距離の短い命令が格納されることになる．

6.3.2　FIFO 形式の動作

パイプライン処理がレジスタリードを要求する前に，LRU 形式と同様にレジスタキャッシュにアクセスする．ここでも図 6.3.1 のような 4 エントリのレジスタキャッシュに対して，レジスタ値として A，B，C，D の 4 つが保持されている例を上げて動作を説明する．

まずレジスタキャッシュの先頭 (図で言う左側) から，目的のレジスタ値を 1 エントリずつ探していく．LRU 形式のときと同様にレジスタ値が図 6.3.1 の状態 1 においてレジスタ値 C を必要としていた場合には

1. 先頭のエントリから順番にレジスタ値 C を探す
2. レジスタ値 C を見つけても置き換えは行わない

以上の手順を踏みレジスタ値 C を見付けたときの動作を行う．このように (i) レジスタキャッシュを読み込んだ際にヒットした場合には，中身を置き換えることはしない．すなわち，FIFO 形式の

場合にはレジスタキャッシュに保持させたレジスタ値の中でも新しいレジスタ値が追い出される期間が LRU に比べると短い．

仮に状態 1 のときにレジスタキャッシュ上に存在しないレジスタ値 E を要求した場合にはレジスタ値 E を先頭に置き，既にエントリに保持していたレジスタ値は右方向へと 1 エントリ分移動する．このとき，最後のエントリに格納されていたレジスタ値はレジスタキャッシュからレジスタファイルへとライトバックされる．すなわち (ii) レジスタリードときのレジスタキャッシュミスときの動作は, LRU 形式と同様に先頭にデータを置いていく．

レジスタキャッシュに書き込む動作は LRU 形式と同様に (iii) レジスタライトする場合にはレジスタキャッシュの先頭にレジスタ値を書き込み，最後のエントリにレジスタ値がある場合にはレジスタファイルへとライトバックする．

以上より FIFO 形式の場合は，レジスタキャッシュに保持したレジスタ値は LRU 形式に比べると利用頻度の高い物が残りにくくなるためヒット率が低くなることが予想される．しかしその一方でレジスタ値が直近の命令を利用することから，レジスタキャッシュのエントリ数の増加に従い LRU との差が埋まる可能性を考慮する．

6.4 レジスタキャッシュを採用した際のパイプラインステージ

STRAIGHT アーキテクチャにおいてレジスタキャッシュを採用する理由の主要員としては大型のレジスタファイルにアクセスすることによる遅延の低減の他に，電力削減効果がある. そこで図 6.4.1 のように NORC と同様のパイプラインステージを持たせることでレジスタファイル自体への負荷を低減させる．

図 6.4.1: レジスタキャッシュ適用ときの STRAIGHT パイプライン

次章では以下の点に留意して STRAIGHT の電力削減効果を示す.

1. ヒット率を検証し，レジスタキャッシュに適したアルゴリズムの決定
2. アルゴリズムの決定を受け，費用対効果を踏まえたエントリ数の決定
3. そのエントリ数によるレジスタファイルの電力削減効果

第7章　レジスタキャッシュを適用した際の評価

7.1　評価環境

この章では予備評価と同様に，STRAIGHT アーキテクチャにおいて専用のシミュレータと専用のバイナリを用いた電力評価を行う．またこの時のアーキテクチャパラメタは表 7.1 のようになる．

パイプラインシミュレータ鬼斬を用いて得られた出力と与えたパラメータから入力を作成し，電力シミュレータである McPat を用いた．ただしレジスタキャッシュの電力評価は，レジスタファイルを少ないエントリ数として測定し，レジスタキャッシュとして近似する事で評価を行った．加えてレジスタキャッシュにはヒット，ミスに関係なくアクセスするとした．すなわち，表**??**と表 4.3 において No.46 の値を (1 - レジスタキャッシュのヒット率) をかけ，No.48 の値からレジスタキャッシュのエントリ分引く事でレジスタファイルの削減効果を測定する．レジスタキャッシュ自体の電力は，No.18 から No.22 の value をレジスタキャッシュのエントリ数とすることで測定する．

表 7.1: 電力削減効果策定用アーキテクチャパラメタ

	ベースライン / STRAIGHT
フロントエンド幅	4 / 8
リタイア幅	6 / 12
スケジューラサイズ	int32+fp16 / int128+fp64
レジスタファイル	int128+fp128 / int512+fp512
レジスタキャッシュ	1 ,2 ,4 ,8 ,16 ,32 ,64 ,128
フロントエンドレイテンシ	7cycle / 5cycle
発行幅	int2,fp2,mem2
D1 キャッシュ	64KB,8way,64Bline,3cycle hit latency
I1 キャッシュ	64KB,8way,64Bline,3cycle hit latency
D1 キャッシュ	64KB,8way,64Bline,3cycle hit latency
メインメモリ	200 cycle

7.2　評価結果

7.2.1　ヒット率

LRU 形式と FIFO 形式で置き換えアルゴリズムを行うレジスタキャッシュを STRAIGHT アーキテクチャに適用した場合のリードヒット率は図 7.2.1 のようになる．

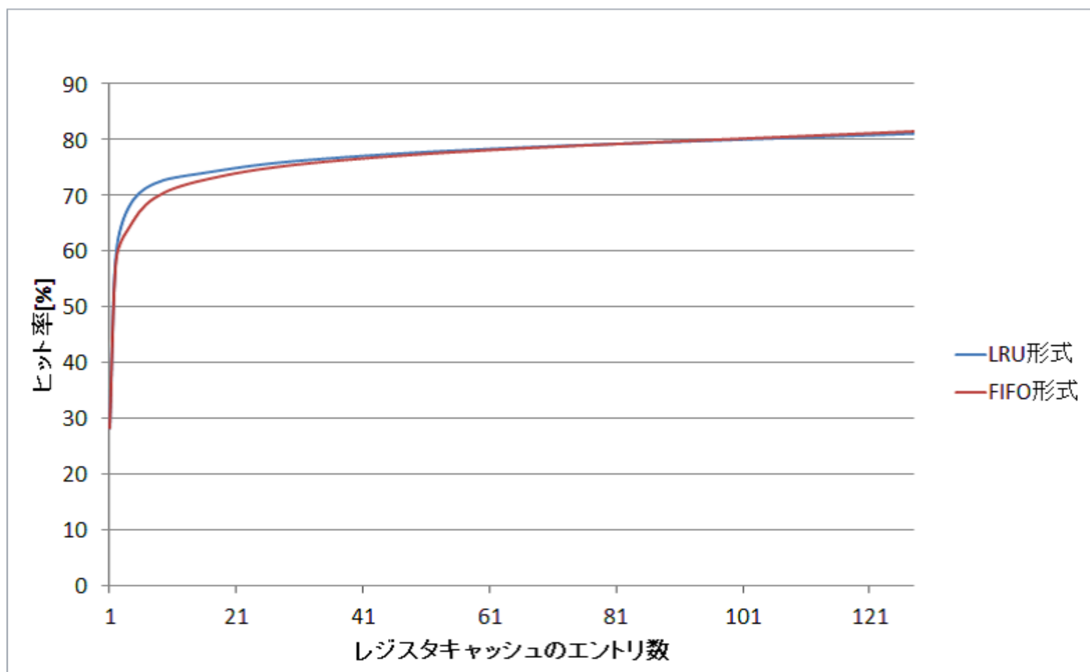

図 7.2.1: レジスタキャッシュのヒット率

第 6 章で述べたようなアルゴリズムによって置き換えがなされる場合に，LRU 形式は FIFO 形式と比較すると 4 エントリの時点でそれぞれ 67.9%と 64.1%になり，8 エントリではそれぞれ 72.1%と 69.5%となる．このように少ないエントリ数では LRU がヒット率に優れているが，16 エントリを境にエントリ数が増えるに従い FIFO 形式とのヒット率の差が縮まっていく，最終的に 128 エントリにおいてはヒット率は逆転する．これは STRAIGHT アーキテクチャのコード記述がライトワンスマナーに従っており，使用するレジスタ値は直近の物を使うことが多いため，少ないエントリ数においては優先度の高いレジスタ値を残す LRU 形式の方がヒット率において上回る．その一方でエントリ数が増えるに従い，そのエントリ数は STRAIGHT の参照制限に迫っていく．その結果レジスタキャッシュを置き換えなくとも必要とするレジスタ値がレジスタキャッシュ内に残る．加えて置き換えを行うことで，今現在は優先度が低いが将来的に何度も利用するようなレジスタ値は最後に近いエントリに移動していく．その結果 LRU 形式のように置き換えを行うことで，そのようなレジスタ値が追い出されてしまう．以上の理由から，特別な置き換えを行わない FIFO 形式が 128 エントリにおいてヒット率の逆転が生じたと考えられる．

STRAIGHT アーキテクチャに用いるレジスタキャッシュは特にレジスタファイルの電力の最適化を目的としている．128 エントリにおいて FIFO 形式が LRU 形式にヒット率が追い付くが，大きなエントリ数に見合う程のヒット率は得られない．また，レジスタキャッシュのエントリ数が 128 エントリになることで，レジスタキャッシュ自身の電力が増大してしまうことが予測される．そのためレジスタキャッシュ自体の電力が抑えられる少ないエントリ数において，より効果の高い LRU 形式を STRAIGHT アーキテクチャ用レジスタキャッシュの置き換えアルゴリズムとして採用する．次節では LRU 形式によるレジスタファイルの電力削減効果を示し，STRAIGHT アーキテクチャに適したレジスタキャッシュのエントリ数を策定する．

7.2.2　レジスタファイルの電力削減効果

前節で議論したLRU形式でレジスタキャッシュを導入する．そのときに，予備評価で得られたレジスタファイルの1サイクル辺りの電力に対しての削減効果は図7.2.2のようになる．

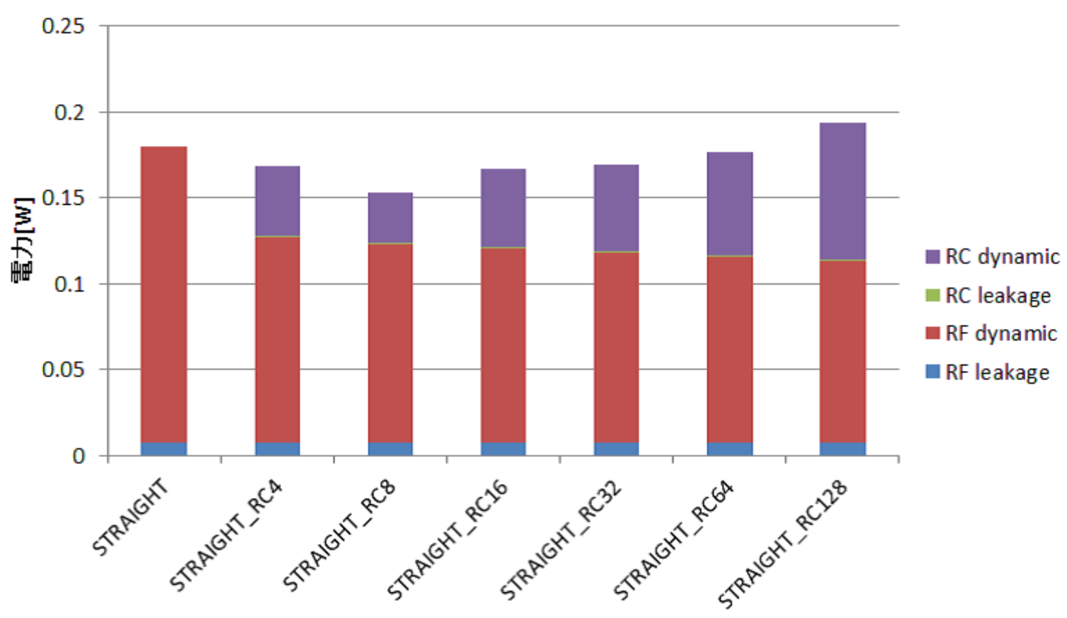

図7.2.2: LRU形式によるレジスタファイルの削減効果

この図は横軸にレジスタキャッシュを適用していないSTRAIGHT，4エントリレジスタキャッシュを適用したSTRAIGHT，8エントリレジスタキャッシュを適用したSTRAIGHT，16エントリレジスタキャッシュを適用したSTRAIGHT，32エントリレジスタキャッシュを適用したSTRAIGHT，64エントリレジスタキャッシュを適用したSTRAIGHT，128エントリレジスタキャッシュを適用したSTRAIGHTとなっている．また縦軸に1サイクル辺りの電力を示しており，レジスタファイルの読み書きによる電力RF dynamic，RF leakageとレジスタキャッシュの読み書きにより生じる電力RC dynamicとRC leakagを示している．

各エントリ数に対して，レジスタファイル自体の消費電力を最も削減したのは128エントリで，消費電力を38.7%削減している．しかしレジスタキャッシュ自体の消費電力も同時に上昇するためにレジスタキャッシュを含めたときの総合的な電力を考慮すると，128エントリのレジスタキャッシュを実装した場合，消費電力は元のSTRAIGHTアーキテクチャに比べて7.4%増加する．STRAIGHTアーキテクチャにおいてレジスタファイルの最適化する場合は，図**??**から8エントリのレジスタキャッシュを用いることで，レジスタキャッシュを含めた消費電力を最も抑えることが可能となる．この8エントリのレジスタキャッシュを適用することで，元のSTRAIGHTアーキテクチャの消費電力と比較して，14.1%削減する効果が示された．

STRAIGHTアーキテクチャにレジスタキャッシュを適用した場合と元のSTRAIGHTアーキテクチャ，そしてAlphaの同じ処理による消費電力を比較した場合には図7.2.3のようになる．

レジスタファイル自体の電力，すなわちレジスタキャッシュを除いた電力は33.1%削減となり，既存研究であるNORCの削減効果と比較すると控えめな削減効果となっている．

STRAIGHTにレジスタキャッシュを適用した場合，ヒット率に従ってレジスタファイルへのリー

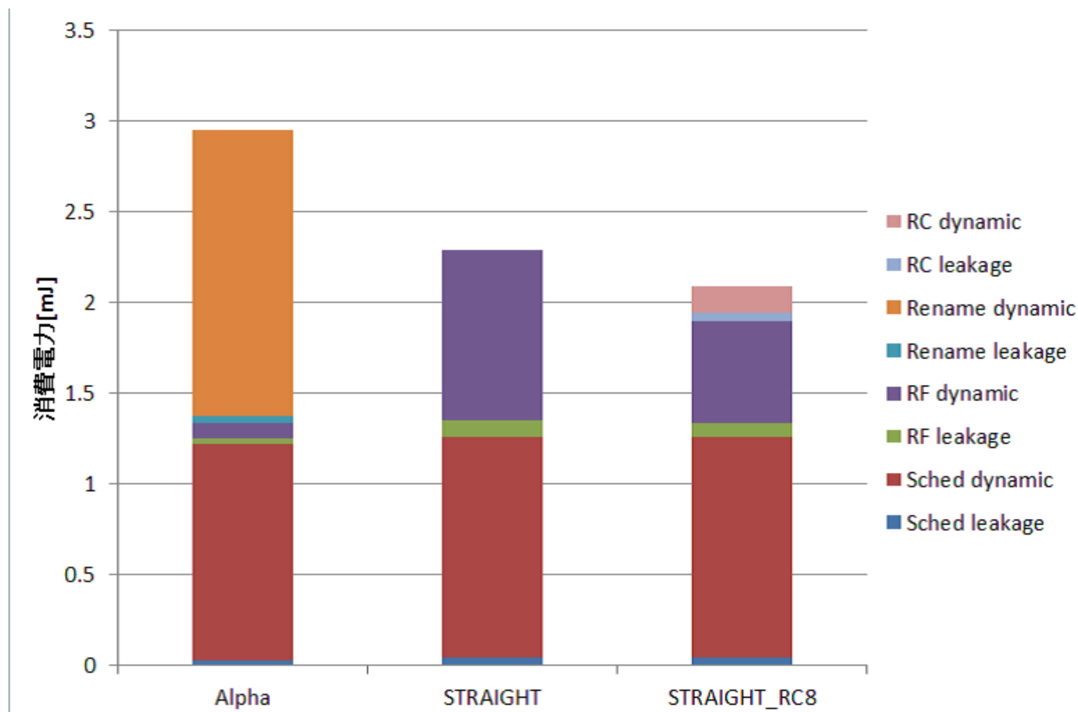

図 7.2.3: レジスタキャッシュを適用した電力評価

ドアクセスは削減出来る．その一方ではレジスタファイルへのライトバックにおいてはエントリ数だけ削減することしか出来無い．前の章でも述べたように，ライトワンスマナーに従うためレジスタキャッシュに一度書きこみをしたレジスタ番号に対して，再び書きこみを行うことはない．すなわちエントリ数だけはライトバック数を削減出来るが，以後は全ての命令をライトバックして行く．そのため，レジスタキャッシュライトの時の電力がオーバヘッドとなってしまい，レジスタファイルへのライトバック数が増えてしまうために電力削減効果が従来研究に比べて比率においては少なくなる．

レジスタファイルへの削減率で比較した場合には，STRAIGHT 用レジスタキャッシュは従来研究と比べて控えめであるが，電力の削減面において考慮すると図 7.2.2 から 8 エントリレジスタキャッシュの適用により，約 0.05W 削減している．その一方で図 4.2.1 における Alpha のレジスタファイルに対して NORC を適用したと仮定すると，レジスタファイル電力は約 0.005W 削減することが見積られる．すなわち，電力自体の削減は STRAIGHT アーキテクチャにレジスタキャッシュを適用した場合 10 倍ほどの差が生じる．

第8章　議論

前章までで，本研究の提案手法である STRAIGHT 向けの LRU 形式レジスタキャッシュとマトリクススケジューラの適用による見積りを行ってきた．ここで，従来のスーパースカラプロセッサである Alpha と我々が提案している STRAIGHT，そして提案手法を適用した STRAIGHT である Matrix_RC8 の同じ処理を行う際の消費電力を図 8.0.1 に示す．

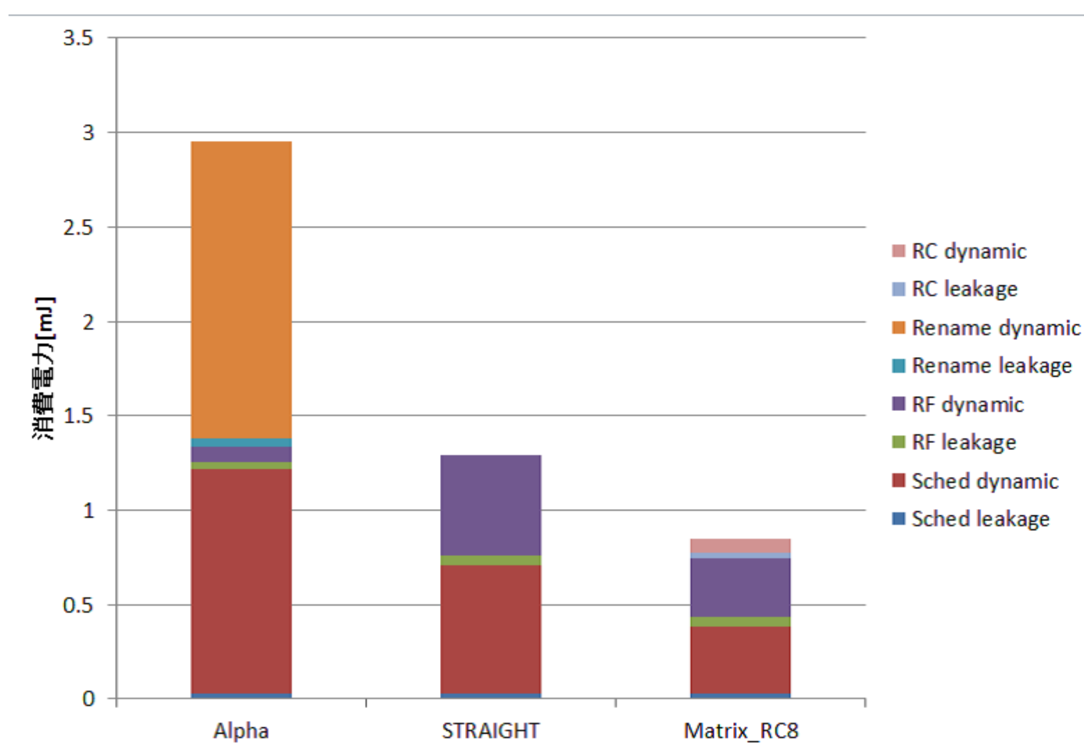

図 8.0.1: 提案手法の適用による削減効果

この図 8.0.1 ではキャッシュを除いた実行部分の電力を比較しており，これまでの評価と同様にリネーム処理である Rename dynamic，レジスタファイルの読み書きにより発生する RF dynamic，スケジューラの利用による Sched dynamic，そしてレジスタキャッシュの読み書きにより生じる RC dynamic を示している．まず始めに STRAIGHT の消費電力は Alpha と比較して 56.4%削減している．序論でも述べた様に，ダークシリコン問題によりチップ上のトランジスタは半分が同時駆動できなくなることが見積られている．ここで STRAIGHT は従来の Alpha と同じ処理を半分以下の消費電力で駆動する事が可能である．すなわち，チップに与える電力が同じでも STRAIGHT であれば倍以上のトランジスタを活用することが可能である．加えて，STRAIGHT は性能が Aplha から 3 倍程度向上しているため，電力辺りの性能が Alpha よりも 6 倍ほど向上している．

以上より，STRAIGHT アーキテクチャのコンセプトである低電力かつ高いシングルスレッド実行能力を有することが示された．ここで図 8.0.1 の Matrix_RC8 を見ると，8 エントリのレジスタキャッシュとマトリクススケジューラを STRAIGHT に導入することで，性能/電力比を向上させた STRAIGHT から更に 33.9%削減している．従来の Alpha と比較すると 71.2%削減する結果となり，提案手法を適用した STRAIGHT であれば Alpha と同じ処理を Alpha の約 3 割ほどの消費電力で実行することが示された．

第9章　結論

本論文では性能/電力比を向上させるために, トランジスタ資源をレジスタ容量に充てることで制御を軽量化するSTRAIGHTアーキテクチャを提案し, STRAIGHTアーキテクチャにおける電力評価と, 電力面でのオーバヘッドの最適化について述べてきた. 同じ命令数に実行サイクル数を反映させた消費電力は, 従来のスーパースカラプロセッサと比較して56.4%削減した. 更にSTRAIGHTのオーバヘッドであるレジスタファイルに対してはレジスタキャッシュを適用する事で, レジスタファイルのアクセス回数を低減させることで最適化を狙い, スケジューラに対してはスケジューラマトリクスを適用することで, 大規模スケジューラにおけるクリティカルパスを削減した. STRAIGHTの消費電力は8エントリのレジスタキャッシュにより, レジスタファイルの消費電力は33.1%削減し, スケジューラマトリクスを適用することで47.4%削減することを示した. また双方利用することによってアーキテクチャ全体で最適化を行っていないSTRAIGHTアーキテクチャと比較して33.9%削減することが可能であり，Alphaと比較すると消費電力を71.2%削減した.

今後の課題としては, STRAIGHT専用のコンパイラを開発し, レジスタキャッシュを適用させた場合の性能の影響について評価することである.

謝辞

本論文の執筆において多数の助言を頂き，指導をして下さった主任指導教員である入江 英嗣 准教授に感謝いたします．そして私の研究に対して貴重な意見をして下さった吉永 努教授と吉見 真聡助教授を頂いたため，よりよい研究となったことに感謝いたします．また自身の研究を進めるに辺り，様々な指摘を頂いた研究室の皆様の方々にも感謝を申し上げます．本研究は，科研費若手研究 25730028「新アーキテクチャによる高効率プロセッサコアおよびそのマルチコア構成の研究」の助成を受けました．

参考文献

[1] Dennis Sylvester and Kurt Keutzer. Getting to the bottom of deep submicron. In *ICCAD'98*, pp. 203 – 211, 1998.

[2] Grot B. Ferdman M. Volos S. Kocberber O. Picorel J. Adileh A. Jevdjic D. Idgunji S. Ozer E. Lotfi-Kamran, P. and B Falsafi. Scale-out processors. In *Int. Symp. on Computer Architecture*, 2012.

[3] Weaver V. Bhadauria, M. and S. McKee. Understanding parsec performance on contemporary cmps. In *Int. Symp. on Workload Characterization*, pp. 99 – 107, 2009.

[4] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *Micro, IEEE*, Vol. 32, No. 3, pp. 122 – 134, 2012.

[5] 佐保田 誠 吉見真聡 吉永努. 入江英嗣. もし ilp プロセッサのレジスタファイルが分散キーバリューストアになったら. 情報処理学会研究報告. 計算機アーキテクチャ研究会報告, pp. 1 – 10, 2013.

[6] 縣 亮慶 中島 康彦 森 眞一郎 北村 俊明 富田 眞治五島 正裕. Dualflow アーキテクチャの命令発行機構. Vol. 42, No. 4, pp. 652 – 662, 2000.

[7] 入江英嗣 五島正裕 坂井修一一林宏憲. 逆 dualflow アーキテクチャ. 情報処理学会論文誌コンピューティングシステム, 第 1 巻, pp. 22 – 33, 2008.

[8] Masahiro Goshima Ryota Shioya and Hideki Ando. A front-end execution architecture for high energy efficiency. In *Int'l Symp. on Microarchitecture*, 2014.

[9] N. P. Jouppi S. Palacharla and J. E. Smith. Quantifying the complexity superscalar processors. *Technical report, Univ. of Wisconsin-Madison*, 1996.

[10] Kitamura Toshiaki Nakashima Yasuhiko Tomita Shinji Goshima Masahiro, Nishino Kengo and Mori Shin-ichiro. A high-speed dynamic instruction scheduling sheme for superscalar processors. In *MICRO 2001*, pp. 225 – 236, 2001.

[11] Edward Brekelbaum Gabriel H. Loh Bryan Black Peter G. Sassone, Jeff Rupley. Matrix scheduler reloaded. In *Int. Symp. on Computaer architecture*, pp. 335 – 346, 2007.

[12] Neil C. Wilhelm Robert Yung. Caching processor general registers. In *1995 IEEE Int. Conf.*, pp. 307 – 312, 1995.

[13] Masahiro Goshima and Shuichi Sakai Ryota Shioya, Kazuo Horio. Register cache system not for latency reduction purpose. In *Int. Symp. on Microarchitecture*, pp. 301 – 312, 2010.

[14] 坂井修一塩谷亮太. プロセッサ・シミュレータ「鬼斬弐」の設計と実装. 2009.

[15] Ahn J.-H. Strong R. Brockman J. Tullsen D. Li, S. and N Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Int. Symp. on Microarchitecture*, pp. 469 – 480, 2009.