修 士 論 文 の 和 文 要 旨

研究科・専攻	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　博士前期課程		
氏　　名	佐保田　誠	学籍番号	1352012
論 文 題 目	プロセッサアーキテクチャ「STRAIGHT」のシミュレータ設計と評価		

要　旨

パーソナルコンピュータやスマートフォンを始めとする多くの情報機器の中核部品であるマイクロプロセッサは，現在の情報化社会を支えている．技術の発展により半導体の微細化が進み，パッケージ内で利用可能なトランジスタの数は増加しているが，チップ上の回路の消費電力は，プロセスの微細化ほどスケールダウンしていない．そのため，プロセスを微細化する度により多くのコアをパッケージ内に搭載することができるようになるが，チップ全体の消費電力もそれに従い増大する．それによって，パッケージあたりの電力や熱の制限から，搭載されたトランジスタを同時に駆動することができない，ダークシリコン問題が指摘されている．この問題を解消するためには，従来のアーキテクチャよりもシングルスレッド実行能力に優れ，なおかつ同じより少ない消費電力で同じ処理を実行できる新しいアーキテクチャが求められている．我々は，シングルスレッド能力の向上と消費電力の削減を同時に達成するアーキテクチャとして，広大なレジスタ空間を持つ STRAIGHT アーキテクチャを提案している. STRAIGHT アーキテクチャはライト・ワンス・コードの実行を想定した広大な論理レジスタ空間を持ち，従来のアウト・オブ・オーダ・プロセッサにおいて主要な電力オーバヘッドであったレジスタ・リネーミングやフリーレジスタの管理，レジスタの解放管理を必要としない．また，このライト・ワンス・コードによって，従来困難であった命令ウィンドウサイズやフロントエンド幅の軽量な拡張が可能になり，シングルスレッド性能を向上させる．

本論文では，シミュレータの設計，アセンブラの構築，専用コードの生成，生成したコードを用いた評価・議論を行った．生成した STRAIGHT 専用の Livermore Loop コードを STRAIGHT アセンブラの入力とすることで，シミュレータの入力となる STRAIGHT バイナリを生成することが可能になり，この STRAIGHT バイナリを，設計した STRAIGHT シミュレータの入力とすることで STRAIGHT アーキテクチャを詳細に評価した．評価では，従来の Alpha アーキテクチャと比べて STRAIGHT アーキテクチャは IPC を最大で 88%，平均で 29%向上させた．さらに，同じ処理を行うために必要な命令数について, 従来の Alpha アーキテクチャと比べて STRAIGHT アーキテクチャは最大で約 90%，平均で約 55%の命令を削減することができた．IPC と 1000 ループの必要命令数を相乗した性能については，Alpha に対して STRAIGHT は最大で 12.5 倍，平均で約 3 倍の性能であることがわかった．

平成２６年度修士論文

プロセッサアーキテクチャ「STRAIGHT」のシミュレータ設計と評価

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 :　1352012
氏 名 :　佐保田 誠
主任指導教員 : 吉永 努 教授
指 導 教 員 :　入江 英嗣 准教授
指 導 教 員 :　小川 朋宏 准教授
提出年月日 :　平成２７年１月２６日

(表紙裏)

目 次

図 目 次

表 目 次

第1章　序論

パーソナルコンピュータやスマートフォンを始めとする多くの情報機器の中核部品であるマイクロプロセッサは，現在の情報化社会を支えている．技術の発展により半導体の微細化が進み，パッケージ内で利用可能なトランジスタの数は増加している．アーキテクチャ技術はこのトランジスタの増加をチップ全体の性能向上に結び付けてきた．

2000 年頃から，トランジスタの資源を利用することでパイプライン段数を拡張し，そのパイプラインスロットを各種投機実行によって埋めるというそれまでのアプローチが消費電力の制約を受けることとなり，配線遅延や熱・電力といった問題が指摘されるようになった [3]．さらに，同時に命令を実行し性能を向上させるため，アーキテクチャの発行幅や実行幅などのパイプライン幅を確保した場合でも，並列に実行可能な命令が存在しないという，ILP(Instruction Level Parallelism) の限界が指摘されるようになった．熱・電力の問題と ILP 限界の問題によって，アーキテクチャ設計の転換が必要となった．これを転機とし，プロセッサの実行部分の強化にあてられていた増加するトランジスタ資源は，コア数の増加やオンチップネットワーク，メモリ帯域の増加にあてられるようになり，これらがプロセッサの性能向上につながっている [4][5].

しかし，コア数を増加させるプロセッサの成長戦略は，コア数の増加とともに，TLP (Thread Level Parallelism) や性能向上の限界が指摘されるようになってきた [6]．十分に並列化されているワークロードであっても，コア毎のスレッド長に偏りがあり，スレッド同期のオーバヘッドが存在するため，コア数が増加するほどシングルスレッド実行能力が全体性能を制限することとなる．また，チップ上の回路の消費電力は，プロセスの微細化ほどスケールダウンしない．微細化する毎に CPU コアは小さくなるが，一つ一つのコアが消費する電力はそれに見合うほど小さくならないのだ．そのため，プロセスを微細化する度により多くのコアをパッケージ内に搭載することができるようになるが，チップ全体の消費電力もそれに従い増大する．それにより，パッケージあたりの電力や熱の制限から，搭載されたトランジスタを同時に駆動することができない，ダークシリコン問題 [7] が指摘されている．この問題を解消するために，メニーコアプロセッサを構成するコア一つ一つのアーキテクチャの転換が求められる．従来のアーキテクチャよりもシングルスレッド実行能力に優れ，なおかつより少ない消費電力で同じ処理を実行できる新しいアーキテクチャが求められている．

我々は，シングルスレッド実行性能の向上と消費電力の削減を同時に達成するアーキテクチャとして，広大なレジスタ空間を持つ STRAIGHT アーキテクチャを提案している [8][1]．STRAIGHT アーキテクチャはライト・ワンス・コードの実行を想定した広大な論理レジスタ空間を持ち，従来のアウト・オブ・オーダ・プロセッサにおいて主要な電力

オーバヘッドであったレジスタ・リネーミングやフリーレジスタの管理，レジスタの解放管理を必要としない．また，このライト・ワンス・コードによって，従来困難であった命令ウィンドウサイズやフロントエンド幅の軽量な拡張が可能になり，シングルスレッド性能を向上させる．さらに，このレジスタファイルに分散キー・バリュー・ストア技術を適用することで，より効率的にハードウェアを構成することができる．

プロセッサシミュレータ「鬼斬弐」[9] と STRAIGHT アーキテクチャに見立てたアーキテクチャパラメタを用いた初期評価では，従来のアーキテクチャと比べ，同じワークロードに対するエネルギー消費を 12%削減しながら，同時に約 30%の IPC(Instructions Per Cycle) 向上が得られ，性能/パワー比を改善する新しい実行方式として有効であることが示された．しかしながら，この評価では特殊レジスタやレジスタリネーミングの排除をはじめとする STRAIGHT アーキテクチャの特徴を再現できていない．より正確に STRAIGHT アーキテクチャを評価するためには，既存のプロセッサシミュレータに STRAIGHT アーキテクチャに見立てたアーキテクチャパラメタを用いるのではなく，ライト・ワンス・コードを実行し，レジスタリネーミングやフリーレジスタの管理，レジスタの開放管理を省略した専用のシミュレータを設計し，それを用いたシミュレーションを行うことが必要である．また，この専用シミュレータが実行するライト・ワンス・コードを用意する必要がある． そこで本研究の目的は，STRAIGHT アーキテクチャの専用シミュレータを設計・実装し，STRAIGHT コードを生成することで，より詳細に STRAIGHT アーキテクチャを評価することである．

本研究では，

- STRAIGHT シミュレータの設計
- STRAIGHT アセンブラの構築
- Livermore Loop の STRAIGHT コード生成
- 設計したシミュレータ，生成したコードを用いた IPC と必要命令数の評価
- STRAIGHT アーキテクチャと STRAIGHT コードに関する議論

を行った．生成した STRAIGHT 専用の Livermore Loop コードを STRAIGHT アセンブラの入力とすることで，シミュレータの入力となる STRAIGHT バイナリを生成することが可能になった．この STRAIGHT バイナリを，設計した STRAIGHT シミュレータの入力とすることで STRAIGHT アーキテクチャを従来より詳細な評価を行った．また，STRAIGHT シミュレータのベースであるプロセッサシミュレータ鬼斬弐とクロスコンパイラを使用することで従来の Alpha アーキテクチャにおいても Livermore Loop コードを用いた評価を行った．STRAIGHT アーキテクチャと従来のアーキテクチャそれぞれの実行結果を基に比較評価を行った．Livermore Kernel を使用した評価では，従来の Alpha アーキテクチャと比べて STRAIGHT アーキテクチャは IPC を最大で 88%，平均で 29%向上させた．さらに，同じ処理を行うために必要な命令数について，従来の Alpha アーキテクチャと比べて STRAIGHT アーキテクチャは最大で約 90%，平均で約 55%の命令を削減すること

ができた．IPC と 1000 ループの必要命令数を相乗した性能については，Alpha に対して STRAIGHT は最大で 12.5 倍，平均で約 3 倍の性能であることがわかった．また，評価を基にした STRAIGHT アーキテクチャと最適な STRAIGHT コードに関する議論を行った．ループアンローリングを用いた評価では，STRAIGHT コードにその最適化を適用することで IPC が従来以上に向上することが分かった．本論文の評価では Livermore Kernel 5 にループアンローリングを適用することで，何もしない Kernel 5 と比べ IPC が最大 46%向上した．また，ループアンローリングを外側ループに 10 段，内側ループに 2 段適用した場合，STRAIGHT アーキテクチャではレジスタ移動命令 RMOV を命令数全体の 26%から 4%まで削減し，ソースオペランドの変位が 32 以上の命令数をゼロから命令数全体の 46%まで向上した．IPC と 1000 ループの必要命令数を相乗した性能については，Alpha に対して STRAIGHT は最大で 12.5 倍，平均で約 3 倍の性能であることがわかった．

本論文は以下のように構成する．2 章で関連研究を紹介する．3 章で STRAIGHT アーキテクチャについて述べ，4 章では STRAIGHT シミュレータの設計について述べる．5 章では STRAIGHT エミュレータについて述べ，6 章でパイプラインシミュレータについて述べる．7 章で Livermore Kernel について述べ，8 章 STRAIGHT アーキテクチャの評価を行い，9 章で議論し，10 章で結論を述べる．

第2章　関連研究

2.1　ダークシリコン問題に対する関連技術

2.1.1　ダークシリコン問題

Esmaeilzadeh らは，マルチコア・スケーリングに関して，パッケージあたりの電力や熱の制限から，搭載されたトランジスタを同時に駆動することができない，ダークシリコン問題を指摘している [7]．2005 年以来，プロセッサの設計者はシングルコアの性能に焦点を当てるのではなく，ムーアの法則のスケーリングを利用するためにコア数を増加させてきたことを背景としている．トランジスタが微細化してもそれらの電力密度は一定のままであり，電力使用量は面積に比例するというデナード・スケーリングが適合しなくなり，シングルコア・スケーリングが縮小されると同時にマルチコア・スケーリングを制限することになると述べている．彼らはこの研究で，でデバイス・スケーリング，シングルコア・スケーリング，マルチコア・スケーリングを組み合わせることにより，並列ワークロードのセットにおいての速度向上の可能性を測定し，マルチコア・スケーリングをモデリングし限界があることを示している．

Esmaeilzadeh らの評価では，22nm プロセスであったとしてもチップ全体の 21%の領域がダークシリコン問題によって同時に駆動することができない (パワーオフ) 領域になり，8nm プロセスではパワーオフ領域がチップ全体の 50%まで増えるという．さらに 2024 年までに，一般的な並列ワークロードに対しての速度向上は平均で 7.9 倍までしか可能でないとし，目標となる性能向上には程遠いと述べている．

2.1.2　電力の動的管理

電力制約の増大によるダークシリコン問題に対する技術として，Leverich らは PCPG(Per-Core Power Gating) を提案している [10]．PCPG は，コアごとに電力を制御する手法である．選択されたコアに対してのみ電力供給を行うことで，電力が供給されていないコアのリーク電力をほぼゼロにすることが可能だと述べられている．商用の 4 コアチップを想定した評価では，大きな性能オーバヘッド無しにプロセッサの消費電力を最大 40%削減できると述べている．

パイプライン全体に適応する資源制御の技術 [11][12][13][14] は，動的な電力の低減に有効であることが示されたが，多くの電力制御回路が必要であり容易に適応することができな

い．また，コア･ハードウェアのサブセットのみを制御する技術 [15][16][17][18][19][20][21][22] は消費電力の削減を制限してしまう．

Paula らは，ダークシリコン問題に対する技術として，動的適応型アーキテクチャFlicker を提案している [23]．この論文では PCPG の問題点として，

- 電力制御が粗く，電力要件に合わせて正確に制御するのが困難
- パワーオンのコアに電力を均等に割り当てるため，ワークロード特性にハードウェアを適応できない
- OS のスケジューラが，利用可能なコアの様々な数に対応する必要がある

を挙げている．

そこで Paula らは，各コアにおいて各パイプラインセクションを再構成可能なレーン (パイプラインの水平スライス) に分割する Flicker を提案している．Flicker の各レーンは，個々に制御可能な電源ドメインで構成されている．そのため PCPG と比べて，パイプライン資源の必要性に応じてアプリケーションへのより細かい電力割り当てを可能にしている．しかし，Flicker システムの問題点として数十のコアでは，PCPG よりも遥かに困難なグローバル最適化問題がある．サンプリング技術を使用したグローバル最適化を行った後の評価では，高電力の割り当て時には，PCPG が有効で Flicker をわずかに上回るとしながらも，厳しい電力制約 (55%) のもとでは，Flicker は PCPG より優れ，平均 27%の性能向上を示している．

Haghbayan らは，消費電力の上限 (Thermal Design Power(TDP) と呼ばれる) を考慮し，PID(Proportional Integral Derivative) と呼ばれるコントローラベースの動的な電力管理方法を提案している [24]．動的なワークロードを実行するメニーコアシステムにおいて，TDP の制約違反を避けるために，PID はニア・スレッショルド・オペレーションを含む，細かい DVFS(Dynamic Voltage and Frequency Scaling) を提供する．加えてこの手法はアプリケーションを，ハード・リアルタイム，ソフト・リアルタイム，ノー・リアルタイム制約で区別し，適切な優先度で扱うことができる．ハード・リアルタイム，ソフト・リアルタイム，ノー・リアルタイムで分類されるアプリケーションを，ランダムで選び動的なワークロードとしたシミュレーションでの評価では，従来の TDP スケジューリング方式と比べてシステム全体のスループットを 43%以上向上させており，この方法が TDP 制約の適合に有効だと述べている．

Pagani らは，TDP のような一定値を電力制約として用いた場合，メニーコアシステムに大きな性能損失をもたらすと述べている [25]．より良い電力配分技術がダークシリコン問題に対処するための大きな一歩であるとし，同時に動作するコア数の関数として電力制約値を提供する，Thermal Safe Power(TSP) と呼ばれる電力制約の概念を提案している．TSP 未満の任意の消費電力でコアを実行すると、動的な熱管理はトリガされないことが保証される．TSP はワーストケースではオフライン，コアのマッピングにはオンラインとなる．評価では，gem5，McPAT，HotSpot が用いられ，6 つの異なる電力制約のもとでシミュレーションが行われている．従来の電力制約を使った場合と比べ TSP を使用した場合，平均でチップ当たり 50.5%，コア当たり 14.2%高い性能を得ている．TSP は一定の電

力制約を使用した推定よりも，ダークシリコンの推定に関して優良であると結論付けている．

2.1.3 命令ウィンドウの動的リサイジング

メモリを集中的に使用するプログラムでは，プロセッサとメインメモリ間に大きな速度の食い違いがあるため，シングルスレッド性能を上げることが困難である．アウトオブオーダ実行によってメモリレベル並列性 (MLP) を利用することは可能であるが，その問題点として大きな命令ウィンドウが必要なことと，単純に拡大するとクロックサイクル時間の低下につながることがある．命令ウィンドウを大きくする手法としてリソースのパイプライン化があるが，それにより命令発行が遅れ，命令レベル並列性 (ILP) の効果的な利用を妨げ，計算負荷の高いプログラムの性能が劇的に低下する．

そこで Kora らは，MLP が利用可能であるかどうかを，ラストレベルキャッシュ・ミスの発生に基づいて予測することにより，命令ウィンドウのリソースサイズを変更する方式を提案している [26]．MLP が利用可能である場合は，命令ウィンドウリソースの拡大とパイプライン化を行い，ILP が利用可能である場合は，命令ウィンドウリソースの縮小とパイプライン化の解除を行う．

彼らの評価では，この動的リサイジングを適応したアーキテクチャは従来のプロセッサコアに対して平均で 21%の性能向上を示しており，追加コストは従来のプロセッサコアに対して 3%，プロセッサチップ全体で 3%となっている．

2.1.4 エネルギー効率に関する研究

Czechowski らはアーキテクチャの革新が，どのようにビッグコアに関連する非効率性を軽減することができるか，その実例を提供している [27]．彼らは，実験的に現行のプロセッサ上で命令レベルのエネルギー効率を測定するための方法を提示することを目標としており，これにより，アーキテクチャの機能だけでなく，性能と消費電力の関係を研究するための新規な技術を分離している．アーキテクチャ機能を分離することにより，エネルギー効率に影響を与える変動コストを示している．

Czechowski らは人工的にカーネルの命令レベルの依存関係を変更するため，レジスタ・スクランブリングという新たな技術を用いている．レジスタ・スクランブリングでは，ランダムに各命令に新しいレジスタ番号が割り振られる．レジスタ・スクランブリングによって，命令レベルの並列性に変化が生じ，性能と消費電力に変化が生じる．一つのカーネルについて，異なるいくつかのスクランブルを生成することによって，性能の電力に関する回帰を生成することができる．この回帰は，どのくらい性能の変化がエネルギー効率に影響を与えるかを推定することで，コアアーキテクチャの性能向上の評価に使用することができる．

評価には，ベンチマークとして Livermore Loop Kernel，プロセッサアーキテクチャとして Penryn，Nehalem，Westmere，Sandy Bridge，Ivy Bridge， Haswell の 6 世代にわたる

アーキテクチャ技術が使用されている．結果として，Penryn に対して Haswell は，平均で 2.9 倍のエネルギー効率が得られている．また，一つの命令を実行するのに必要な電力 EPI(Energy per instruction) が IPC に大きく依存すると結論付けている．

2.2 Dualflow アーキテクチャ

2.2.1 Dualflow アーキテクチャ

命令のスケジューリング・ロジックを簡単化するため，レジスタ・リネーミングを行わない命令セット・アーキテクチャとして，五島らは Dualflow アーキテクチャを提案している [28][29][30]．Dualflow アーキテクチャでは，通常の制御駆動型の命令セット・アーキテクチャにあるようなレジスタを定義しない．その代わりに，プロデューサと呼ばれるデータを生産する命令と，コンシューマと呼ばれるデータを消費する命令を指定することで，明示的にデータを受け渡す．

Dualflow アーキテクチャは，命令スケジューラ・ロジックを簡単化し，処理を高速化するために提案された．しかし，プロデューサがコンシューマを明示的に指定することができるので，レジスタ・リネーミングの処理が不要になるというメリットがある．Dualflow アーキテクチャの命令列は，一つ一つがレジスタ・リネーミング済みの命令であるとみなすことができる．

しかし，Dualflow アーキテクチャには，命令の配置に関しての制約が強いという問題がある．Dualflow アーキテクチャでは，コンシューマを指定する場合に n 命令後というように指定する．そのため，パスの分岐が発生した際にその分岐命令を跨いでデータを受け渡す場合には，分岐後のそれぞれのパスにおいてコンシューマの位置を揃えなくてはならない．正しい位置にコンシューマの命令を配置することができない場合には，何もしない NOP 命令など本来実行には不必要な命令を挟むことでコンシューマの位置を揃える必要がある．このような無用な命令によって，Dualflow アーキテクチャでは通常の制御駆動型の命令セット・アーキテクチャと比べて，同じ処理を行うための命令数が増加するという問題がある．

2.2.2 逆 Dualflow アーキテクチャ

一林らは，上で述べた Dualflow アーキテクチャを基にレジスタ・リネーミングそのものを省略することを目的とした，逆 Dualflow アーキテクチャを提案している [31]．逆 Dualflow アーキテクチャは，命令の出現した順番と同じ順番で，命令の実行結果をサイクリックな物理レジスタ・ファイルに格納する．また，物理レジスタ・ファイルとは別に，論理レジスタの値を保持するための論理レジスタ・ファイルが用意されており，命令の実行結果がインオーダに格納される．初回実行時の命令変換では，ソース・オペランドを物理レジスタ・ファイル上の変位で指定するようになっている．実際に命令に記述される値としてソース・オペランドには，何命令前の実行結果を参照するかが与えられることになる．

逆 Dualflow アーキテクチャの効果として，

- レジスタ・リネーミングに必要なハードウェア電力の削減
- レジスタ・リネーミングの省略による，分岐予測ミス・ペナルティの減少
- フロントエンドにおいて命令間依存解消による，フロントエンド幅増加の可能性

がある．

しかしながら，逆 Dualflow アーキテクチャでは消費電力を低減できる代わりに，変換した命令を経路によって区別して格納する必要が生じてしまい，デコードされたマイクロ命令を保持するトレースキャッシュのキャッシュミス率が増加するという問題がある．またこれにより，IPC は低下してしまう．

STRAIGHT アーキテクチャは，この逆 Dualflow アーキテクチャのレジスタ・リネーミング省略の手法とソース・オペランドを物理レジスタ・ファイル上の変位で指定する方式を命令セットに導入している．また，コードを静的に生成することで，制御コストの更なる削減と，従来にないレジスタ容量の活用を目指している．

第3章　STRAIGHTアーキテクチャ

3.1　STRAIGHTアーキテクチャの構想

前章で紹介したように，プロセッサアーキテクチャは様々な技術によりIPCを向上させ電力効率を向上させている．しかしながら，ダークシリコン問題を解消しながらプロセッサ性能の成長を継続させるためには，IPCまたは電力効率どちらか一方の向上に焦点を当てたアーキテクチャではなく，IPCとIPC/電力比の双方を従来アーキテクチャよりも向上させるようなアーキテクチャが必要である．現存のアーキテクチャの組み合わせによるメニーコアプロセッサでは性能向上の限界が指摘されている．その一方で，技術の発展により同パッケージ内に搭載可能なトランジスタの数は増加を続けている．半導体を3次元的に積層する技術[32]も進展し，増加するトランジスタ資源を活用してこれからのプロセッサ成長を持続可能にするアーキテクチャの有効性が高まっている．

IPC向上のために，発行幅や実行幅を増加させる技術は提案されているが，広い発行幅を有効活用できるようなワークロードは少ない．一方で，並列性の高いワークロードは一般に並列化され，マルチコア・プロセッサでより効率的に並列実行することができる．しかし，性能のオーバヘッドとなるシーケンシャルな部分の実行時間各コアのILP性能によってしか縮めることができない．シングルスレッド実行方式に求められているのは，並列度の低いコードをより低レイテンシで実行する性能である．命令ウィンドウサイズを増加させることで，バックエンドパイプラインの稼働率を増加させることを考えたとしても，物理レジスタ数の増加につながり，レジスタ・リネーミングのためのRMT(Register Map Table)の容量が増加してしまう．それに加え，大きな命令ウィンドウを最大限に活用するためにはフロントエンド幅を増加させる必要があり，リネーミングの処理をより複雑にしてしまう．

次にIPC/電力比に関して，消費電力の制約がある中で性能を高めるためには，同じ命令をより少ない消費電力で処理できるアーキテクチャが必要である．それぞれの命令を処理するパイプラインにおいて，その命令を制御する部分の消費電力を極力削減し，核となる命令実行部分に電力を費やすことが望まれる．命令の順番を変えて実行することにより、複数命令の同時実行の可能性を広げる最適化手法の1つにアウト・オブ・オーダ実行がある．しかし，アウト・オブ・オーダ実行は制御が複雑化し，その制御の消費電力においては特にレジスタリネーミングが主要な負荷として知られている．

以上のようにIPCの向上とIPC/電力比の向上の両方においてレジスタの管理方法が重要な要素となっている．そこで我々は，トランジスタの増加をレジスタ容量の増加に用いて制御を軽量化するSTRAIGHTアーキテクチャを提案している．以下がSTRAIGHTアー

キテクチャの構想である．

1. 充分に広い論理レジスタ空間をライト・ワンス・マナーで使用するようなプログラムコードを仮定すると，そのコードの実行には偽依存が発生しない．そのため，レジスタリネーミングの処理を行う必要がなく，コードに書かれた通りのレジスタオペランド値で実行できる．

2. 物理レジスタ数が充分にあれば，一つ一つのレジスタを解放するタイミングを管理する必要がなくなり，物理デスティネーションレジスタ番号を命令をフェッチした順番に割り振ることが可能になる．レジスタリネーミングがなく，物理デスティネーションレジスタ番号がフェッチ順に定まるようなフロントエンド処理では，命令間の依存ループが存在せず，フェッチブロックをそのまま並列にデコード・ディスパッチすることができる．

3. 管理が単純な大容量レジスタと拡張の容易なフロントエンド幅によって，命令ウィンドウサイズを拡張することが容易であり，バックエンドパイプラインの稼働率を増加させる．

レジスタを大容量化する背景には，トランジスタの数が増加していることの他に，大容量化したとしても分散構成であれば，アクセス電力は増加しないことが挙げられる．STRAIGHT アーキテクチャでは近年急速に進歩を遂げている分散キー・バリュー・ストア技術の適用を構想している．分散キー・バリュー・ストア技術の適用については 3.5 節で詳しい動作例を述べる．さらに，PCPG[10] を始めとするリーク電力や電源遮断に関する技術が進んでおり，容量増加によって生じる電力は対策することができる．また，レジスタリネーミングの処理は，稼働率の高い多ポート RAM(Random Access Memory) アクセスであり，ひとつの命令実行ごとに多くの電力を消費することになる．加えて，レジスタリネーミングは，主要なクリティカルループの一つとして知られている [33]．このように，STRAIGHT アーキテクチャでは，論理・物理レジスタともに数を増やすことにより，レジスタリネーミングを始めとする制御の消費電力を削減し，発行・実行幅を増やさずに IPC を向上させる設計を可能とする．

3.2 命令セットアーキテクチャ

STRAIGHT アーキテクチャのレジスタは再書き込みされないライト・ワンス・マナーに従い動作する．STRAIGHT 命令は一般的な RISC(Reduced Instruction Set Computer) 命令と同様の簡単なオペレーションを行う．また，演算や転送，制御，システムといった典型的な命令群を備えている．STRAIGHT の命令セットは 5 章で紹介する．STRAIGHT アーキテクチャにおいて処理は命令単位である．命令は 32 ビットの固定長であり，最上位ビットである MSB(Most Significant Bit) 側からオペレーションコード，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 の指定フィールドとなる．命令形式はその命令がいくつのソースレジスタを使用するかで変わる．STRAIGHT の命令形式を図 3.2.1 に示す．

ソースオペランドは dualflow 形式のように命令位置の距離で指定される．例えば，ソースオペランドレジスタ L の値が 5 の場合，プログラム中で 5 つ前の命令の実行結果（デスティネーションレジスタの値）がオペランド L の値となる．ソースオペランド R は即値をとる場合もあり，オペレーションコードからその命令の形式を判断することで区別する．デスティネーションレジスタ番号は，命令のフェッチ順に定まるため，命令中での指定を省くことが可能になる．この STRAIGHT 命令による簡単なアセンブリコードを図 3.2.2 に示す.

STRAIGHT 命令において，ソースオペランドレジスタのフィールド長は 10 ビットであり，これにより STRAIGHT 命令は 2^{10} 個前までの命令の実行結果（デスティネーションレジスタの値）を参照できることとなる．これが STRAIGHT アーキテクチャの汎用レジスタに相当する．この命令形式に従うことにより，ライト・ワンス性を保つことが可能になり，2^{10} 個後の命令実行終了後にはそのレジスタは参照されない，というレジスタの寿命が保証される．コミットとデコード間の最大命令数を m とすると，コミット中の命令のレジスタ番号の後に最大で m 個の番号が割り当てられる．このとき，m は命令ウィンドウサイズであり，スケジューラエントリ数にほぼ等しい．RP は $2^{10}+m$ までの数値をとり，ターンアラウンドして 0 に戻る．レジスタファイルは，$2^{10}+m$ エントリのテーブルとなりスケジューラマトリクスは $m \times m$ のサイズで構成される [1].

汎用レジスタの他にも STRAIGHT アーキテクチャは特殊レジスタとしてレジスタポインタを備えている．このレジスタポインタについては STRAIGHT においてのパイプラインで説明する．また，上書き可能な特殊レジスタとして，スタックポインタとフレームポインタを備えており，メモリを介することで，ライト・ワンス性やレジスタの寿命制約のない柔軟な参照記述が可能である．また，フェッチは通常の RISC アーキテクチャと同様に，上書き可能な PC レジスタによって行われる．

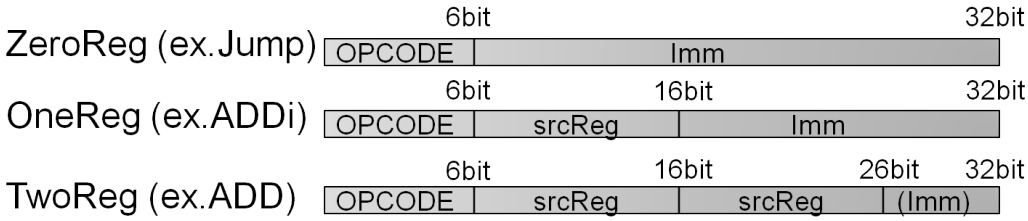

図 3.2.1: STRAIGHT アーキテクチャの命令形式

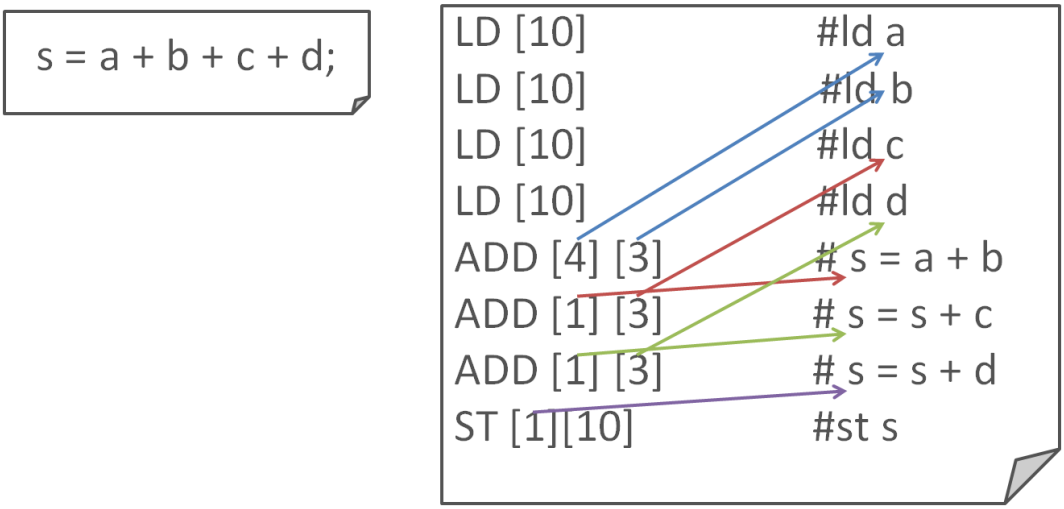

図 3.2.2: STRAIGHT コードサンプル

3.3 STRAIGHTコードの生成

オペランドの指定方式として，逆 Dualflow 方式のようにコンシューマからプロデューサへの命令距離を指定する方式では，制御の合流時において辿ってきたパスによって指定する命令への距離に変化が生じてしまう問題がある．図 3.3.1 (a) のような分岐では， パス A とパス B に分岐し，それぞれのパス中で変数 x が更新される．その後合流し，それぞれで更新された変数 x を参照するような命令がある場合，パス A での変数 x の更新位置とパス B での変数 x の更新位置が，参照命令から異なった距離に配置されているとオペランド指示値が定まらなくなってしまう．

この問題を解決するためには，分岐後のそれぞれのパスの長さおよび変数更新の順番を調整するコンパイラアルゴリズムが必要である．まず，分岐前に定義された値を合流後に参照するような場合について考える．このような場合においては，分岐した時点から合流するまでの間に取りうる全てのパスについて，その長さが全て等しければ距離指定は整合することになる．そこで，パスの長さが静的に定まるような場合においては，全てのパスの長さが等しくなるように，何もしない NOP(No Operation) 命令を挿入すれば良い (図 3.3.1 (b))．しかし，関数の呼び出しが入れ子になっているような場合や，ループの回数が動的に決まるような場合については，パスの長さが静的に解析できないので，分岐前の値を参照することができない．このような場合については，合流後に引き続き参照されるデータを SP(Stack Pointer) を用いてメモリに退避する必要がある．このようなメモリアクセスを軽減させることが STRAIGHT コンパイラ最適化の課題である．

次に，図 3.3.1 (c) のように，分岐後のそれぞれのパス中で更新される値を合流後に参照するような場合について考える．このような場合は図に示すように，それぞれのパスにおいて合流前の等距離の位置にレジスタ値を移動する RMOV (Register Move) 命令をはさむことで位置調整を行えば，どのパスを通ったとしても正しい値を参照することが可能になる．また，関数の呼び出し前後の参照についても同様に，引数や返り値の距離をあらかじめ定めておくことにより，制御合流の問題を解決することができる．

図 3.3.2 にフィボナッチ数列を計算するコードを示す．右側の STRAIGHT コードでは一つ前のループの結果 ($x[i-1]$) を RMOV 命令を用いることで移動し，メモリに読みに行かなくても良いコードになっている．加えて，このような 1 重ループのコードの場合，STRAIGHT コードではループ毎に使われる変数 (図 3.3.2 の例では i や $loop$ といった変数) はロードする必要がなく RMOV を用いることで繰り返し使いまわすことができる．また，STRAIGHT コードにおいてこのようなコードではループの 1 回目をループから外に出し，コードに明示的に書くことで，２つ前のループの値 ($x[i-2]$) も同様に RMOV を使いメモリ読み出しを削減するといった最適化も行うことが可能である．

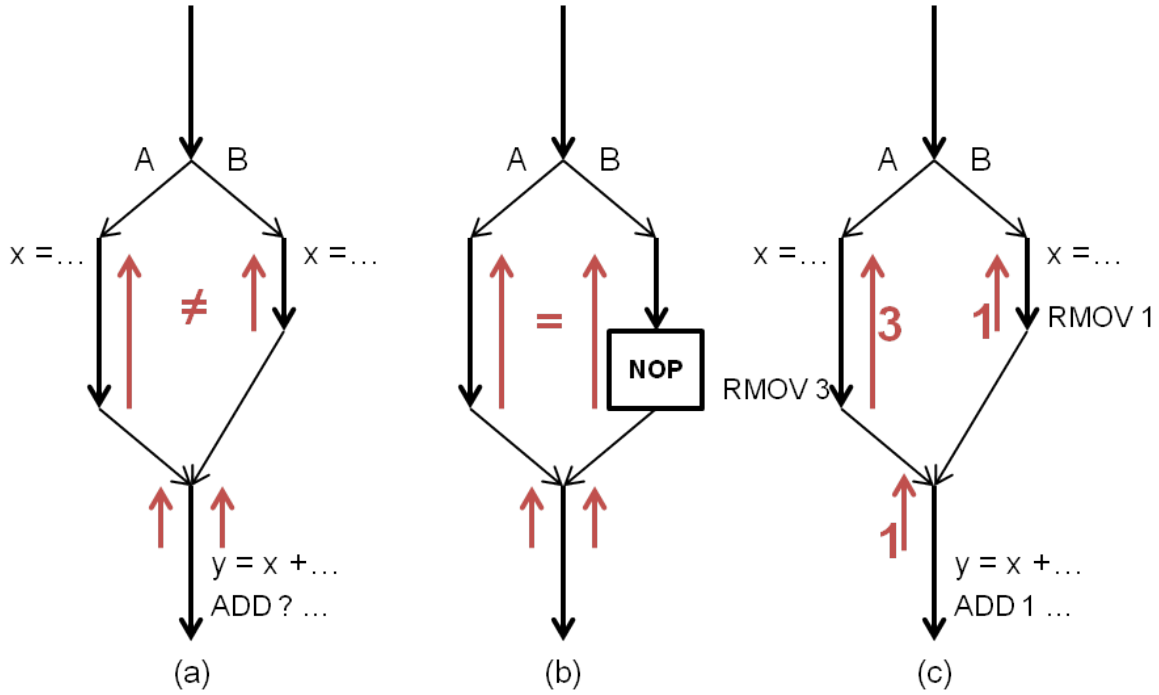

図 3.3.1: STRAIGHT コードにおける分岐・合流時の制御

C言語コード

```
x[0] = 1;
x[1] = 1;
for(i=2; i<loop; i++)
   x[i] = x[i-2] + x[i-1];
```

STRAIGHTコード

```
1. SPLD   -4        // x[0]のアドレス
2. ADDi   0  1      // 1を生成
3. ST     1  2      // x[0] = 1
4. ADDi   3  32     // x[1]のアドレス
5. ST     3  1      // x[1] = 1
6. SPLD   -8        // loop
7. ADDi   0  2      // i = 2
8. NOP              // 距離調整
9. SLT    2  3      // i < loop
10.BEZ    1  8      // end of I Loop
11.LD     1  -64    // x[i-2]をロード
12.RMOV   10        // x[i-1]を前ループから利用
13.ADD    2  1      // x[i-2] + x[i-1] (最終目的値)
14.ADDi   10  32    // x[i]のアドレス
15.ST     1  4      // x[i]にストア
16.RMOV   10        // loop
17.ADDi   10  1     // i++
18.J      -9        // back to I Loop
19.NOP              // EOF
```

図 3.3.2: フィボナッチ数列を計算するコード

3.4 STRAIGHT アセンブラ

4 章で述べる STRAIGHT シミュレータの入力として STRAIGHT バイナリが必要となる．そこで前章で述べたような特徴を持つSTRAIGHT アセンブリコードを入力としSTRAIGHT バイナリを出力する STRAIGHT アセンブラを実装した．入力となる STRAIGHT アセンブリコードはテキストファイルの形式を想定している．このテキストファイルは，1 行 1 命令として記述され，前からその命令の種類，ソースオペランドレジスタ L の値，ソースオペランドレジスタ R の値の順で並んでいる．

STRAIGHT アセンブラは図 3.4.1 のように入力されたアセンブリコードをバイナリコードに変換する．まず，読み込んだ 1 命令の先頭に記述されている命令の種類を解読し，その命令の Opcode 値を決める．命令の種類を解読するとその命令形式を判別することができるので，以降記述されている値がレジスタ値なのか即値なのかを特定することができる．

図 3.4.1 の例では，ADD 命令の Opcode が 8 なので Opcode 分の最初の 6 ビットに 8 を入れる．ADD 命令は 2 つのレジスタを取るので，以降の数字は 2 つともレジスタ値とし，それぞれ 10 ビットに変換する．バイナリコードは 8 ビット毎に記述するので，変換した値を連結し先頭から 8 ビット毎に分ける必要がある．STRAIGHT において 1 命令は 32 ビットなので 4 つの 8 ビットの値を作る．先頭から区切るので足りない分は 0 で埋める必要がある．出力となる STRAIGHT バイナリはこれらのバイナリコードをファイルの先頭から記述したものになる．そのため，STRAIGHT シミュレータはこの専用バイナリを入力として読み込み，先頭から命令として格納すればよいことになる．グローバルデータの初期値やバイナリにデータを載せる機能を追加することが今後の課題である．

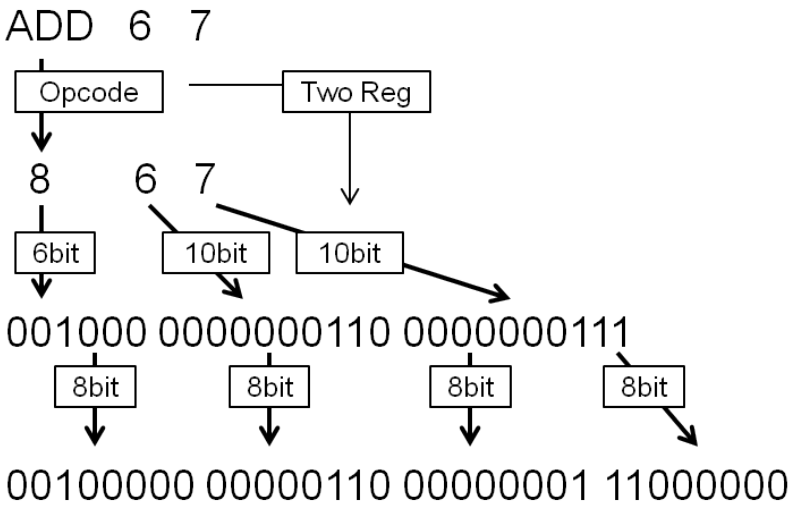

図 3.4.1: STRAIGHT アセンブラ実行例

3.5 動作の概要

STRAIGHT の命令処理はフェッチ，デコード，ディスパッチ，イシュー，レジスタリード，エクセキューション，メモリ，レジスタライトの各ステージからなる．このステージ構成は，レジスタリネーミングがないことを除けば，スーパースカラプロセッサに類似している．STRAIGHT プロセッサのブロック図を図 3.5.5 に示す．メニーコアプロセッサを構成する一つ一つの SAC (STRAIGHT Architecture Core) が並列実行パイプラインと分散レジスタファイルを備えている．

STRAIGHT アーキテクチャにおけるフロントエンドパイプラインでは，まず図 3.5.1 のように複数分岐をまたぐ命令を同時にフェッチする．ここで，RP(Register Pointer) と呼ぶ，命令 1 つごとにシーケンシャルに値の増加するアーキテクチャレジスタを導入する．それぞれの命令はこの RP の値に従い，RP からオペランドフィールドで指定される距離を引くことでソースレジスタ番号を決定する．デスティネーションレジスタ番号はその命令での RP の値と同じである (図 3.5.2)．レジスタ番号は該当命令とその時点での RP の値のみによって決定するため命令同士に依存がなく，並列化が容易な処理となっている．

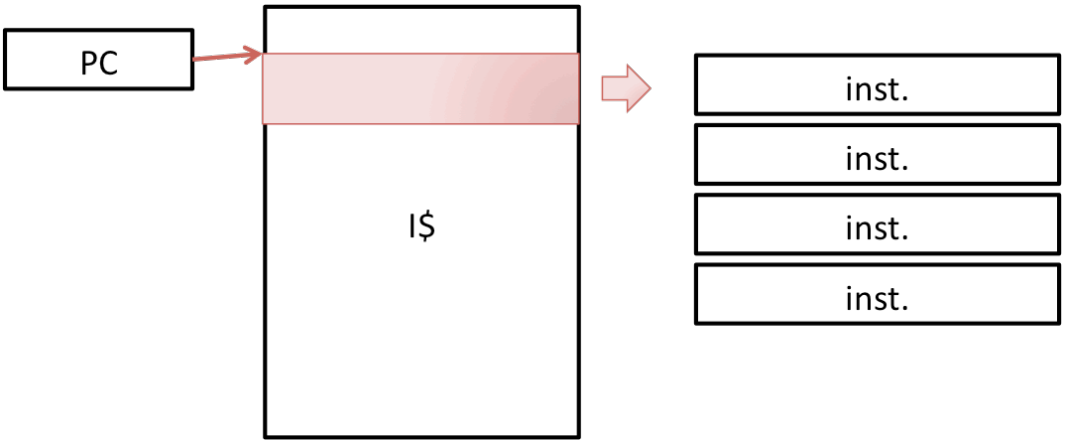

図 3.5.1: フェッチステージ

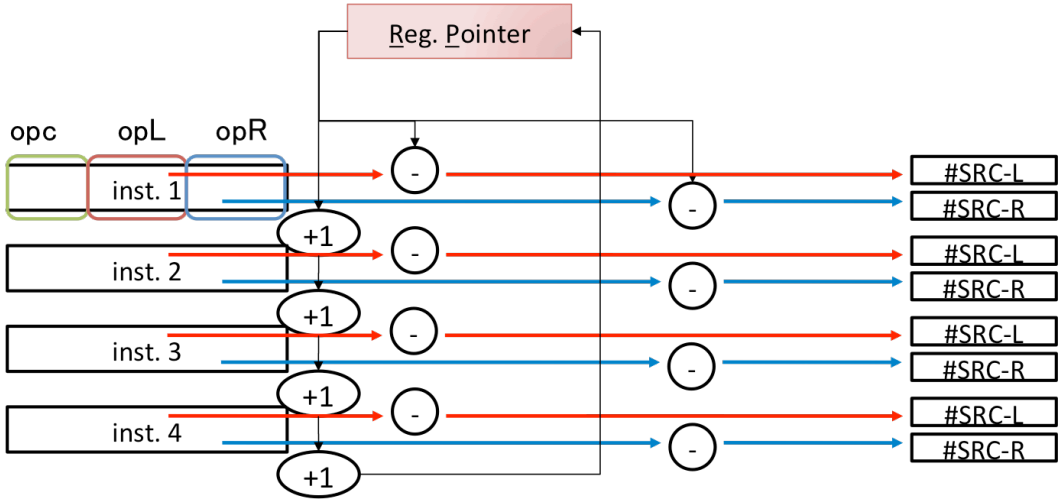

図 3.5.2: デコードとレジスタ番号の決定

命令はデコード後そのままディスパッチされる．レジスタの大容量化による命令ウィンドウ幅の恩恵を得るためにはスケジューラサイズを大きくする必要がある．スケジューラでは，ディスパッチ後からコミットまでのインフライトな命令を対象として管理する (図 3.5.3)．それぞれの命令情報を管理するペイロード RAM は，サブアレイに分割することでスケジューラのエントリ数に対してスケーラブルな実装が可能になる．ウェイクアップやセレクトに関しては，発行幅は従来のアーキテクチャと同様の規模であるが，エントリ数が増加するため，従来よりも大規模なものになる．そのためこの部分については，マトリックス方式 [34][35] を用いることにより軽量化する．

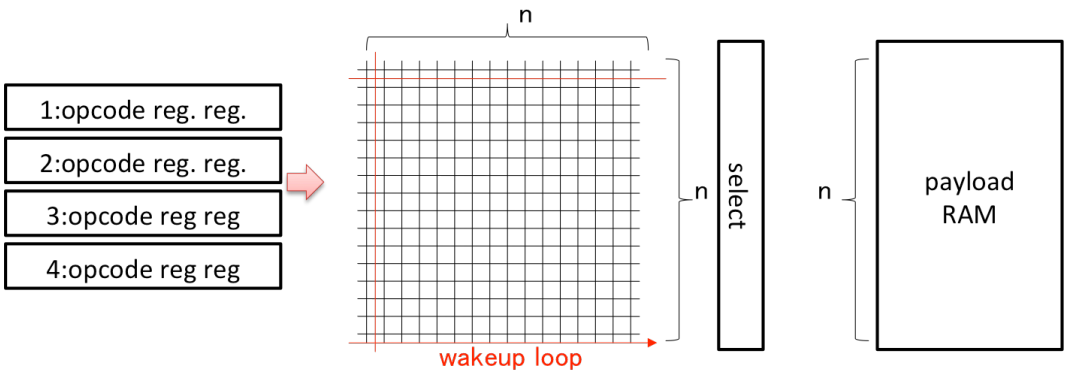

図 3.5.3: スケジュールステージ

命令が発行された後は，ペイロード RAM の情報に従ってソースレジスタが読み出される．STRAIGHT アーキテクチャのレジスタファイルは，レジスタ番号とレジスタ値を対とした大きなキー・バリュー・ストアを分散した実装となる．複数のテーブルで構成され，レジスタ番号にハッシュ演算を行うことで該当するテーブルの ID を得て問い合わせ

る．例えば，簡単な実装として，存在しうる全てのレジスタ番号について，1 対 1 に対応する物理レジスタを備えている場合，レジスタ番号がそのままテーブル ID およびエントリ ID となる．レジスタファイル全体は大容量なものとなるため，ワイヤ長が長くなり読み出しレイテンシが増加してしまう可能性があるが，このレイテンシについては性能への影響が小さいことが知られている [33]．また，アクティブ電力は該当したテーブルのみで発生するため，レジスタ読み出し 1 回あたりのアクティブ電力の増加は大きくない．

STRAIGHT アーキテクチャにおけるバックエンドパイプラインでは図 3.5.4 のように，従来の RISC 同様に機能ユニットを備えていて，実行は 1 命令単位で行われる．ここでは，発行幅や実行幅の増加は行わず，バックエンドのクラスタ化なども行わない．従来と同様のクリティカルパス長を持つ集中型データパスと，バイパスネットワークを備えている．実行後のコミットはインオーダに行われるが，実行と例外の確認を行った後に，次に実行すべき命令のアドレスを保持する PC(Program Counter)，グローバル宣言された変数をメモリから参照する際に使用する GP(Global Pointer)，メモリ上のスタックで最も最近参照された位置のアドレスを保持する SP(Stack Pointer)，関数呼び出しの際に SP の値を保持する FP(Frame Pointer)，RP のコミットステートを更新するのみで良い．RP の更新を行うことで，参照されないレジスタが生じ，暗黙的にフリーレジスタが更新されるため，イシューした後は管理しなくてよい軽量なコミットとなる．

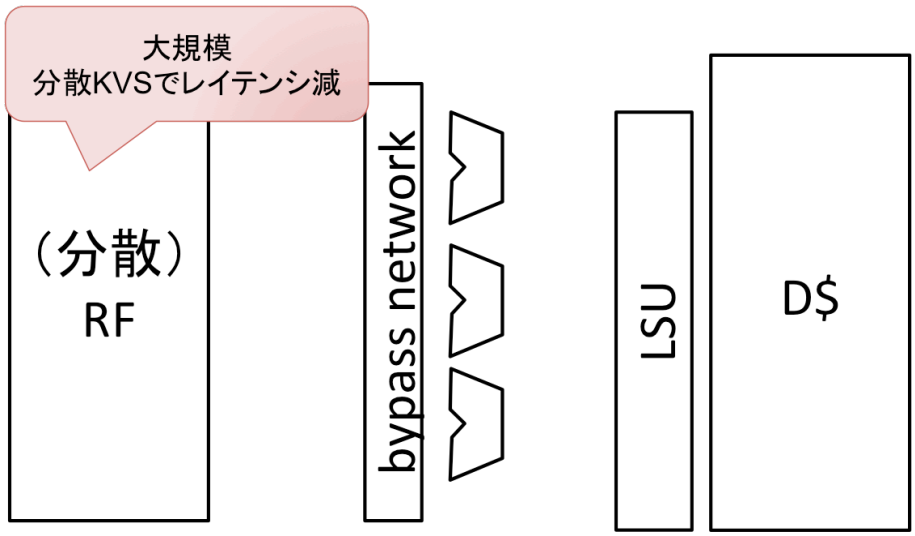

図 3.5.4: STRAIGHT アーキテクチャのバックエンド

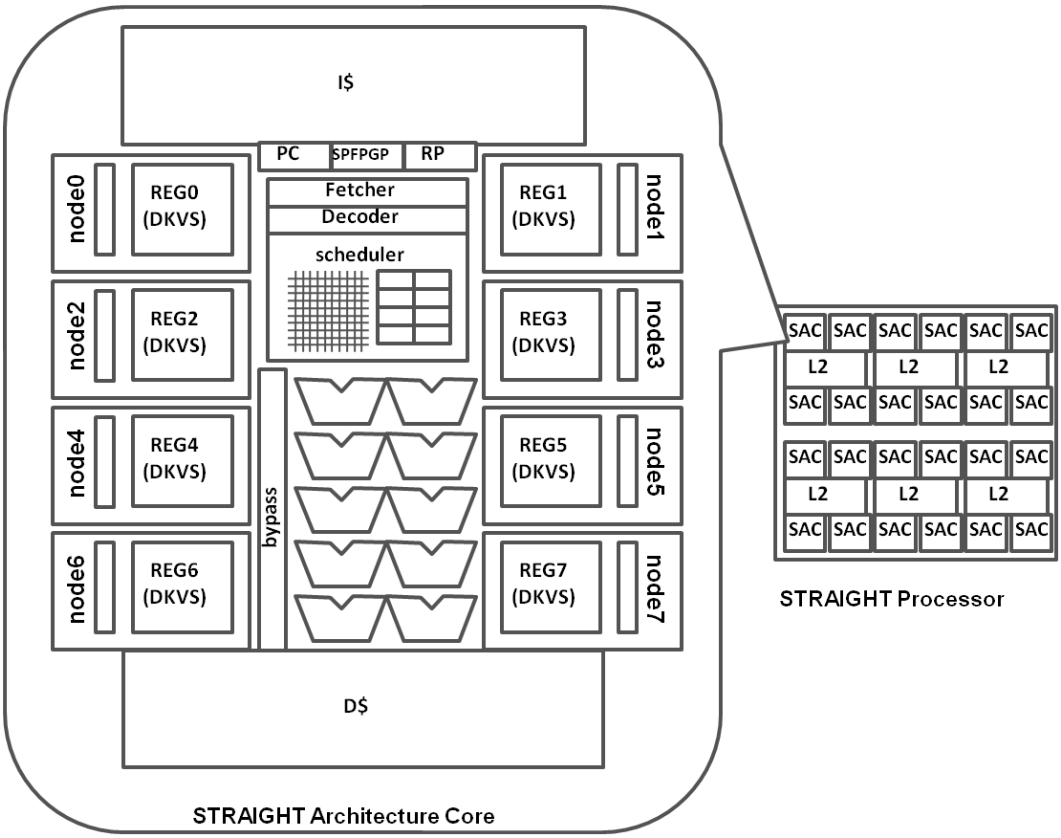

図 3.5.5: STRAIGHT アーキテクチャブロック図

第4章　STRAIGHTシミュレータの設計

4.1　初期評価 [1]

STRAIGHT アーキテクチャの初期評価では，既存のプロセッサシミュレータ「鬼斬弐」と STRAIGHT アーキテクチャに見立てたアーキテクチャパラメタを用いた [1]．ベンチマークには SPECCPU2006 の全プログラムを用いた．さらに，電力評価には電力シミュレータ McPAT[36] を用い，22nm プロセスを仮定した上で，シミュレーションの出力統計から入力を作成した．プロセッサシミュレータを用いた評価では，STRAIGHT アーキテクチャパラメタのフロントエンド幅をベースラインから 1 倍，2 倍，4 倍，命令ウィンドウ幅をベースラインから 1 倍，4 倍，8 倍に変化させた．フロントエンド，命令ウィンドウ幅の増加とともに IPC の向上が見られ，メモリレイテンシが 200 サイクルの場合ベースラインから 30%の性能向上が見込まれ，今後の積層技術などによってこのポテンシャルはさらに約 10%向上することがわかった．電力評価では，STRAIGHT アーキテクチャではリネームロジックの電力が必要ない一方で，レジスタのリーク電力やスケジューラ電力の増加により，トータルではベースラインプロセッサの実行コア部分に対して 18%の電力増となった．しかし，同じ処理への実行サイクル数を反映させた，消費電力の相対比較では，STRAIGHT アーキテクチャは同じ処理に必要なエネルギー量を約 12%削減していることがわかった．初期評価より，STRAIGHT アーキテクチャは従来のアーキテクチャと比べ，同じワークロードに対するエネルギー消費を 12%削減しながら，同時に約 30%の IPC 向上が得られ，性能/パワー比を改善する新しい実行方式として有効であることを示すことができた．

しかしながらこの初期評価では，プロセッサシミュレータ「鬼斬弐」に初期実装されているアーキテクチャである Alpha AXP アーキテクチャを実行モデルとしている [37][38]．そのため，この評価では特殊レジスタやレジスタリネーミングの排除をはじめとする前章で紹介した STRAIGHT アーキテクチャの特徴を再現できていない．より正確に STRAIGHT アーキテクチャを評価するために，アーキテクチャ仕様に忠実なシミュレータを設計・実装する．この章では，ベースとして用いたプロセッサシミュレータ鬼斬の紹介と，鬼斬への STRAIGHT の実装について述べる．

4.2　プロセッサシミュレータ鬼斬

ベースとなるプロセッサシミュレータには鬼斬弐 [9] を採用した．鬼斬弐はアーキテクチャの評価に使用されるサイクルアキュレットなプロセッサシミュレータである．また，

スーパースカラ・プロセッサの基本的な性質を抜き出して設計されており，様々なプロセッサアーキテクチャに共通な基本機能によって核となる実行モデルが記述されている．基本機能を各アーキテクチャの仕様に派生させることで，ターゲットアーキテクチャの挙動を忠実に再現する．マルチコアやマルチスレッドにも対応しており，新アーキテクチャを実装すればそれをマルチコアで動かした際の評価も可能である．

図 4.2.1 にプロセッサシミュレータ鬼斬の構成図を示す．プロセッサシミュレータ鬼斬弐は，ベンチマークプログラムをコンパイルし，Linux ELF バイナリを入力とすることで，シミュレーションを行うことができる．出力は，ベンチマークとしたプログラムの実行結果，IPC やキャッシュヒット率などのシミュレーション結果であり，XML 形式のファイルで出力される．図 4.2.1 の通り，鬼斬シミュレータは大きく分けてエミュレータとシミュレータの２つで構成されている．エミュレータ側には，アーキテクチャエミュレータ，ELF ローダ，バーチャルメモリなどが含まれ，シミュレータ側には，パイプライン処理に必要な各モジュールに加え，分岐予測器，レジスタファイル，キャッシュ，メモリ，インオーダリストなどが含まれる．

プロセッサシミュレータ鬼斬のパイプラインモデルでは，まずフェッチとリネームに関してはインオーダに行い，制御と実行に関してアウトオブオーダ実行される．その後エクセキューションステージでは，リネーム時に解析した依存関係を満たすようにスケジューリングされた命令を，アウトオブオーダに実行する．スケジューラはプロセッサ内に１つまたは複数存在している．最後に，コミットとリタイヤはフロントエンド同様インオーダに実行される．STRAIGHT シミュレータのより詳しいパイプライン動作については 6 章で紹介する．

プロセッサシミュレータ鬼斬は，シミュレーション時に入力としてスキップ数が指定された場合，指定された回数のエミュレーション後，指定された命令数のパイプラインシミュレーションに移行する．エミュレーションはシミュレータが管理する物理レジスタ上の値を使用して実行される．このエミュレーションによって，ハードウェアモデルにミスがあった場合正しくない実行結果が得られ，性能バグの検出が容易になるということと，誤った値を使用しての実行継続が可能なことから，データ系投機の実行が可能である．鬼斬に初期実装されているアーキテクチャエミュレータには，Alpha と PowerPC 64bit がある．鬼斬シミュレータにおけるエミュレータの主な役割は，対象アーキテクチャにそった命令の実行であり，エミュレータは入力としてデコード済みの命令情報が渡される．命令情報には，オペコード，ソースレジスタ，即値，デスティネーションレジスタなどが含まれる．命令実行はオペコード値で場合分けされ，ソースレジスタ値と即値を用いたそれぞれの処理が行われる．実行結果はデスティネーションレジスタに書きこまれ，命令情報として返す．エミュレータに備わる各実行ルーチンは，シミュレータのパイプライン処理のタイミングで呼び出される．また，プロセッサシミュレータ鬼斬と同時に公開されているツールとしてクロスコンパイラがあり，これを使用することで，任意の C 言語コードをベンチマークとして鬼斬シミュレータを用いて評価することができる．

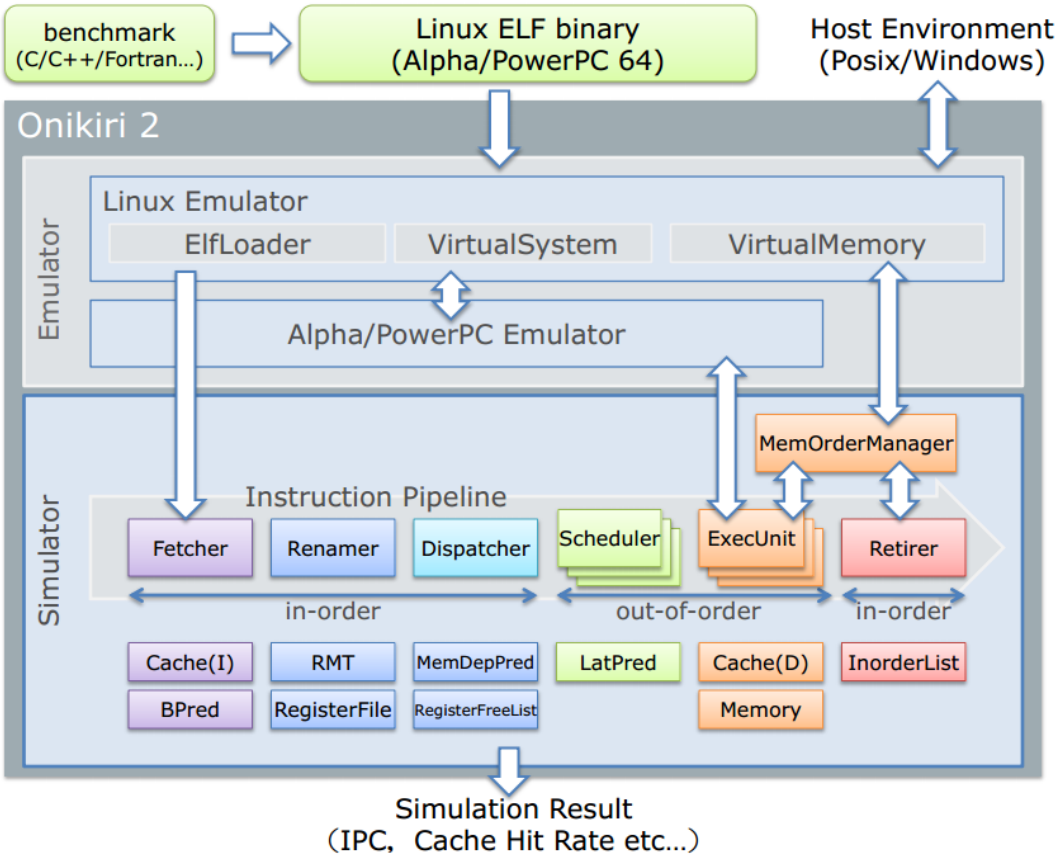

図 4.2.1: プロセッサシミュレータ鬼斬弐の構成図 [2]

4.3 鬼斬をベースとしたSTRAIGHTシミュレータの実装

前節で紹介したプロセッサシミュレータ鬼斬をベースにSTRAIGHTシミュレータを構築した．STRAIGHTエミュレータの構成要素を図4.3.1に示す．図4.3.1のように，STRAIGHTエミュレータは，エミュレータ，ローダ，デコーダ，オプステート，オプインフォ，ISAインフォ，アーキテクチャステートで構成する．鬼斬への実装としてまず，専用のSTRAIGHTバイナリを読み込むためのSTRAIGHTローダを新たに実装した．このSTRAIGHTローダは，図4.2.1のELFローダの役割を果たす．また，鬼斬シミュレータに初期実装されているアーキテクチャエミュレータであるAlpha，PowerPC 64bitに加えて，新たなアーキテクチャエミュレータとしてSTRAIGHTエミュレータを実装した．STRAIGHTエミュレータの主な機能は従来のアーキテクチャエミュレータ同様，前節で説明した命令の実行である．STRAIGHTデコーダは命令のデコードを行う．STRAIGHTオプステートは，ソース・デスティネーションレジスタの取得，ソースオペランドとRPの差分計算によるソースレジスタ番号の決定を行い，動的な命令情報を管理する．STRAIGHTオプインフォは，オペコード，ソース・デスティネーションレジスタ，ソース・デスティネーションレジスタ数，即値などそれぞれの命令情報を保持する．STRAIGHTISAインフォは，命令ビットサイズ，ソースレジスタの最大個数，デスティネーションレジスタの最大個数，即値の最大個数，アーキテクチャが保有するレジスタの個数を保持する．STRAIGHTアーキテクチャステートはSTRAIGHTの特殊レジスタであるRP，SP，FPの保持・管理を行う．エミュレータの具体的な動作は次章で述べる．

ここまでは図4.2.1上側のエミュレータ部分の実装について述べた．ここからは図4.2.1下側のシミュレータ部分の実装について述べる．シミュレータ側では主に図4.2.1右下に示されるインオーダリストについて，STRAIGHTの特徴を取り入れる実装を行った．インオーダリストでは，命令の生成・管理・破棄を行っている．STRAIGHTシミュレータでは図4.3.2のように，この命令生成時にエミュレータのアーキテクチャステートからRPの値を取得する．このRPの値をIDとし命令が生成される．さらに，その命令の情報として，取得したRPの値をデスティネーションレジスタ番号としてオプインフォに格納する．また，RPとソースオペランドの差分を取ることでソースレジスタ番号を決定し，オプインフォに格納するよう実装した．これはシミュレーション時の動作であり，エミュレーション時は差分計算をオプステートが担当する．１命令の生成が終わるごとにRPは１インクリメントするようにした．鬼斬でのシミュレータパイプライン上にあるリネーミングは，STRAIGHTシミュレータでは排除した．加えて，分岐予測ミスをした場合には，インクリメントされているRPを戻すという処理を行っている．それと同時に，先行して発行されている命令の使用するレジスタ間にある依存関係をすべてリセットするよう実装した．この操作を行う理由は，依存関係がそのまま残っている場合，巻き戻されたRPが正しいパスを通ってくると，そのRPに対応する命令のデスティネーションレジスタには既に間違った依存関係が生成されていることになってしまうからである．パイプラインシミュレータの詳しい動作は第６章で述べる．

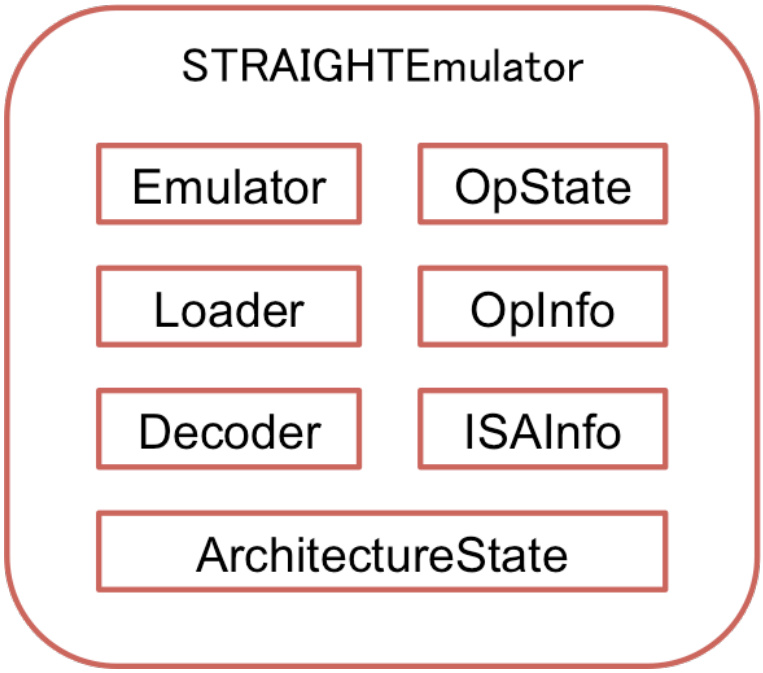

図 4.3.1: STRAIGHT エミュレータの構成要素

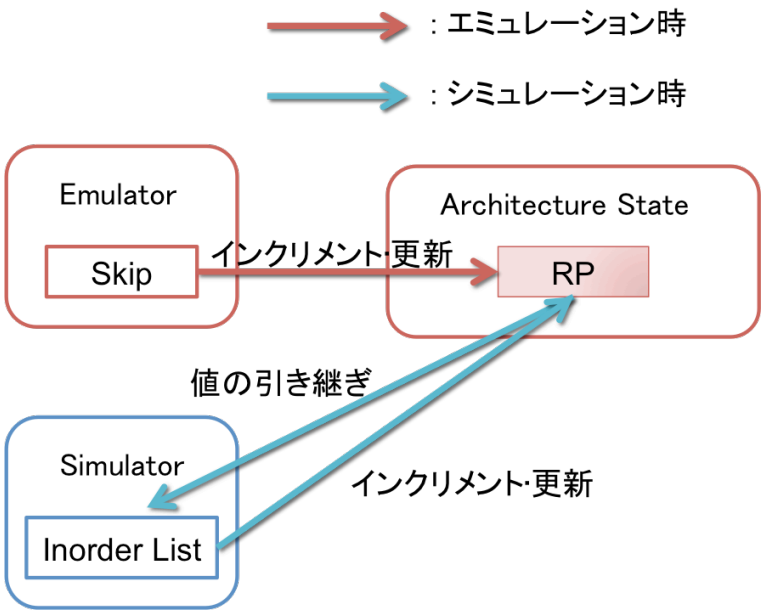

図 4.3.2: エミュレーション・シミュレーションにおける RP の引き継ぎと更新

第5章　STRAIGHTエミュレータ

5.1　命令セットver1.0

STRAIGHT命令セットは一般的なRISC命令と同様，演算，転送，制御，システムといった典型的な命令群を備える．STRAIGHT命令セットver1.0を表5.1に示す．STRAIGHT命令セットver1.0は，何もしない命令(NOP)，ジャンプ(J，JR，JAL)，ブランチ(BEZ，BNZ)，FPを用いたブランチ(FPBEZ，FPBNZ)，演算(ADD，SUB，MUL)，即値を用いた演算(ADDi，SUBi，MULi)，整数・浮動小数点数変換(ITOF，FTOI)，浮動小数点演算(FADD，FADDi，FSUB，FSUBi，FMUL，FMULi，FDIV,，FDIVi)，シフト演算(SHL，SHLi，SHR，SHRi)，論理演算(AND，ANDi，OR，ORi，XOR，XORi)，SP・FPの移動(SPADD，SPADDi，FPADD，FPADDi)，ロード・ストア(LD，SPLD，FPLD，ST，SPST，FPST)，RPの移動(RPINC)，比較(SLT)，レジスタ移動(RMOV)からなる．STRAIGHTアセンブラやSTRAIGHTデコーダはこれらの命令に対応するように実装した．また，STRAIGHTエミュレータの実行部分では，これらの命令一つ一つの処理を実装した．

表5.1: STRAIGHT命令セットver1.0

オペコード	種類	命令形式	オペコード	種類	命令形式	オペコード	種類	命令形式
0	NOP		16	FSUB	TwoReg	32	SPADD	OneReg
1	J	ZeroReg	17	FSUBi	OneReg	33	SPADDi	ZeroReg
2	JR	OneReg	18	FMUL	TwoReg	34	FPADD	OneReg
3	JAL	ZeroReg	19	FMULi	OneReg	35	FPADDi	ZeroReg
4	BEZ	OneReg	20	FDIV	TwoReg	36	LD	OneReg
5	BNZ	OneReg	21	FDIVi	OneReg	37	SPLD	ZeroReg
6	FPBEZ	OneReg	22	SHL	TwoReg	38	FPLD	ZeroReg
7	FPBNZ	OneReg	23	SHLi	OneReg	39	ST	TwoReg
8	ADD	TwoReg	24	SHR	TwoReg	40	SPST	OneReg
9	ADDi	OneReg	25	SHRi	OneReg	41	FPST	OneReg
10	SUB	TwoReg	26	AND	TwoReg	42	RPINC	ZeroReg
11	SUBi	OneReg	27	ANDi	OneReg	43	SLT	TwoReg
12	FTOI	OneReg	28	OR	TwoReg	44	RMOV	OneReg
13	ITOF	OneReg	29	ORi	OneReg	45	MUL	TwoReg
14	FADD	TwoReg	30	XOR	TwoReg	46	MULi	OneReg
15	FADDi	OneReg	31	XORi	OneReg			

5.2 エミュレータの動作

エミュレータの各メソッド，エミュレーション及びシミュレーション時の各メソッドとパイプラインモジュールの関係図を図 5.2.1 に示す．STRAIGHT エミュレータは，入力された専用バイナリファイルから実行する命令情報を取得するために STRAIGHT ローダが呼び出される．STRAIGHT ローダは入力として与えられている専用バイナリを先頭から読み込み，1 命令 32 ビットとして STRAIGHT エミュレータが管理するメモリに格納する．その後，入力として与えられるスキップ数だけ STRAIGHT エミュレータのスキップ関数が呼ばれる．このスキップ関数では，PC を利用した命令の取得，各命令のデコード，RP とソースオペランドの差分計算によるソースレジスタ値の取得，命令の実行，アーキテクチャステートの更新が行われる．また，ここで使用される STRAIGHT エミュレータが持つ，デコードや命令実行の関数は，スキップが終わった後シミュレータ側からも呼び出され使用される (図 5.2.1).

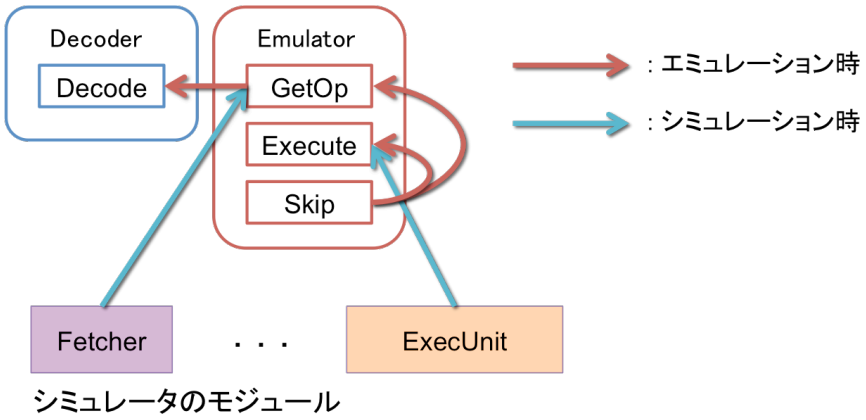

図 5.2.1: 各メソッドとパイプラインモジュールの関係図

図 5.2.1 のようにスキップ関数ではまず，現在の PC の情報を元に STRAIGHT エミュレータが保持・管理するメモリより，格納されている命令を取得するエミュレータの GetOp 関数が実行される．その後，その命令が既にデコード済みであるかどうかをデコード済み命令の配列より判断し，デコードされていない場合は STRAIGHT デコーダによりデコードされる．STRAIGHT デコーダでは図 5.2.2 のように，渡される 32 ビットの命令から先頭の 6 ビットをオペコードとして抜き出す．抜き出したオペコードの値より，その命令の種類を判断し命令形式を決定する．命令形式が定まることにより，第一および第二ソースオペランドがそれぞれ何ビットで格納されているかがわかるので，それぞれの値を取得し格納する．例えば，オペコードが 9 で命令の種類が ADDi であった場合 (表 5.1 参照)，命令形式はソースに一つだけレジスタを使用する OneReg なので，オペコード 6 ビットの後の 7 ビット目から 10 ビットを第一ソースオペランド値として格納する．残った 16 ビットは即値と判断し，第二ソースオペランド値として格納する (命令形式については図 3.2.1 参照).

スキップ関数では，デコード後から命令実行までの間にレジスタ値の決定が行われる．STRAIGHT アーキテクチャステートは，特殊レジスタである RP，SP，FP を保持・管理しているが，その中から現在の RP を取得する．RP の値はその命令のデスティネーションレジスタ番号としてそのまま格納される．ソースレジスタ番号には，現在の RP の値からソースオペランド値を引いた値が格納される．このとき，

$$SrcReg[i] = (RP - SrcOperand[i] + MAX\_RP)\%MAX\_RP \tag{5.2.1}$$

上の式 5.2.1(RP の取りうる最大値を MAX_RP とする) を利用することで，RP が 0 にターンアラウンドした直後でもレジスタを遡って参照できるようにしている．

スキップ関数における命令実行部分ではエミュレータの Execute 関数が呼び出され，ソースレジスタを使用する場合は格納されているソースレジスタ番号のレジスタを参照し値を取得する．その後，オペコードの値からどの命令か判断し取得したレジスタ値，即値を使用することで演算・処理を行う．実行の最後で格納されているデスティネーションレジスタ番号のレジスタに結果が格納される (図 5.2.3)．命令実行が終わるとアーキテクチャステートの更新が行われ，RP の値が 1 インクリメントされる．スキップ関数は，一連の処理が終わった後カウンタを 1 インクリメントする．このカウンタ値が，入力として与えられたスキップ数になるまでスキップ実行はループする．

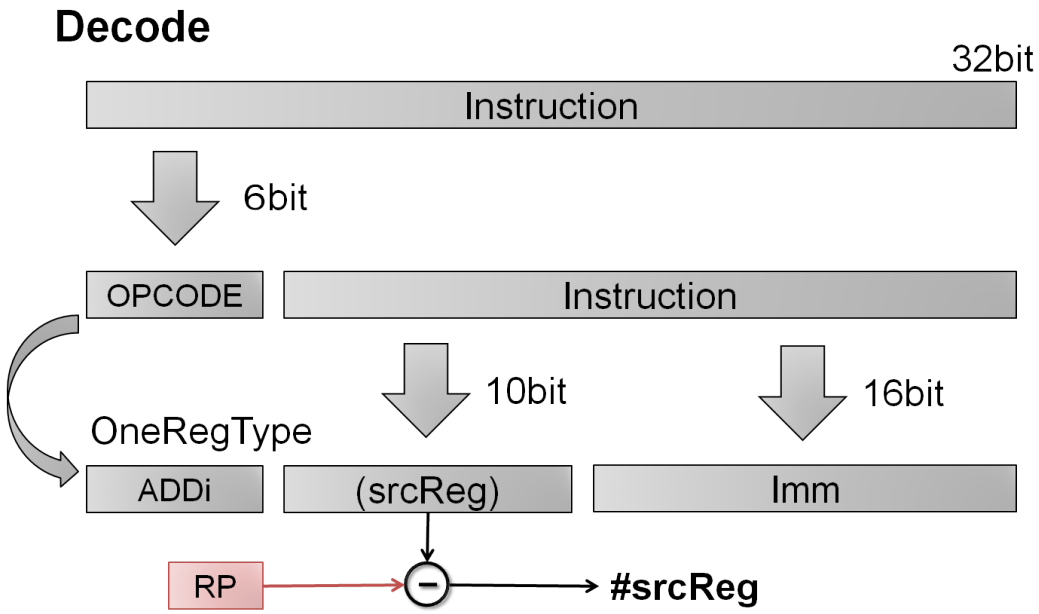

図 5.2.2: デコードの実行例

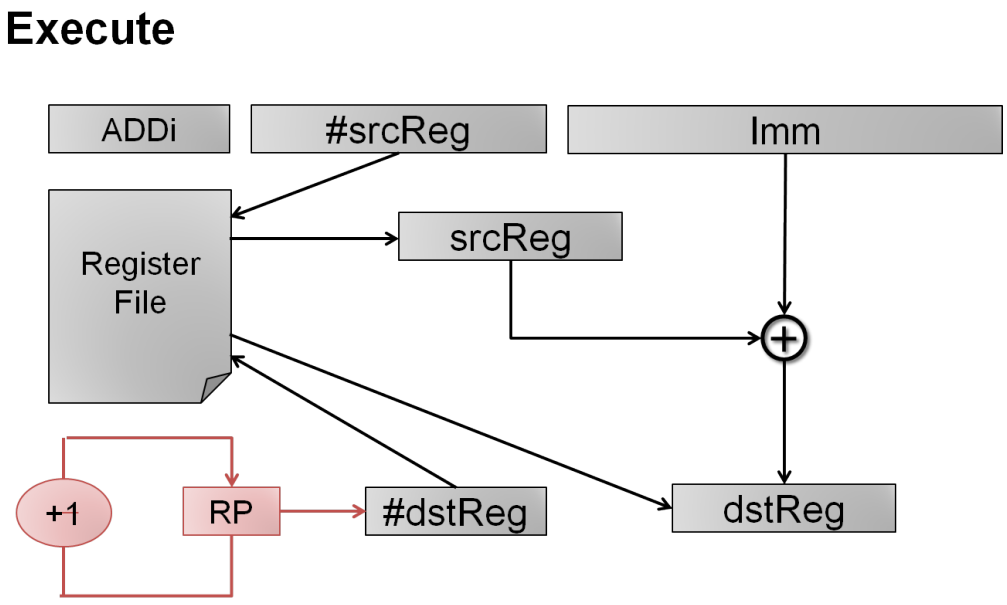

図 5.2.3: エクセキュートの実行例

第6章　パイプラインシミュレータ

6.1　フェッチ・デコード

パイプラインの各ステージと担当するシミュレータモジュール，命令ステータスの関係を図6.2.1 に示す．STRAIGHT シミュレータの Fetcher モジュールは，パイプラインのフェッチステージを担当する．命令がフェッチステージに入ると，命令ステータスは Fetching に設定される．シミュレータの入力としてスキップ数が指定されている場合は，STRAIGHT エミュレータによって STRAIGHT アーキテクチャステートが保持する特殊レジスタ RP，SP，FP の値が変更されているので，それらの値を初回実行時に引き継ぐ．この引き継いだ RP は，１命令生成毎に 1 インクリメントされる．RP の値を ID として，配列に新規の命令を生成する．また，この命令生成時に５章で説明した STRAIGHT エミュレータの GetOp 関数が呼び出され (図 5.2.1)，フェッチと同時にデコードされる．命令の種類やオペランドの値は命令情報としてオプインフォに格納される．命令を生成すると同時に，現在の RP の値をその命令のデスティネーションレジスタ番号としてオプインフォに格納する．さらに，RP の値からソースオペランド値を引くことでソースレジスタ番号を計算しオプインフォに格納する．このとき，エミュレータと同様に式 5.2.1 を使用され，RP が 0 にターンアラウンドした直後でも，正しいレジスタ番号を遡って参照することを可能にしている．命令がリネームステージに入る前に，命令ステータスは Fetched に設定される．

6.2　リネーム

プロセッサシミュレータ鬼斬の Renamer モジュールは，パイプラインのリネームステージを担当し，依存関係解消のためレジスタリネーミングの処理が進められる．しかし，STRAIGHT アーキテクチャでは３章で説明したように，レジスタリネーミングの処理を省略することができるので，STRAIGHT アーキテクチャの Renamer モジュールは，前段のパイプラインから渡された命令を，後段のパイプラインに命令を渡す．命令がリネームステージに入ると，命令ステータスは Renaming に設定され，リネームステージが終わると，命令ステータスは Renamed に設定される．

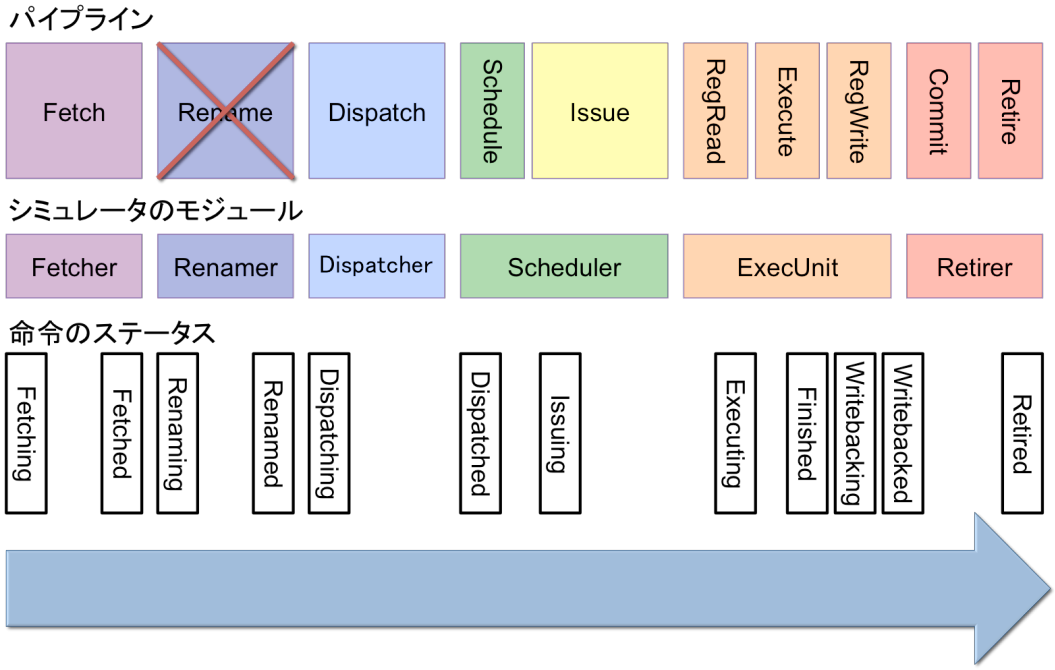

図 6.2.1: STRAIGHT アーキテクチャのパイプラインステージ，シミュレータモジュール，命令ステータスの関係

6.3 ディスパッチ・イシュー・スケジュール

6.3.1 Dispatcher・Scheduler の動作

STRAIGHT シミュレータの Dispatcher モジュールはパイプラインのディスパッチステージを担当し，Scheduler モジュールはイシュー，スケジュールステージを担当する．命令がディスパッチステージに入ると，命令ステータスは Dispatching に設定される．Dispatcher は Scheduler に空きがあるかを判断し命令を Scheduler にディスパッチする．ここで，スケジューラが複数ある場合は，命令情報のスケジューラインデックスを取得し，その ID に対応するスケジューラのパイプラインに命令を登録する．Scheduler はディスパッチされてきた命令を受け取り，演算器にこれからセレクトが始まることを通知する．これにより，パイプライン化されている演算器はカウンタをリセットする．セレクトでは，準備ができている命令の中から，最も古い命令が選択される．選択された命令がセレクト可能であれば，レジスタウェイクアップする．可能でない場合は再スケジュールイベントが登録される．Scheduler がディスパッチされた命令を受け取ったとき，または命令が再スケジュールされたとき，命令ステータスは Dispatched に設定される．命令がイシューされると，命令ステータスは Issuing に設定される．

6.3.2　STRAIGHT シミュレータにおける分岐予測ミス時の処理

STRAIGHT シミュレータでは，分岐予測ミスなどで命令の再スケジューリングが必要となった場合，図 6.3.1 のように RP の巻き戻しと依存関係のリセットを行う．分岐予測によって RP の値がインクリメントされながら命令が発行，それぞれの命令に対してデスティネーションレジスタ番号が割り振られている状態で分岐予測ミスが発覚した場合，RP は分岐時点の値に戻し，正しいパスを通る命令に対して新たにデスティネーションレジスタ番号として割り振る．しかし，これだけでは間違っていたパスを通っていた時の依存関係が残っている状態でレジスタが再利用されるため，依存関係のリセットを行う．

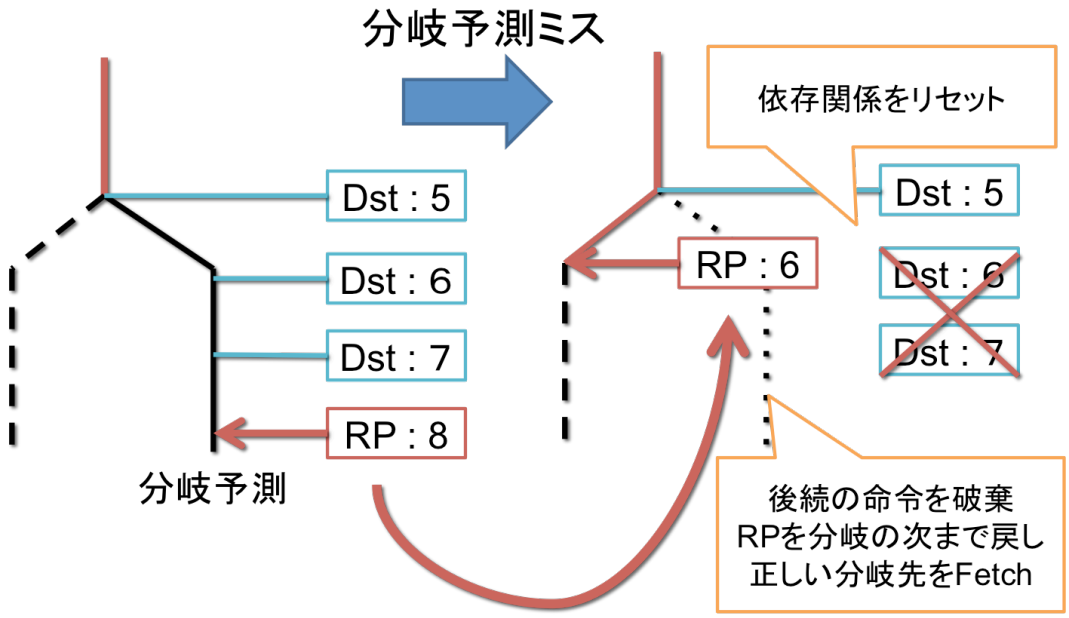

図 6.3.1: 分岐予測ミス時の処理

6.4　レジスタリード・エクセキュート・レジスタライト

STRAIGHT シミュレータの ExecUnit モジュールは，パイプラインのレジスタリード，エクセキュート，レジスタライトステージを担当する．レジスタリードとレジスタライトに関しては，STRAIGHT シミュレータのベースとなる鬼斬プロセッサシミュレータと同様の動作をする．レジスタリードはエクセキュートの最初で，命令情報のソースレジスタ値をキーとしてレジスタファイルを参照することでレジスタ値を受け取る．演算器を使用して実行が行われる前に，命令ステータスは Executing に設定される．前節で説明した Scheduler による，命令実行イベントの登録により，順次命令の実行が行われる．エクセキュートについては，５章でも紹介した STRAIGHT エミュレータの命令実行関数が呼び出される．命令の実行は，表 5.1 の命令セットにある命令で場合分けされ，図 5.2.3 の例

のように動作する．命令の実行が完了すると，命令ステータスは Finished に設定される．レジスタライトに関して実行結果は，実行終了時にデスティネーションレジスタ値として命令情報に追加され，レジスタファイルに書き込まれる．命令ステータスは，ライトバック前に Writebacking に設定され，ライトバックが終わると Writebacked に更新される．

6.5 コミット・リタイア

STRAIGHT シミュレータの Retirer モジュールは，パイプラインのコミット，リタイアステージを担当する．Retirer は，インオーダリストの先頭の命令に対して，リタイア幅未満のリタイアカウントであるか，命令は空でないか，命令ステータスが Finished 以降の段階であるかどうか，を判定する．全ての条件に当てはまっている場合，インオーダリストの先頭の命令を取り出す．その後，コミットが行われ，命令ステータスは Retired に設定される．最後に命令を生成した時の配列から，該当命令を解放し命令のリタイアとなる．リタイアカウントは命令がリタイアされた後，カウントアップされる．命令の種類が分岐であり，PC の次の値が 0 の場合はスレッドが非アクティブに設定され，コミット前にシミュレーションは終了する．STRAIGHT アーキテクチャにおけるバックエンドは 3 章で説明した通りである．

第7章　Livermore Kernel

7.1　Livermore Loops

Livermore Loop は科学技術計算用コンピュータの処理能力を測るために，アメリカ合衆国のローレンス・リバモア研究所で作られたベンチマークプログラムである [39][40]．流体，連立 1 次方程式，三重対角行列の処理，偏微分方程式接解法，差分予測，粒子シミュレーションなど大規模計算の中核コードを集めた 24 個のループコードで構成されており，スーパーコンピュータなどの性能評価に広く利用されている．それぞれの Kernel とループ内容を表 7.1 に示す.

表 7.1: Livermore Loop Kernel とループ内容

Kernel	Description	Kernel	Description
1	hydrodynamics fragment	13	2-D particle in a cell
2	incomplete Cholesky conjugate gradient	14	1-D particle in a cell
3	inner product	15	casual Fortran
4	banded linear systems solution	16	Monte Carlo search
5	tridiagonal linear systems solution	17	implicit conditional computation
6	general linear recurrence equations	18	2-D explicit hydrodynamics fragment
7	equation of state fragment	19	general linear recurrence equations
8	alternating direction implicit integration	20	discrete ordinates transport
9	integrate predictors	21	matrix-matrix transport
10	difference predictors	22	Planckian distribution
11	first sum	23	2-D implicit hydrodynamics fragment
12	first difference	24	location of a first array minimum.

7.2 Kernel 5

本論文では，Kernel 1， 2， 3， 4， 5， 6， 7， 9， 10， 11， 12， 19 の STRAIGHT コードを作成した．例として Livermore Kernel 5 の C 言語コードを図 7.2.1 に示す．また，STRAIGHT アセンブラの入力となる STRAIGHT アセンブリコードを図 7.2.2 に示す．図 7.2.2 に示す STRAIGHT コードは一つのコードであるが，左側の青枠内部の処理は初期値設定の処理であり，右側の赤枠内部の処理が図 7.2.1 に示す C 言語コードと同様の Livermore Kernel 5 の処理である．図 7.2.2 は 1 行 1 命令として，左から行番号，命令の種類，第一ソースオペランド，第二ソースオペランド，コメントを表している．従来のアーキテクチャ Alpha と STRAIGHT アーキテクチャのコードを比べると，STRAIGHT コードは RMOV を使用し差分で前の命令の実行結果を移動することができるので，ストアやロードといったレイテンシの大きい命令を削減することができている．

C言語コード

```
for ( l=1 ; l<=loop ; l++ ) {
        for ( i=1 ; i<n ; i++ ) {
                x[i] = z[i]*( y[i] - x[i-1] );
        }
}
```

図 7.2.1: Livermore Kernel 5 C 言語コード

STRAIGHTコード

```
1. ADDi   0        10000     // loop
2. SPST   1        -8
3. ADDi   0        2
4. ADDi   0        1600      // &x
5. ST     2        1         // x[0]=2
6. SPST   2        -16
7. ADDi   0        3
8. ADDi   0        2400      // &y
9. ST     2        1         // y[0]=3
10.SPST   2        -24
11.ADDi   0        4
12.ADDi   0        3200      // &z
13.ST     2        1         // z[0]=4
14.SPST   2        -32
15.ADDi   0        20        // n=20
16.ADDi   0        0         // i=0
17.NOP
18.SLT    3        2         // i<n
19.BNZ    1        15
20.ADDi   17       0
21.ADDi   17       32        // &x[i]
22.ST     2        1         // x[i]=2
23.NOP
24.ADDi   17       0
25.ADDi   17       32        // &y[i]
26.ST     2        1         // y[i]=3
27.NOP
28.ADDi   17       0
29.ADDi   17       32        // &z[i]
30.ST     2        1         // z[i]=4
31.NOP
32.RMOV   17                 // n
33.ADDi   17       1         // i++
34.J      -16
35.ADDi   0        20        // n
36.SPST   1        -40
```

```
37.SPLD   -8                 // Loop
38.NOP
39.ADDi   0        1         // I
40.NOP
41.SLT    4        2         // loopL
42.BNZ    1        32
43.SPST   4        -48
44.ADDi   0        1         // i
45.MULi   1        32        // &i
46.SPLD   -16                // &x
47.SPLD   -24                // &y
48.SPLD   -32                // &z
49.SPLD   -40                // n
50.ADD    5        4         // &x[i]
51.LD     1        -32       // x[i-1]
52.NOP
53.SLT    9        4         // loopI
54.BEZ    1        16        // End of loopI
55.ADD    8        10        // &y[i]
56.LD     1        0         // y[i]
57.SUB    1        6         // y[i] - x[i-1]
58.ADD    10       13        // &z[i]
59.LD     1        0         // z[i]
60.MUL    1        3         // z[i] * (y[i] - x[i-1])
61.ST     1        11        // x[i] = z[i] * (y[i] - x[i-1])
62.ADDi   18       1         // i++
63.MULi   1        32        // &i
64.RMOV   18                 // &x
65.RMOV   18                 // &y
66.RMOV   18                 // &z
67.RMOV   18                 // n
68.ADD    5        4         // &x[i]
69.RMOV   9                  // x[i-1]
70.J      -17                // back to loopI
71.SPLD   -8                 // Loop
72.SPLD   -48                // I
73.ADDi   1        1         // I++
74.J      -33                // back to loopL
75.NOP                       // EOF
```

図 7.2.2: Livermore Kernel 5 STRAIGHT コード

7.3 STRAIGHTコードの最適化

STRAIGHTアーキテクチャは大規模なレジスタを持っていることから，命令間距離が大きい場合であっても差分指定によりレジスタを確実に参照することができる．そのため，前節でも述べた通り，STRAIGHTコードは命令間の差分指定やRMOVを使用し前の命令の実行結果を利用することで，メモリアクセスするロード命令などレイテンシの大きい命令を削減することができる．STRAIGHTコードにおいてループが1重である場合や，ループを含まないようなコードの場合，関数やループに入る際に変数をメモリ上に退避させる必要がなく，レジスタ移動のRMOV命令を使用することで必要な値を再利用することができる．

一般にLivermore Kernelなどの多重ループが含まれるようなコードでは，全体の命令数が静的に定まらないので，ループで使用されるループインデックスやループの終了条件に使用される変数はメモリにスタックしなければならない．しかし，このような場合においてもSTRAIGHTコードでは，外側ループのループインデックスなどの変数が内側ループ内部で使用されるとき，その変数値をレジスタ移動命令RMOVを使用し内側ループ内に移動させれば，外側ループのループ終了条件から差分指定により静的な命令間距離で参照することができる．そのため，メモリに退避させることなくレジスタの差分表記によって再利用することが可能である．これは，最内ループの分岐終了条件の命令と最内ループ内部の命令の距離が静的であることから実現できる．

STRAIGHTコードには，レジスタ参照距離の調整のため本来実行には不必要な何もしないNOP命令が含まれる．図7.2.2の例で右側の赤枠で囲まれたKernel部分を見ると，外側ループが始まる前に２つ，内側ループが始まる前に１つのNOP命令が挿入されていることが分かる．これはコード全体の命令数からしてみれば少ないが，ループ回数が増えるにつれ本来不必要な動的命令数が増えるということにもなる．

STRAIGHTの命令形式では，ソースオペランドのために10ビット確保しており，最大で2^{10}の値を用いることで2^{10}個前の命令の実行結果を参照することができるが，図7.2.2の例で使用されている最大値は18であり簡単なループコードであれば10ビットのソースオペランドで充分であり，容量不足によってレジスタを差分指定できなくなることは無いことがわかる．

以上のことから，STRAIGHTアーキテクチャのための最適なコードは，なるべく参照距離調整のためのNOP命令を少なくし，かつSTRAIGHTの命令形式を有効利用しメモリアクセスする命令を削減したコードであることがわかる．そのためSTRAIGHTコードにおいては短くまとめられた動的命令数が多いコードではなく，コード側に展開して記述されている静的命令数が多いコードの方が理想的である．例として，Livermore Kernelのような多重ループのコードの場合，ループ展開処理であるループアンローリングの最適化がある．ループアンローリングは本来，ループの分岐回数を削減するために，独立した命令ブロックの連続に書きかえることで，ループ終了までのループ回数を減少させ実行速度を向上させる手法である．STRAIGHTでは，大容量レジスタとその命令形式によって2^{10}個前の命令の実行結果を参照できることから，ループアンローリング段数を容易に増やす(ループ終了までのループ回数を減少させ独立した命令ブロックを増やす)ことがで

きる．STRAIGHT コードにこのループアンローリングの最適化をすると，毎回ループ毎に実行しなければいけない NOP 命令を削減することができる．NOP 命令の削減により同じ処理を行うための命令数は減少する．さらに，ループアンローリング段数を増やすことでコードに記述される静的命令数が増加し，差分表記により実行結果を使いまわしメモリアクセスする命令を削減することができる．STRAIGHT にとってこのようにコード側の表記量 (静的命令数) が増加することは 10 ビットのソースオペランド部分を有効に利用することができるメリットである．

第8章　評価

8.1　評価環境

実装したSTRAIGHTシミュレータ，STRAIGHTアセンブラ，Livermore KernelのSTRAIGHTコードを使用してSTRAIGHTアーキテクチャおよびSTRAIGHTコードの性能評価を行う．ベンチマークには作成したLivermore Kernel 1， 2， 3， 4， 5， 6， 7， 9， 10， 11， 12， 19のSTRAIGHTコードを用いた．また，比較対象となる従来アーキテクチャには，プロセッサシミュレータ鬼斬に初期実装されているAlpha AXPアーキテクチャを使用する．Alphaの入力には，STRAIGHTに使用するベンチマークと同じLivermore KernelのC言語コードをクロスコンパイラを使用しコンパイルしたものを使う．しかし，本研究ではプロセッサシミュレータと同時に公開されているクロスコンパイラを使用することで，Alpha側のLivermore Kernelをコンパイルしたが，最適化オプションを加えると指定命令数のシミュレーションを最後まで完了することができなかったため，最適化は行っていない．STRAIGHTとAlphaのアーキテクチャパラメタを表8.1に示す．STRAIGHTのアーキテクチャパラメタは初期評価[1]から，フロントエンド幅とリタイア幅をAlphaの2倍，スケジューラサイズとレジスタファイルをAlphaの4倍にしたものを使用する．これらを使用し，STRAIGHTアーキテクチャの性能や特徴，STRAIGHTコードの質を詳細に評価する．

表 8.1: アーキテクチャパラメタ

	Alpha	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	7 cycle	5 cycle
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	
L2 キャッシュ	4MB, 16way, 64Bline, 12cycle hit lateny, with stream + stride prefetcher	
メインメモリ	200cycle	

8.2 IPC 比較

作成した Livermore Kernel の全てのベンチマークで 2k 命令スキップ後，1M 命令実行のシミュレーションを行った．実行結果として出力された IPC を図 8.2.1 に示す．ここで IPC は 1 章に記述した通り Instructions Per Cycle の略であり，1 サイクル当たりどれだけの命令を実行することができるかという指標である．横軸はベンチマークとして使用した Livermore Kernel の Kernel 番号，縦軸は IPC を表している．グラフは左から，STRAIGHT シミュレータに STRAIGHT アーキテクチャパラメタを使用したもの，鬼斬シミュレータの Alpha に STRAIGHT アーキテクチャパラメタを使用したもの，鬼斬シミュレータの Alpha に Alpha アーキテクチャパラメタを使用したものである．アーキテクチャパラメタは前節の表 8.1 に示した通りである．

図 8.2.1 を見ると，全てのベンチマークで STRAIGHT アーキテクチャの IPC が Alpha アーキテクチャの IPC を向上させていることが分かる．加えて，STRAIGHT アーキテクチャは鬼斬シミュレータの Alpha に STRAIGHT アーキテクチャパラメタを使用したものと比べてもほとんどのベンチマークで IPC が向上している．一番 IPC を向上させているのは Kernel 19 であり IPC は 3.28 で，STRAIGHT は Alpha から 88%，STRAIGHT パラメタを使用した Alpha から 83%の IPC を向上させている．また，STRAIGHT アーキテクチャの IPC の平均は 2.57 であり，STRAIGHT は Alpha から 29%，STRAIGHT パラメタを使用した Alpha から 23%の IPC を向上させている．

しかしながら，命令セットが変わる場合，単純に IPC のみを用いて性能を比較することはできない．命令セットが違うアーキテクチャの性能を比較するためには，同じ処理に必要な命令数を比較し IPC とその必要命令数を相乗した値の比較が必要になる．

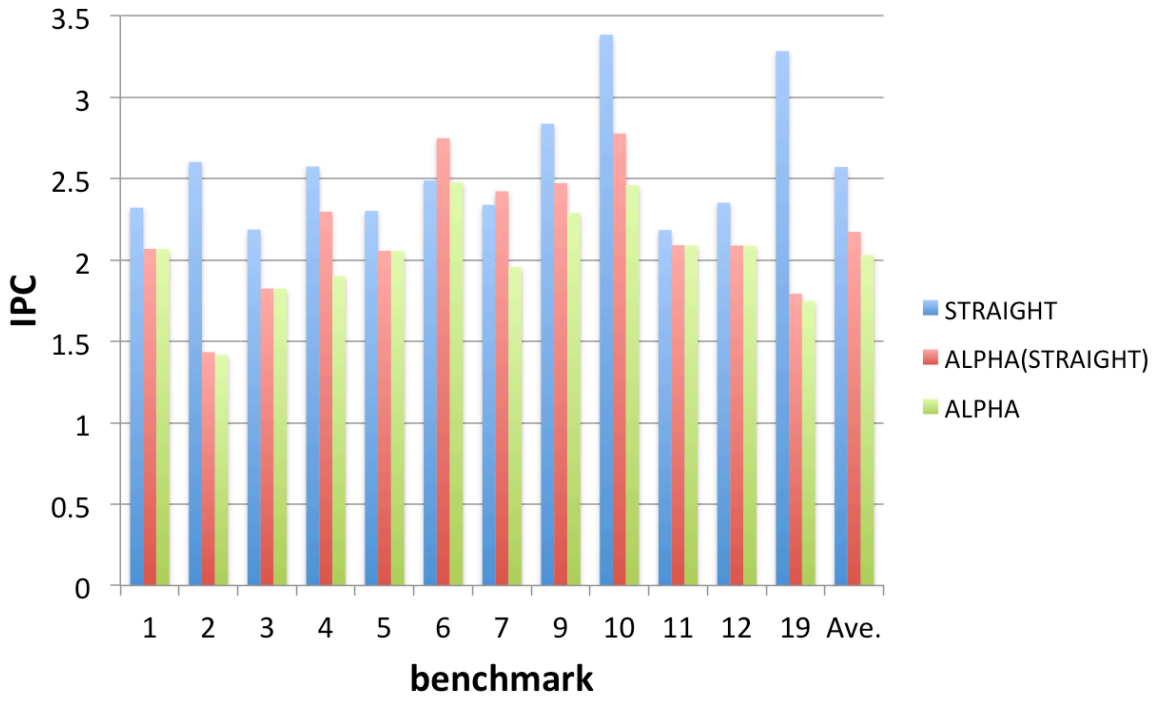

図 8.2.1: 1M 命令実行時の IPC

8.3 STRAIGHT コードの評価

STRAIGHT アーキテクチャ及び STRAIGHT コードを評価するために，それぞれのベンチマークでループ回数を固定し Alpha アーキテクチャとの比較を行った．作成した Livermore Kernel の全てのベンチマークで 1000 ループを実行させ出力結果を得た．また，初期値設定部分を省き Kernel 本体の評価を行うために，1000 ループと 2000 ループを実行し差分の 1000 ループについてを評価対象としている．

図 8.3.1 はそれぞれのベンチマークにおける 1000 ループ実行時の命令数とサイクル数，図 8.3.2 はそれらの Alpha に対する STRAIGHT の割合を示している．図 8.3.2 を見ると，全てのベンチマークにおいて 1000 ループ実行時の命令数とサイクル数は，Alpha と比べて STRAIGHT の方が少ないことが分かる．Kernel 10 では，Alpha と比べて STRAIGHT の命令数は 11%であり，約 90%の削減，サイクル数についても 9%であり，約 90%を削減している．全てのベンチマークの平均では，命令数が 44%，サイクル数が 41%となっており，それぞれ半分以上を削減していることが分かる．これは，STRAIGHT では Alpha と同じ処理を行う場合に必要な命令数が半分以下であることを意味している．

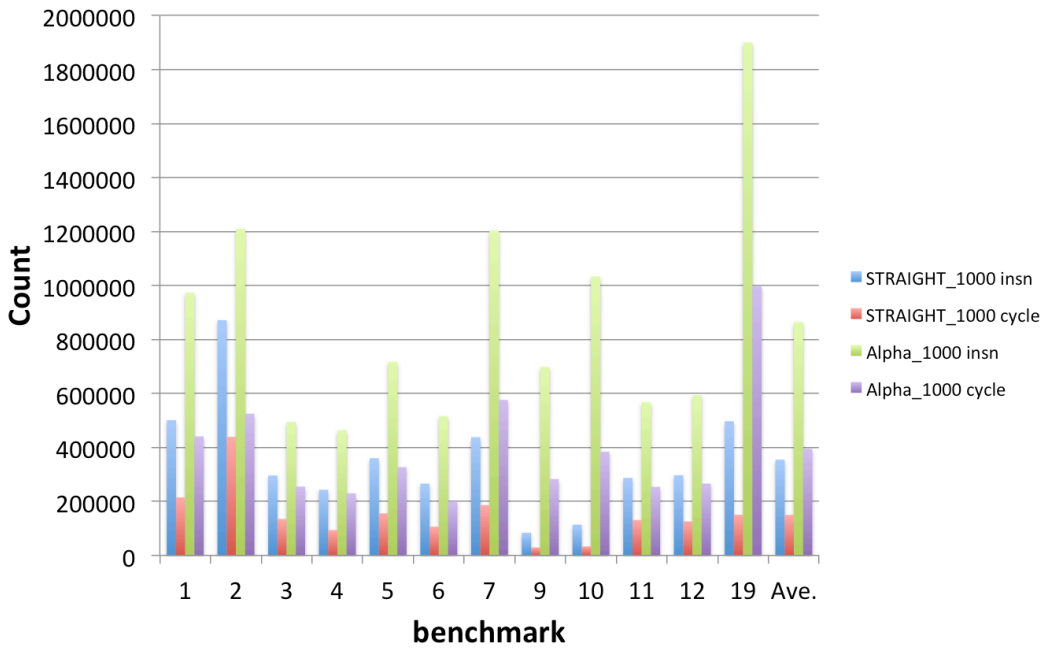

図 8.3.1: 1000 ループ実行時の命令数とサイクル数

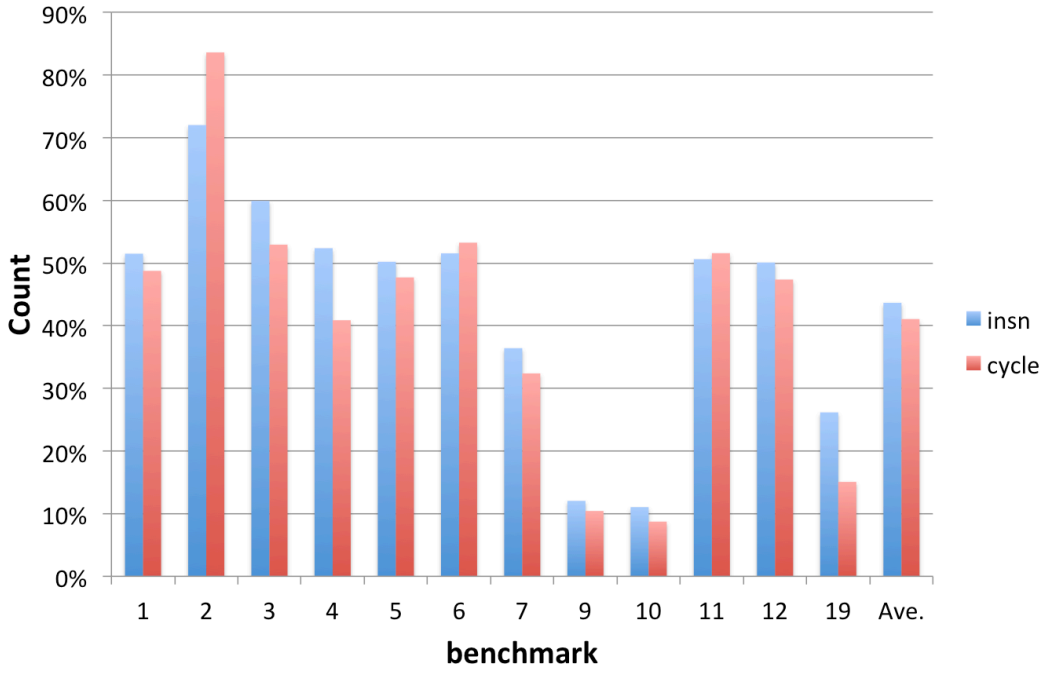

図 8.3.2: Alpha に対する STRAIGHT の 1000 ループ実行時の命令数とサイクル数の割合

図 8.3.3 に STRAIGHT における 1000 ループ実行時のロード・ストア数，図 8.3.4 に Alpha における 1000 ループ実行時のロード・ストア数を示す．また，1000 ループ実行時のロードとストアの合計であるメモリ使用命令数について，Alpha に対する STRAIGHT の割合を図 8.3.5 に示す．図 8.3.3，図 8.3.4，図 8.3.5 を見ると，全てのベンチマークにおいて 1000 ループ実行時のロード・ストア数は，Alpha と比べて STRAIGHT の方が少ないことが分かる．Kernel 10 では，Alpha と比べて STRAIGHT のロード数は 10%，ストア数は 21%であり，ロードとストアの合計のメモリ使用命令数は 13%で 87%を削減している．全てのベンチマークの平均で Alpha と比べて STRAIGHT は，ロード数が 26%，ストア数が 33%であり，ロードとストアの合計のメモリ使用命令数は 26%である．このことから，STRAIGHT アーキテクチャは Alpha アーキテクチャと同じ処理をするときメモリ使用命令を 7 割削減することができ，これが IPC の向上に繋がることがわかる．

従来のアーキテクチャではソースオペランドは 5 ビットであるが，STRAIGHT の命令形式ではソースオペランドに 10 ビットを割り当て 2^{10} 命令前の結果を指定できるようにしている．そこでその 10 ビットを有効利用できているか調べるために，ソースオペランドの値 (レジスタ変位値) が従来アーキテクチャの 5 ビットで指定できる最大値の 31 より大きい命令をカウントした．図 8.3.6 は，STRAIGHT における 1M 命令実行中の RMOV と変位 32 以上の命令数を示している．図 8.3.6 を見ると，Kernel 7 では変位 32 以上の命令が 27%あり STRAIGHT の命令形式を有効に使うことができているが，半分以上のベンチマークでは変位 32 以上の命令が全く無く，ソースオペランド値に 6 ビット以上使っていないことがわかる．また，レジスタ移動命令である RMOV は，全てのベンチマークの平

均で 18%であった.

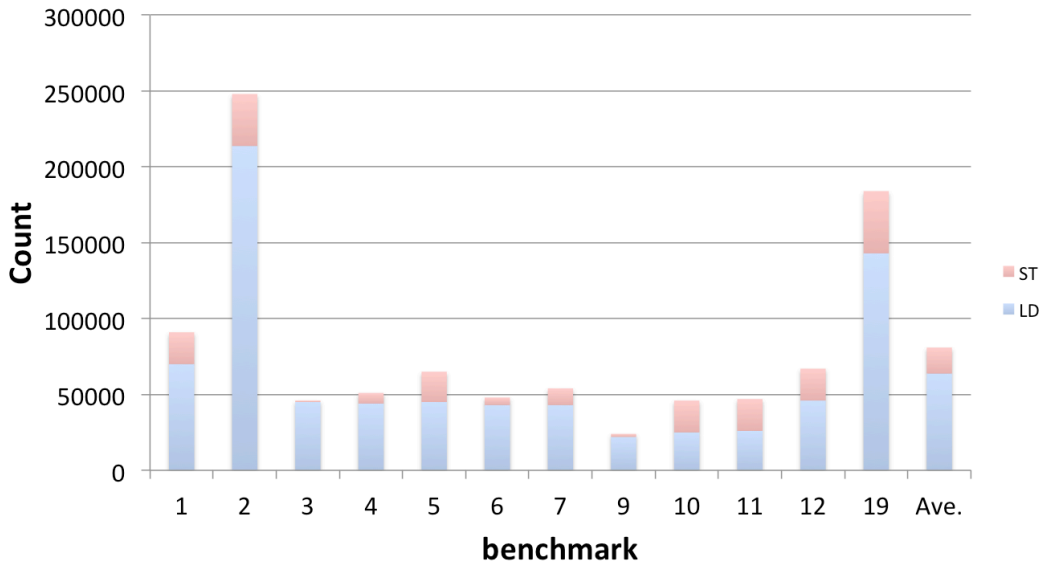

図 8.3.3: STRAIGHT における 1000 ループ実行時のロード・ストア数

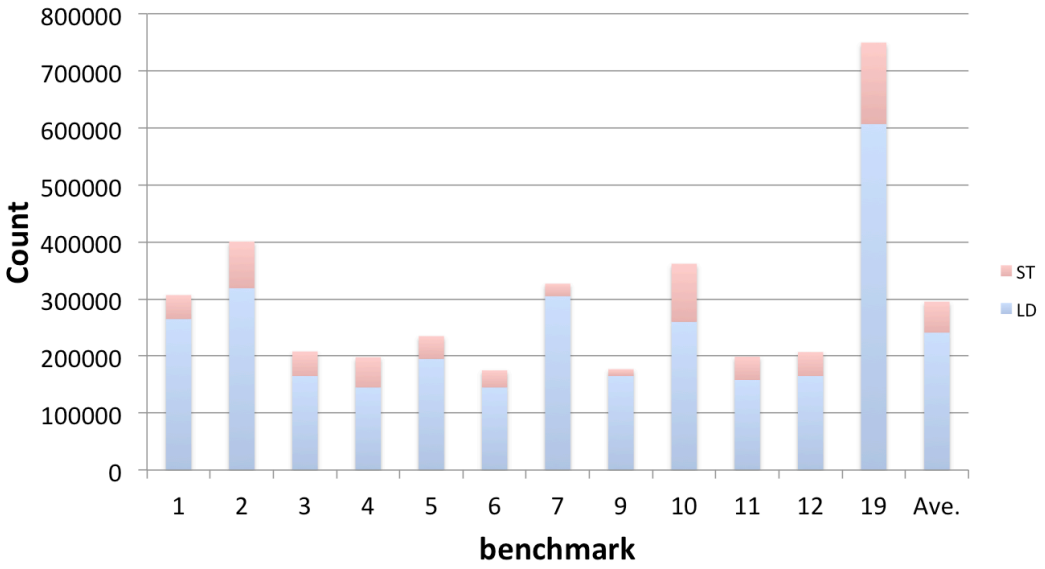

図 8.3.4: Alpha における 1000 ループ実行時のロード・ストア数

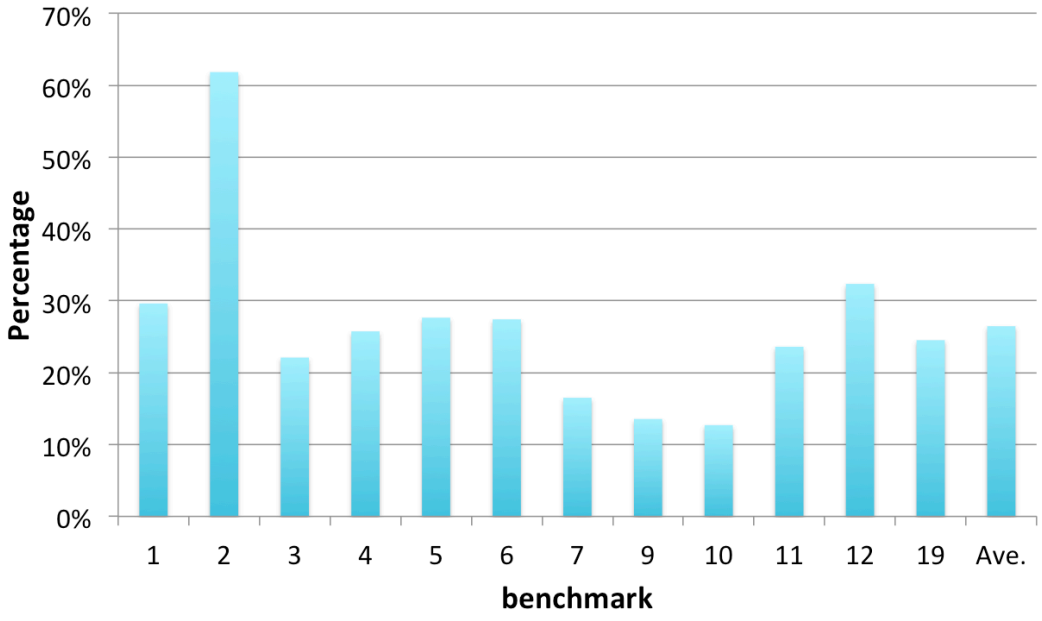

図 8.3.5: Alpha に対する STRAIGHT の 1000 ループ実行時のメモリ使用命令数の割合

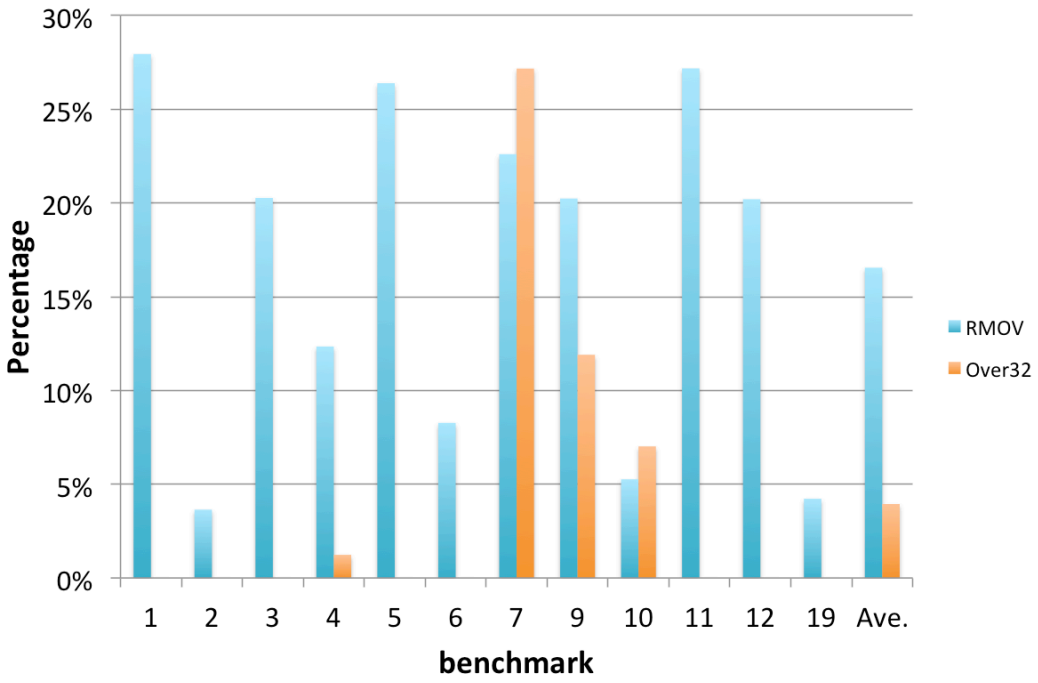

図 8.3.6: STRAIGHT における 1M 命令実行中の RMOV と変位 32 以上の割合

8.4 性能評価

IPC 比較の通り，本論文で設計した STRAIGHT シミュレータを使用した STRAIGHT アーキテクチャと従来の鬼斬シミュレータを使用した Alpha アーキテクチャの比較では IPC が平均で 29%向上した．一般的に本論文で用いた Livermore Loop のようなループコードにおいて，ただ IPC だけを向上させることを考えた場合，より無駄な命令を省いて賢いコードにするよりも，ロードやストアなどメモリアクセスする命令以外の低レイテンシな命令を多くすればよい．そこで，本論文では 1000 ループという与えられた処理を実行することで，IPC の向上と同時に同じ処理に必要な命令数を削減していることを示した．

例えば，IPC が 2.0 であるアーキテクチャA，B があるとしたとき，同じ処理に必要な命令数が A は 10，B は 100 だった場合，A は 5 サイクルで終わるのに対し B は 50 サイクルかかってしまう．また，A は B が終わる前に同じ処理を 10 回行うことができるということにもなる．このことから，STRAIGHT アーキテクチャは従来の Alpha アーキテクチャに対して図 8.2.1 の通り IPC を向上させながら，図 8.3.2 の通り同じ処理に必要な命令数を大幅に削減しているので，実際に IPC で表記される値以上の性能を期待することができる．図 8.4.1 に Alpha と STRAIGHT のそれぞれにおける IPC を，1000 ループ実行時の必要命令数で割った性能を示す．また，その時の Alpha の性能を 1 とした場合の STRAIGHT の性能を図 8.4.2 に示す．IPC については図 8.2.1 の値を，必要命令数については図 8.3.1 の値を使用した．図 8.4.2 を見ると，Alpha に対して STRAIGHT の性能は最大で 12.5 倍，平均で 3.1 倍となっている．従来アーキテクチャと比べて STRAIGHT アーキテクチャは約 3 倍の性能を期待することができる．

一方で，フロントエンド幅を大きくしたことで分岐予測時により多くの命令が先回りし発行されることから，分岐予測ミスが発生した場合に従来より多くの命令がフラッシュされることになる．性能を向上させた中でもこのような余分な電力はまだ存在することが本研究で明らかとなった．

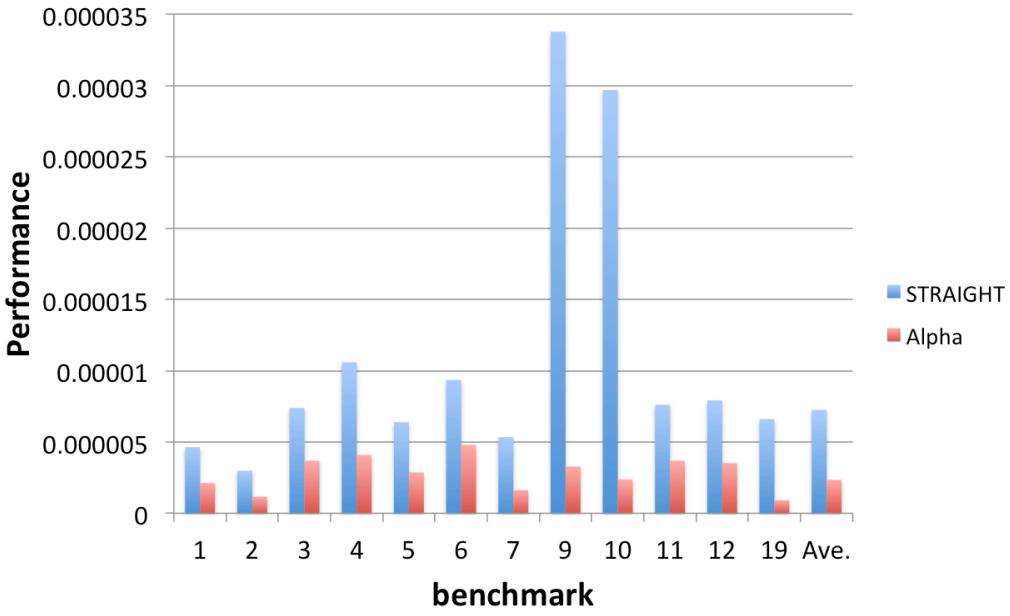

図 8.4.1: IPC と必要命令数を相乗した性能比較

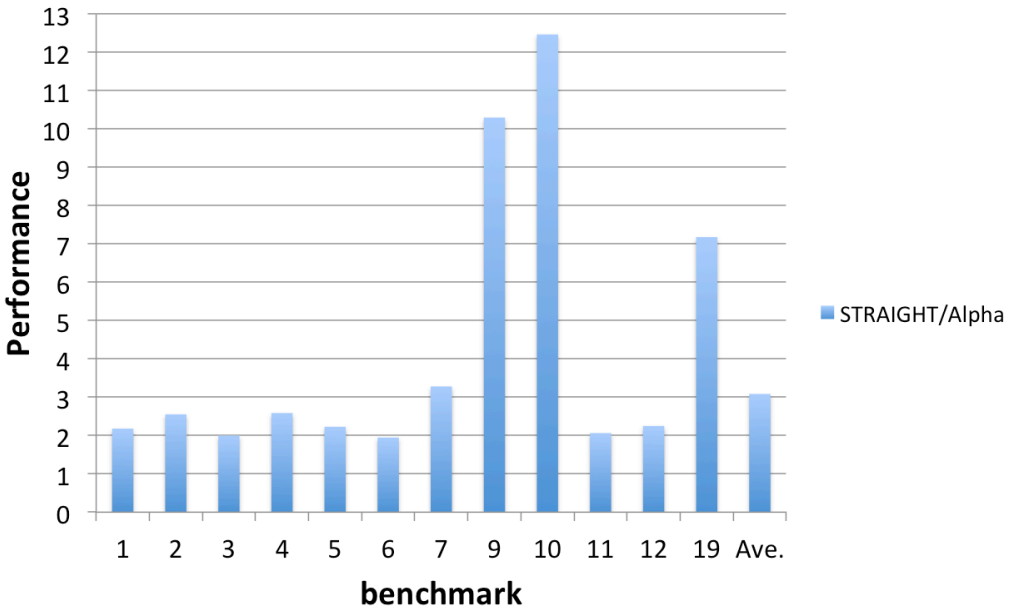

図 8.4.2: Alpha に対する STRAIGHT の相対性能

8.5 ループアンローリング

STRAIGHT コードにおいても従来のアーキテクチャ同様，ループアンローリングすることでどれだけの性能向上が得られるか調べるため，さらに 10 ビットのソースオペランドを有効利用できることを期待し，7 章で取り上げた Livermore Kernel 5 についてループアンローリングの最適化をした．ループアンローリングの最適化については 7 章で紹介した通りである．図 7.2.1 を見てわかる通り，Kernel 5 は 2 重ループなので外側と内側 2 つのループに関してループアンローリングすることができる．ループアンローリングした Kernel 5 を 2k 命令スキップ後 1M 命令実行したときの IPC は図 8.5.1 のようになった．横軸のベンチマークは左から，そのままの Kernel 5，外側ループは変更なし内側に 2 段ループアンローリング，外側ループは変更なし内側に 5 段ループアンローリング，外側に 2 段ループアンローリングと内側に 2 段ループアンローリング，外側に 5 段ループアンローリングと内側に 2 段ループアンローリング，外側に 10 段ループアンローリングと内側に 2 段ループアンローリングである．

図 8.5.1 を見ると，内側のループアンローリングを 2 段にすると IPC がそのままのときの 2.30 から 2.63 に上がっていることがわかる．しかしながら，内側を 5 段にまで増やしても IPC は 2.04 と逆に低下している．これは内側のループ回数がそこまで大きくないのでアンローリング段数を増やしても性能が上がらなかったと推測できる．さらに，内側を 2 段に留めて，外側ループのアンローリング段数を 2 段，5 段，10 段と増やしたところ IPC は，2.80，3.22，3.35 と上がった．特に外側ループのアンローリング段数を 10 段，内側ループのアンローリング段数を 2 段にしたとき，性能は何もしない場合と比べて IPC が 46%上がっている．

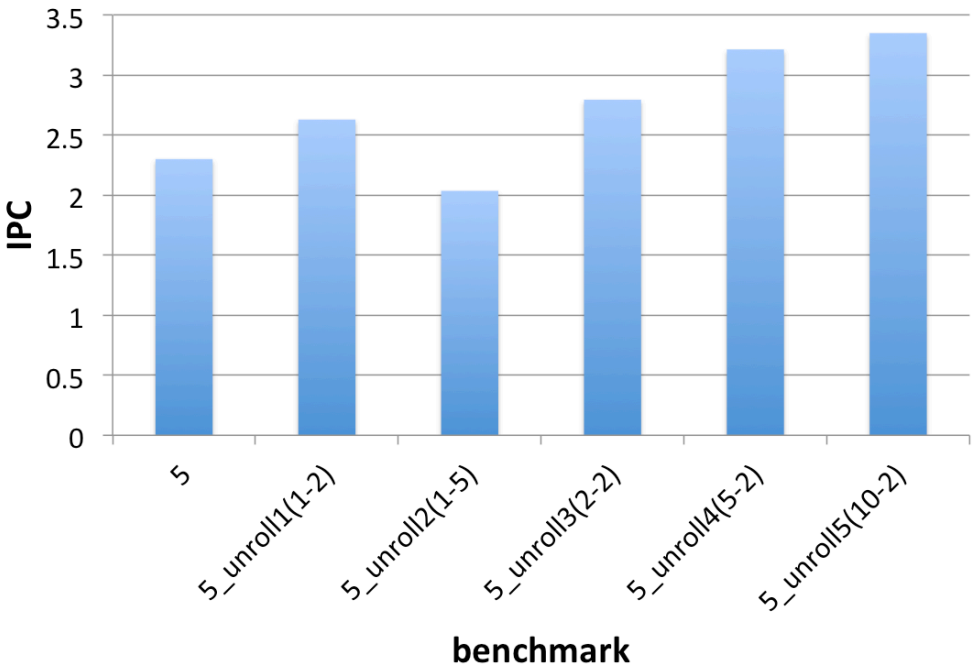

図 8.5.1: ループアンローリングをした Kernel 5 の IPC

次に，ループアンローリングした Kernel 5 を 1000 ループ実行したときの命令数とサイクル数を図 8.5.2 に，その時のロード・ストア数を図 8.5.3 に示す．図 8.5.2 を見るとループアンローリングの段数を大きくする毎に，同じ処理に必要な命令数やサイクル数が削減できていることが分かる．特に図 8.5.1 で IPC が一番高かった外側 10 段内側 2 段については何もしない Kernel 5 に対して，命令数が 35%，サイクル数が 24%になっている．これはループアンローリングによって，1 ループする分岐時に必要な SLT，BEZ，J といった命令やループインデックスのロード・ストア・インクリメントといった命令を削減することができるからである．しかしながら図 8.5.3 をみると，図 8.5.2 の命令数の削減ほどロード・ストア命令は減っていないことが分かる．図 8.5.3 において，外側 10 段内側 2 段についてはメモリ使用命令数が何もしない Kernel 5 に対して 80%となっている．これは上で述べた，ループアンローリングによって削減できる命令中にロードやストアといった命令が多く含まれていないことを意味している．

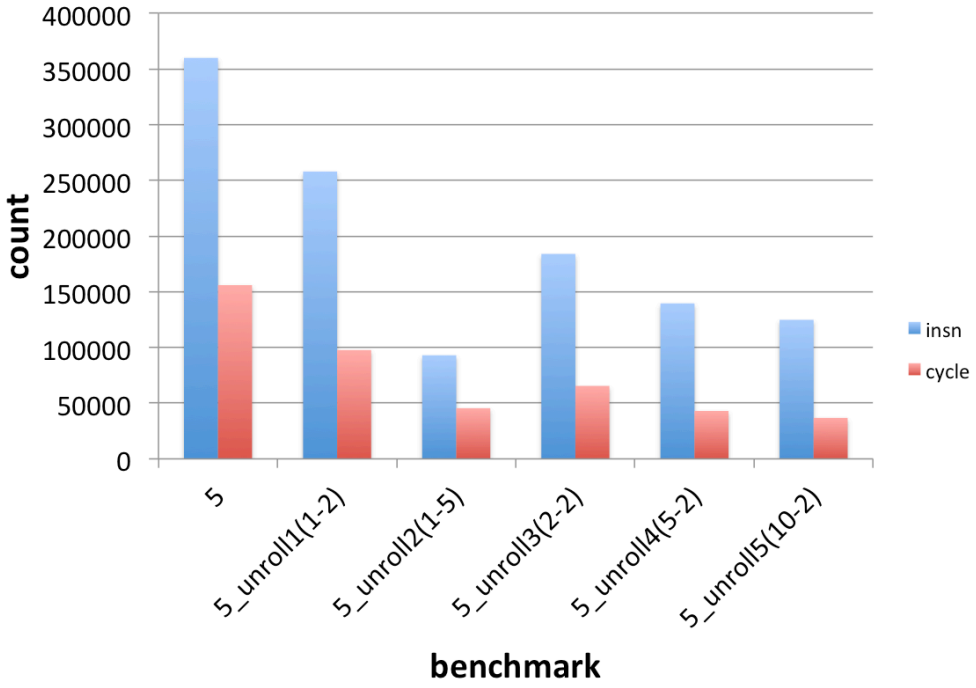

図 8.5.2: ループアンローリングをした Kernel 5 の命令数とサイクル数

ループアンローリングした Kernel 5 を 1M 命令実行したときの，RMOV 数とソースオペランドで指定される変位が 32 以上の命令数を図 8.5.4 に示す．図 8.5.4 を見てわかる通り，ループアンローリングの段数を大きくすると，RMOV を削減できると同時に，変位 32 以上の命令を増加させることができている．特に外側 10 段内側 2 段については，RMOV が何もしない Kernel 5 では命令数全体の 26%であるのに対し 4%と大きく削減できている．また，変位 32 以上の命令は，何もしない Kernel 5 では全く無かったのに対して，命令数全体の 46%と半分近くまで上げることに成功した．

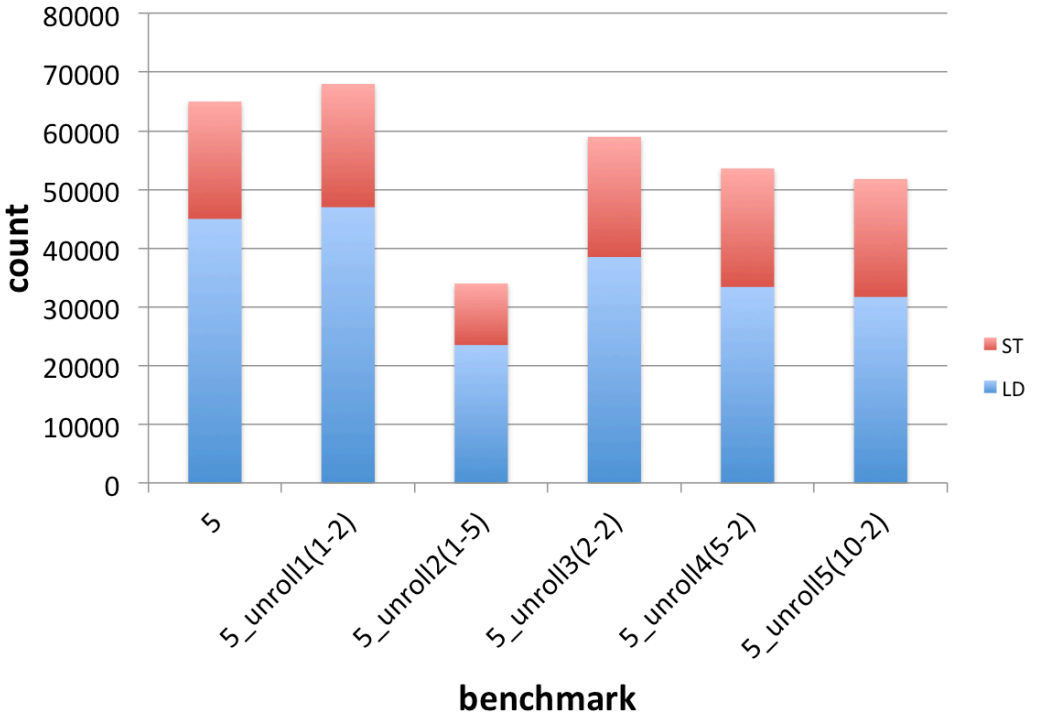

図 8.5.3: ループアンローリングをした Kernel 5 のロード数とストア数

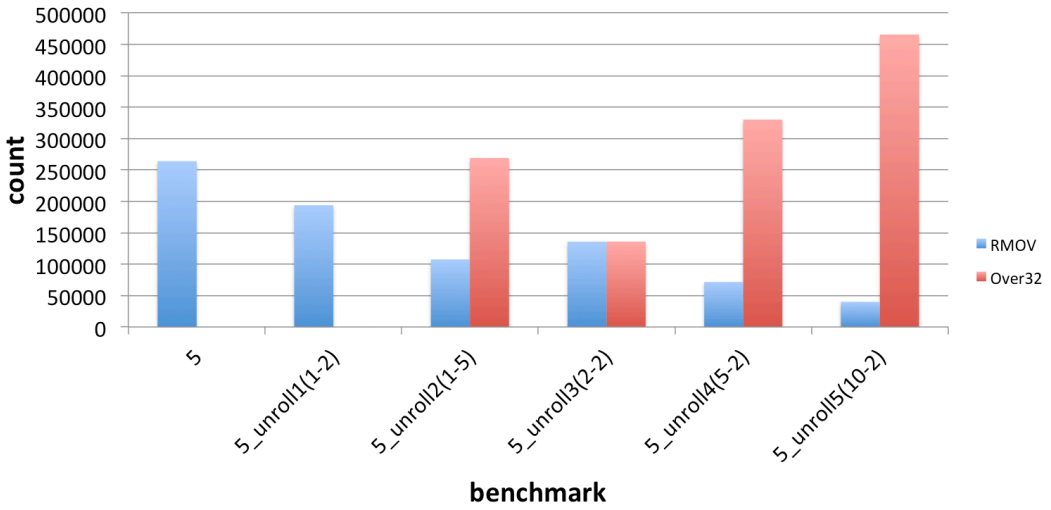

図 8.5.4: ループアンローリングをした Kernel 5 の RMOV 数と変位 32 以上の数

最後に，図 8.5.1 の IPC を，図 8.5.2 の必要命令数で割った総合的な性能を比較する．図 8.2.1 に示す Kernel 5 の Alpha の IPC を，図 8.3.1 に示す Kernel 5 の Alpha の必要命令数で割った値を Alpha の総合的な性能とした．Kernel 5 における Alpha の性能を 1 として，ループアンローリングを適用した STRAIGHT の性能を図 8.5.5 に示す．図 8.5.5 を見ると，ループアンローリングを適用しない場合 2.2 倍に留まっていた性能が，外側ループ 10 段，内側ループ 2 段のループアンローリングを適用することで，9.4 倍にまで向上したことがわかる．

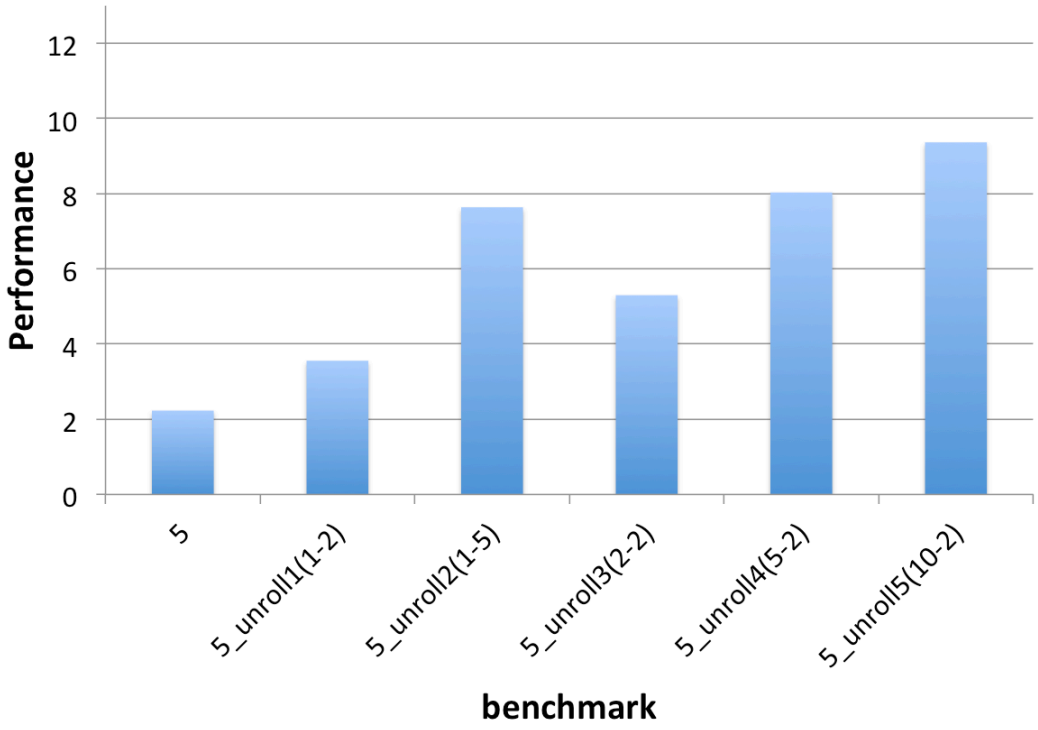

図 8.5.5: ループアンローリング適用時の Alpha に対する STRAIGHT の相対性能

第9章　議論

前章の終わりでは，ループアンローリングによる評価を述べた．本論文ではSTRAIGHTアーキテクチャが従来のアーキテクチャ同様，ループアンローリングによってIPCが向上することを示した．それと同時に，レジスタ移動命令RMOVを削減し，さらに10ビットのソースオペランドを有効利用できるという利点があることを明らかにした．しかしながら，ループアンローリングの最適化については単にアンローリング段数を増やせばIPCが上がるということではなく，Kernel 5の例では内側ループを2段から5段にしたところ，何もしないKernel 5よりIPCが下がってしまった．さらに，図8.5.1では外側ループのアンローリング段数を，2段，5段，10段と増やすとIPCは上がっているが，2段から5段の向上が0.42であるのに対し5段から10段は0.15と減少していることがわかる．このことから，ループアンローリングについては従来のアーキテクチャ同様ループの回数を考慮した最適化が効果的であることがわかる．しかし，ソースオペランドの変位が32以上の命令数が大幅に増えていることはSTRAIGHTアーキテクチャにとって利点に違いなく，以上のことを考慮したSTRAIGHTアーキテクチャにとって最適な専用コンパイラの設計が今後の研究の鍵となる．

3章で述べた通り，STRAIGHTアーキテクチャは管理が単純な大容量レジスタと，拡張の容易なフロントエンド幅によって命令ウィンドウサイズを拡張することが容易である．命令ウィンドウサイズの拡張によりバックエンドパイプラインの稼働率が増加する．しかし，設計・実装したSTRAIGHTシミュレータを用いた評価により，命令ウィンドウ拡張により投機ミスが増加することが明らかとなった．これは，従来アーキテクチャより性能が向上した一方で，分岐予測ミスなどを起こした場合に，分岐予測であらかじめ発行される命令が従来より多くなることが原因である．これにより，レジスタリネーミングの省略等で全体の消費電力は従来より削減できるものの，本来必要ない命令実行への消費電力が残ってしまう．命令ウィンドウ拡張による投機ミスの消費電力削減は今後の課題である．

第10章　まとめ

技術の発展により半導体の微細化が進み，パッケージ内で利用可能なトランジスタの数は増加しているなか，パッケージあたりの電力や熱の制限から，搭載されたトランジスタを同時に駆動することができない，ダークシリコン問題が指摘されている．ダークシリコン問題を解決するためには，IPC の向上と IPC/電力比の向上の両方においてレジスタの管理方法が重要な要素となっている．そこで我々は，トランジスタの増加をレジスタ容量の増加に用いて制御を軽量化する STRAIGHT アーキテクチャを提案している．

本研究は，STRAIGHT アーキテクチャを詳細に評価することを目的とし，STRAIGHT シミュレータの設計，STRAIGHT アセンブラの構築，STRAIGHT 専用 Livermore Loop コードの生成，設計したシミュレータ，生成したコードを用いた IPC と必要命令数の評価，STRAIGHT アーキテクチャと STRAIGHT コードに関する議論を行った．これにより，生成した STRAIGHT 専用の Livermore Loop コードを STRAIGHT アセンブラの入力とすることで，シミュレータの入力となる STRAIGHT バイナリを生成することが可能になった．また，出力された STRAIGHT バイナリと，設計した STRAIGHT シミュレータを用いることで STRAIGHT アーキテクチャを従来より詳細に評価することが可能となった．

設計した STRAIGHT シミュレータと Livermore Kernel を使用した評価では，従来の Alpha アーキテクチャと比べて STRAIGHT アーキテクチャは IPC を最大で 88%，平均で 29%向上させた．さらに，同じ処理を行うために必要な命令数について，従来の Alpha アーキテクチャと比べて STRAIGHT アーキテクチャは最大で約 90%，平均で約 55%の命令を削減することができた．我々は，本当の意味での性能比較をするため，シミュレーションで得た IPC を同じ処理に必要な命令数で割った値を用いて性能比較を行った．IPC と 1000 ループの必要命令数を相乗した性能については，Alpha に対して STRAIGHT は最大で 12.5 倍，平均で約 3 倍の性能であることがわかった．ループアンローリングを用いた評価では，従来コード同様 STRAIGHT コードにおいてもその最適化を適用することで IPC が向上することが分かった．

本論文の評価では Livermore Kernel 5 にループアンローリングを，外側ループに 10 段，内側ループに 2 段適用することで，何もしない Kernel 5 と比べ IPC が 46%向上した．また，ループアンローリングを外側ループに 10 段，内側ループに 2 段適用した場合，STRAIGHT アーキテクチャではレジスタ移動命令 RMOV を命令数全体の 26%から 4%まで削減し，ソースオペランドの変位が 32 以上の命令数をゼロから命令数全体の 46%まで上げることに成功し，STRAIGHT の命令形式を有効利用することができた．Livermore Kernel 5 では，IPC を 1000 ループ実行時の必要命令数で割った性能が，Alpha を 1 としたとき STRAIGHT は 2.2 倍に留まっていた性能を，ループアンローリングの適用によって最大で 9.4 倍まで向

上させることができた．今後はSTRAIGHTアーキテクチャにおいて最適な専用コンパイラの設計，命令ウィンドウ増大による投機ミスの消費電力削減が課題である．

謝辞

本研究を進めるにあたり，ご指導を頂いた主任指導教員の吉永努教授，指導教員の入江英嗣准教授，小川朋宏准教授に感謝いたします．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」 の助成を受けたものであり，支援に感謝いたします．また，STRAIGHT アーキテクチャやシミュレータについて，多くの議論やアドバイスをしていただいた五島正裕教授，塩谷亮太助教に感謝いたします．最後に，互いの研究生活を支えあった研究室の皆様に感謝します．

参考文献

[1] 入江英嗣, 山中崇弘, 佐保田誠, 吉見真聡, 吉永努. 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.

[2] 塩谷亮太. プロセッサ・シミュレータ「鬼斬弐」. http://www.mtl.t.u-tokyo.ac.jp/~onikiri2/wiki/.

[3] Dennis Sylvester and Kurt Keutzer. Getting to the bottom of deep submicron. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-aided Design*, pp. 203–211. ACM, 1998.

[4] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi. Scale-out processors. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pp. 500–511. IEEE Computer Society, 2012.

[5] A. Duran and M. Klemm. The intel many integrated core architecture. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pp. 365–366. IEEE, 2012.

[6] M. Bhadauria, V.M. Weaver, and S.A. McKee. Understanding parsec performance on contemporary cmps. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 98–107. IEEE, 2009.

[7] 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, May 2012.

[8] H. Irie, D. Fujiwara, K. Majima, and T. Yoshinaga. Straight: Realizing a lightweight large instruction window by using eventually consistent distributed registers. In *Networking and Computing (ICNC), 2012 Third International Conference on*, pp. 336–342. IEEE, 2012.

[9] 塩谷亮太, 五島正裕, 坂井修一. プロセッサ・シミュレータ「鬼斬弐」の設計と実装. 先進的計算基盤システムシンポジウム, pp. 120–121. SACSIS2009, 2009.

[10] Jacob Leverich, Matteo Monchiero, Vanish Talwar, Parthasarathy Ranganathan, and Christos Kozyrakis. Power management of datacenter workloads using per-core power gating. *IEEE Comput. Archit. Lett.*, pp. 48–51, 2009.

[11] David H. Albonesi, Rajeev Balasubramonian, Steven G. Dropsho, Sandhya Dwarkadas, Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis, Michael L. Scott, Greg Semeraro, Pradip Bose, Alper Buyuktosunoglu, Peter W. Cook, and Stanley E. Schuster. Dynamically tuning processor resources with adaptive processing. *Computer*, pp. 49–58, 2003.

[12] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M.L. Scottt. Integrating adaptive on-chip storage structures for reduced dynamic power. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pp. 141–152, 2002.

[13] M.C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pp. 157–168, 2003.

[14] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pp. 90–101, 2001.

[15] R.I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pp. 218–229, 2001.

[16] R. Balasubramonian, D. Albones, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on*, pp. 245–257, 2000.

[17] A. Buyuktosunoglu, T. Karkhanis, D.H. Albonesi, and P. Bose. Energy efficient co-adaptive instruction fetch and issue. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pp. 147–156, 2003.

[18] J.R. Herkert. Something old and something new. *Technology and Society Magazine, IEEE*, pp. 4–8, 2004.

[19] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pp. 230–239, 2001.

[20] Dan Gibson and David A. Wood. Forwardflow: A scalable core for power-constrained cmps. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pp. 14–25. ACM, 2010.

[21] K. Khubaib, M.A. Suleman, M. Hashemi, C. Wilkerson, and Y.N. Patt. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pp. 305–316, 2012.

[22] Yasuko Watanabe, John D. Davis, and David A. Wood. Widget: Wisconsin decoupled grid execution tiles. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pp. 2–13. ACM, 2010.

[23] P. Paula, M.I. Adam, H.A. David, and A.S. Christine. Flicker: a dynamically adaptive architecture for power limited multicore systems. In *Int. Symp. on Computer Architecture*, pp. 13 – 23, 2013.

[24] M.-H. Haghbayan, A.-M. Rahmani, A.Y. Weldezion, P. Liljeberg, J. Plosila, A. Jantsch, and H. Tenhunen. Dark silicon aware power management for manycore systems under dynamic workloads. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pp. 509–512, 2014.

[25] S. Pagani, H. Khdr, W. Munawar, Jian-Jia Chen, M. Shafique, Minming Li, and J. Henkel. Tsp: Thermal safe power - efficient power budgeting for many-core systems in dark silicon. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2014 International Conference on*, pp. 1–10, 2014.

[26] Y. Kora, K. Yamaguchi, and H. Ando. Mlp-aware dynamic instruction window resizing for adaptively exploiting both ilp and mlp. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 37 – 48, 2013.

[27] Kenneth Czechowski, Victor W. Lee, Ed Grochowski, Ronny Ronen, Ronak Singhal, Richard Vuduc, and Pradeep Dubey. Improving the energy efficiency of big cores. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, pp. 493–504. IEEE Press, 2014.

[28] 五島正裕, グェンハイハー, 縣亮慶, 森眞一郎, 富田眞治. Dualflow アーキテクチャの提案. 並列処理シンポジウム JSPP 2000, pp. 197–204, 2000.

[29] 五島正裕, グェンハイハー, 縣亮慶, 中島康彦, 森眞一郎, 北村俊明, 富田眞治. Dualflow アーキテクチャの命令発行機構. 情報処理学会論文誌 Vol. 42, pp. 652–662, 2001.

[30] 五島正裕. Out-of-Order ILP プロセッサにおける命令スケジューリングの高速化の研究. 博士論文 京都大学, 2004.

[31] 一林宏憲, 入江英嗣, 五島正裕, 坂井修一. 逆 Dualflow アーキテクチャ. 情報処理学会研究報告計算機アーキテクチャ, pp. 1–6. ARC, 2007.

[32] Ang-Chih Hsieh, TingTing Hwang, Ming-Tung Chang, Min-Hsiu Tsai, Chih-Mou Tseng, and Hung-Chun Li. TSV redundancy: Architecture and design issues in 3D IC. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pp. 166–171, 2010.

[33] Subbarao PALACHARLA. Complexity-effective superscalar processors. *Proc. 24th Int. Symp. on Computer Architecture, 1997*, pp. 206–218, 1997.

[34] 五島正裕, 西野賢悟, ハイハーグェン, 縣亮慶, 中島康彦, 森眞一郎, 北村俊明, 富田眞治. スーパースケーラのための高速な動的命令スケジューリング方式.

[35] Peter G. Sassone, Jeff Rupley, II, Edward Brekelbaum, Gabriel H. Loh, and Bryan Black. Matrix scheduler reloaded. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pp. 335–346. ACM, 2007.

[36] Sheng Li, Jung-Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 469–480, 2009.

[37] Richard L. Sites. Alpha axp architecture. *Commun. ACM*, pp. 33–44, 1993.

[38] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, pp. 24–36, 1999.

[39] Frank H. McMahon. The Livermore Fortran Kemels: A Computer Test of the Numerical Performance Range. *Lawrence Livermore Laboratory technical report LLNL UCRL-53724*, 1986.

[40] Frank H. McMahon. The livermore fortran kernels test of the numerical performance range. In *Martin JL (ed) Performance evaluation of supercomputers*, pp. 143–186, 1988.

発表論文

[1] 佐保田誠, 山中崇弘, 吉見真聡, 吉永努, 入江 英嗣 (2014): STRAIGHT シミュレータによるループ実行の評価. In: 情報処理学会 第 204 回 計算機アーキテクチャ研究発表会（ARC）, pp. 1, Information Processing Society of Japan (IPSJ), 2014.

[2] 入江英嗣, 山中崇弘, 佐保田誠, 吉見真聡, 吉永努. 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.

[3] Midoriko Chikara, Makoto Sahoda, Tomohiro Inaba, Masato Yoshimi, Tsutomu Yoshinaga, Hidetsugu Irie,. “Pre-promotion: Synergizing Prefetching and Anti-thrashing Replacement Policy,” In Proc. 42th Int. Symp. on Computer Architecture (ISCA).（投稿中）