修士論文

題目

レジスタ書込ポート予測を用いた小面積レジスタファイルの研究

指導教員

近藤　利夫　教授

平成 2016 年度

三重大学大学院 工学研究科 情報工学専攻
コンピュータ・アーキテクチャ研究室

川島　弘晃 (414M507)

内容梗概

高性能なアウトオブオーダー (out-of-order: OoO) 型スーパースカラプロセッサ，すなわち複数の命令を同時にロードし，依存の無い命令を同時に実行するプロセッサは，多ポートのレジスタファイルが必要不可欠となっている．一般にレジスタファイルを構成する SRAM の回路面積はポート数の 2 乗に比例するためレジスタファイルはその容量のわりに非常に大きな面積を占める．回路面積の増大は，チップ製造コストだけではなく，アクセスタイム及び，消費電力の増大につながる．しかし，近年の物理レジスタ方式の OoO スー パースカラプロセッサは，より高い性能を達成するために命令ウィンドウサイズや同時発行命令数を増加させる傾向にあり，より多ポートのレジスタファイルを必要としている．本研究では，FPGA や安価な ASIC の設計フロー上で OoO スーパースカラプロセッサを実装することを想定しているが，そのような設計フローでは多ポートのマルチポートメモリを実装することは非常に困難である．そこで，リードポート・ライトポートを 1 つずつ持つ SRAM を多重化して多ポートのメモリを作成する手法が広く用いられている．しかし，SRAM の多重化手法は多ポートのメモリになるほど大量の SRAM を必要とする傾向があるためレジスタファイルのような多ポートのメモリは多重化によるオーバーヘッドが非常に大きい．このオーバーヘッドを減らす方法の一つとしてバンクメモリが挙げられる．バンクメモリとは，2 つもしくはそれ以上の領域にデータを分割し，管理するメモリのことであり，この領域 1 つ分のことをバンクと呼ぶ．バンクメモリは，1 つのバンクに複数のメモリアクセスが集中しないかぎりマルチポートメモリを等価な働きをすることができる．しかし，複数のメモリアクセスが同一バンクに集中すると競合が発生し，メモリアクセスを 1 つずつ処理しなければならず性能が低下してしまう．この競合をバンクコンフリクトと言う．レジスタファイルにバンクメモリを適用した場合，バンクコンフリクトが発生すると 次の命令が演算できずに性能が大きく低下する危険性がある．一般に高性能プロセッサでは，命令実行の並列性を高めるためにレジスタ番号を動的にリネームする．バンク型レジスタファイルを最も単純に実装する場合，同時にリネームされた命令の書込先にそれぞれ異なるバンクを割り当てるような実装になるが，実際には命令の実行サイクル数が異なっていたり，先行する命令との依存関係により実行タイミングがずれる

ため，バンクコンフリクトの発生回数を大きく低減することはできない．そこで本研究は，バンクコンフリクトによる性能低下を改善するためレジスタ書込ポート予測を提案する．これは，同時にリネームされる命令列の依存関係に着目し，同時にリネームされるが，同時には実行できない命令の書込先バンク番号を記憶する．記憶したバンク番号をもとに次の命令列の書込先バンクを割り当てることによりバンクコンフリクトの回避を狙う．また本研究は，提案する予測機構を HDL で 詳細設計した上で評価を行った．評価結果によると，書込予測を使用しないバンクレジスタファイルよりも予測を用いた方がベンチマークプログラムの実行時間を最大 3.9%，平均 0.8%減 少させることに成功した．また，予測機構実装を実装したプロセッサの消費電力を 11.11%削減し，レジスタファイルの面積を 77.6%，アクセスタイムを 8.3%削減することができた．

Abstract

A large multi-port register file is an indispensable component to achieve higher computing performance, especially in recent out-of-order superscalar processors which fetches multiple instructions and execute multiple instructions in one cycle. The number of SRAMs ports effects to not only fabrication cost of a VLSI chip but also circuit scale, access latency and power consumption significantly. Although those reasons, recent superscalar processors require a larger multi-port register file to achieve high performance. In this paper, we assume that superscalar processor is implemented by FPGA or reasonable ASIC, these design flow cannot implement a large multi-port memory easily. Therefore, SRAM multiplexing method is widely used. However, SRAM multiplexing method tends to use huge amount of SRAM, if we want to implement a large multi-port memory. A bank memory can reduce multiplexing overhead. A bank memory diveds data across two or more memories. A bank memory can work as multi-port memory if two or more memory access operations concentrate into a specific bank. However, if multiple memory access concentrates to a specific bank, bank conflict occurs. Bank conflicts cause significant performance degradation because succeeding instructions can not execute until bank conflict is eliminated. In general, high performance processor dynamically rename instructions to exploit instruction level parallelism. If designer implements banked register file simply, destination of renamed instructions is allocated different bank, respectively. However, the number of bank conflict cannot reduce significantly because simple allocation method does not consider instruction's execution latency and instruction dependency. Therefore, we propose a register write back port prediction mechanism to reduce performance degradation caused by bank conflict. Our proposed mechanism stores destination bank of dependency instruction. Using this information in physical register allocation, our proposed mechanism aim to reduce bank conflict. This paper also implements the proposed prediction mechanism into a superscalar processor and estimates performance, access latency, circuit scale, and power consumption. According to the result, our pro-

posed mechanism can reduce execution cycles by 3.9% in maximum and 0.8% in average. Proposed mechanism also reduces power consumption, circuit scale and access time by 11.11%, 77.6% and 8.3%, respectively.

目 次

1 はじめに **1**

2 アウトオブオーダー型スーパースカラプロセッサの概要 **5**
2.1 スーパースカラプロセッサのアーキテクチャ 5
2.1.1 パイプライン処理 6
2.1.2 スーパースカラ 9

3 SRAM 多重化手法とバンクメモリ **10**
3.1 SRAM の多重化手法 10
3.1.1 N-リードマルチポートメモリ 10
3.1.2 N-write マルチポートメモリ 11
3.1.3 N-read，N-write マルチポートメモリ 12
3.2 バンクメモリ 14
3.2.1 バンクコンフリクト 14

4 関連研究 **17**

5 バンク競合低減手法の提案 **19**
5.1 マイクロアーキテクチャの拡張 19
5.2 レジスタ書込ポート予測の詳細 25

6 評価 **30**
6.1 評価環境 30
6.2 性能評価 31
6.3 アクセスタイム評価 32
6.4 面積評価 33
6.5 電力評価 34

7 結論 **36**

謝辞 **37**

参考文献 **38**

図 目 次

2.1 標準的なパイプライン構造 5
2.2 スーパースカラプロセッサの構造 5
3.3 リードポートの多重化 11
3.4 ライトポートの多重化 12
3.5 リードポート及びライトポートの多重化 13
3.6 4 バンクメモリ 15
3.7 バンクコンフリクトの発生メカニズム 16
5.8 マイクロアーキテクチャの拡張 19
5.9 Free List の問題点 21
5.10 Free List の拡張 22
5.11 Active List 24
5.12 命令列の例と提案アルゴリズムの動作 29
6.13 性能評価 32

表 目 次

6.1 プロセッサの構成 30
6.2 使用した EDA/CAD ツール 31
6.3 アクセスタイム評価 33
6.4 面積評価 33
6.5 RAM 部を除いたプロセッサ全体の回路規模 35
6.6 RAM 部を除いたプロセッサ全体の消費電力 35
6.7 レジスタファイル以外の RAM 部を除いたプロセッサ全体の消費電力 35

1　はじめに

高性能なアウトオブオーダー (out-of-order: OoO) 型スーパースカラプロセッサの構成要素のうち，最も高コストなものの 1 つとしてレジスタファイルが挙げられる．特に，物理レジスタ方式のスーパースカラプロセッサでは命令レベル並列性 (instruction level parallelism: ILP) を活用するために，大容量かつ多ポートのレジスタファイルが必要とされる．また近年では，さらなる ILP 活用のために命令ウィンドウサイズや同時発行命令数を増加させる傾向にあるため，レジスタファイルのエントリ数・ポート数も増加している．

レジスタファイルは多ポートの SRAM で構成されており，通常 1 命令あたり，2 つのリードポートと 1 つのライトポートが必要となる．従って，4 つの命令を同時に実行するスーパースカラプロセッサのレジスタファイルのポート数は合計 12 にもなる．SRAM の回路面積は，ポート数の 2 乗に比例するためレジスタファイルはその容量のわりに非常に大きなものとなる．SRAM の回路面積の増大は，消費電力やアクセス時間の増大につながる．

特に，FPGA や ASIC の設計フローでは，任意の多ポート SRAM を用意するのは困難である．以下に多ポートの SRAM の作成方法を示す．

1. 論理合成

2. メモリコンパイラ

3. 手設計

4. SRAM の多重化

1. の論理合成では，設計者は RTL レベルで多ポートメモリを設計し，スタンダードセル（DFF や AND/OR ゲート）を用いて論理合成を行う．しかし，論理合成によって生成された多ポートメモリは，回路規模が非常に大きくなるという問題がある．2. のメモリコンパイラでは 3 ポートを超えるような多ポートメモリに対応していない場合やそもそも利用できないような安価な設計フローもある．また，FPGA では多ポートのメモリが組み込まれていないため，メモリコンパイラを使用することはできない．3. の手設計では，設計に膨大な時間がかかってしまう．4. の SRAM の多重化は比較的コストが低く，ほとんどの設計フローにおいて利用できる．そこで，本稿では SRAM の多重化によって多ポートメモリを作成する手法 [1] に注目する．しかし，SRAM の多重化手法では，第 3.1.3 項で述べるように，多重化によるオーバーヘッドが問題となっている．

バンクメモリは多重化によるオーバーヘッドを解決する 1 つの手法であ

る．バンクメモリとは，2つもしくはそれ以上の領域にデータを分割し，管理するメモリのことであり，この領域1つ分のことをバンクという．バンクメモリは，1つのバンクに複数のメモリアクセスが集中しないかぎり理想的なマルチポートメモリと等価な働きをすることができる．しかし，複数のメモリアクセスが同一バンクに集中すると競合が発生し，メモリアクセスを1つずつ処理しなければならず，性能が低下してしまう．この競合をバンクコンフリクトという．レジスタファイルにバンクメモリを適用した場合，バンクコンフリクトが発生すると 次の命令が演算できずに性能が大きく低下する危険性がある．一般に高性能プロセッサでは，命令実行の並列性を高めるためにレジスタ番号を動的にリネームする．バンク型レジスタファイルを最も単純に実装する場合，同時にリネームされた命令の書込先にそれぞれ異なるバンクを割り当てるような実装になるが，実際には命令の実行サイクル数が異なっていたり，先行する命令との依存関係により実行タイミングがずれるため，バンクコンフリクトの発生回数を大きく低減することはできない．

そこで本稿では，バンクコンフリクトによる性能低下を抑えるための機構として，レジスタ書込ポート予測を提案する．レジスタ書込ポート予測によってバンクコンフリクトによる性能低下を抑えつつ，小面積に

レジスタファイルを実装することが可能となる．

以降，本稿は次のように構成する．まず，次章でアウトオブオーダー型スーパースカラプロセッサの概要について，第3章ではSRAM多重化手法について，第4章ではバンクレジスタファイルに関する関連研究について議論する．第5章では提案手法を実装するためのマイクロアーキテクチャの拡張について述べ，第6章では提案手法について詳細な説明を行い，第7章で評価を行う．

2　アウトオブオーダー型スーパースカラプロセッサの概要

本章では，本論文で想定している物理レジスタをベースとするアウトオブオーダー型スーパースカラプロセッサの概要について述べる．

2.1　スーパースカラプロセッサのアーキテクチャ

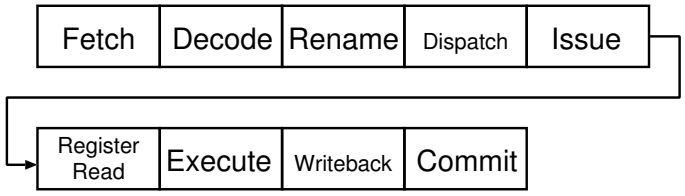


図 2.1: 標準的なパイプライン構造

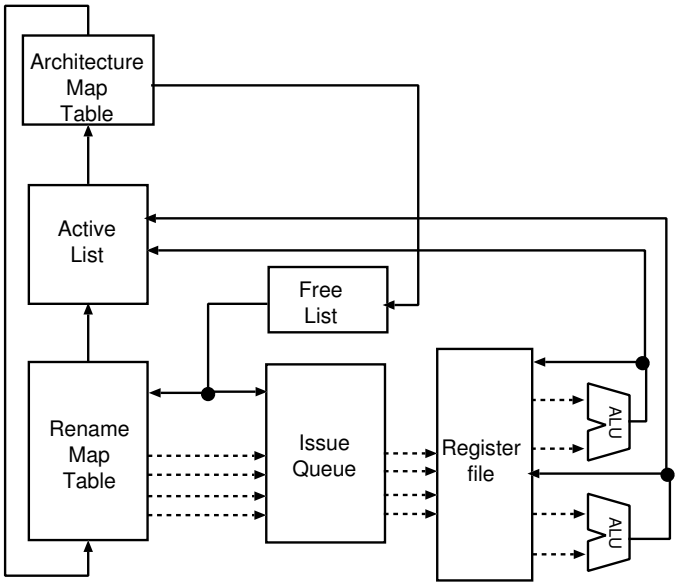


図 2.2: スーパースカラプロセッサの構造

図 2.1 及び図 2.2 は，それぞれ標準的なパイプライン構造と本研究に関連のある部分のスーパースカラプロセッサの構造を示している．図 2.1 及び図 2.2 を用いてスーパースカラプロセッサの基本動作について説明する．

2.1.1　パイプライン処理

一般に高性能なプロセッサは，命令メモリからの読み出し，命令の解析，実行などの処理を同時に 1 サイクル内で行うことはせず，各処理要素を直列に連結し，ある要素の出力が次の要素の入力となるようにして，並行に処理させている．よって，1 サイクル以内に処理すべき要素が減ることで動作周波数の向上につながり，またスループットの向上にもつながる．これをパイプライン処理と呼ぶ．以下に各パイプライン要素の動作の説明をする．

Fetch

図 2.1 に示す Fetch ステージでは，Program Counter（PC）の値に従って命令キャッシュにアクセスし，命令を読み出す．その後，PC を更新する．

Decode

図 2.1 に示す Decode ステージでは，Fetch ステージで読み出した命

令を解析し，命令の種類やオペランドといった情報を得る．

Rename

図 2.1 に示す Rename ステージでは，デコードされた命令のソースレジスタ番号が図 2.2 に示す Rename Map Table（RMT）を参照し物理レジスタ番号に変換される．同時にその命令のデスティネーションレジスタとして新しい物理レジスタ番号が図 2.2 に示す FreeList から割り当てられる．フリーリストとは未使用の物理レジスタを管理している FIFO である．Free List から割り当てられた物理レジスタ番号はデスティネーションレジスタ番号に対応する RMT に登録される．この一連の操作をレジスタ・リネーミングと呼び，レジスタ・リネーミングによって命令間の偽の依存が取り除かれる．

Dispatch

図 2.1 に示す Dispatch ステージでは，命令のコミットを命令順通りに行うためリネームされた命令が図 2.2 に示す Active List に命令順通り登録される．このとき，命令のデスティネーションレジスタ番号と，現在割り当てられている物理レジスタ番号が Active List に登録される．この情報は，例外発生時にアーキテクチャステートを

復元するために使われる．また，全ての命令は図 2.2 に示す Issue Queue に書き込まれる．

Issue

図 2.1 に示す Issue ステージでは，ソースオペランドが揃い，実行可能になった命令からアウトオブオーダーに発効される．

Register Read

図 2.1 に示す Register Read ステージでは，発行された命令のソースオペランドに割り当てられたレジスタを読み出す．

Execute

図 2.1 に示す Execute ステージでは，図 2.2 に示す ALU などの演算器を用いて演算を行う．

Writeback

図 2.1 に示す WriteBack ステージでは，演算結果をデスティネーションに割り当てられたレジスタに書き込み，Active List に命令が完了したことを知らせる．

Commit

図 2.1 に示す Commit ステージでは，完了した命令が Active List か

ら読み出され，その命令のデスティネーションレジスタ番号に対応した図 2.2 に示す Architecture Map Table（AMT）のエントリに割り当てられた新しい物理レジスタ番号を登録し，該当エントリの物理レジスタを解放し Free List に戻す．

2.1.2　スーパースカラ

プロセッサにはスカラ型とベクトル型がある．スカラ型とは原理的に 1 つの命令で 1 つのデータを操作するものであり，ベクトル型とは 1 つの命令で複数個のデータを扱えるものを指す．スーパースカラとはスカラ型の流れを継承しつつ，同時に複数命令を実行できるようにすることで，結果的に複数個のデータも同時に扱えるようにしたものである．

3　SRAM多重化手法とバンクメモリ

本章では，提案手法の説明に先立ち，まずFPGAやASICの設計フローでレジスタファイルのような多ポートメモリを実装する前提条件となるSRAMの多重化手法について説明する．SRAM多重化手法の問題を明らかにした後，その解決法の一つであるバンクメモリについて説明する．

3.1　SRAMの多重化手法

ASICの設計フローにおいて利用されるメモリコンパイラは，3ポートを超えるような多ポートメモリに対応していない場合が多い．そのため，ポート数とエントリ数の組み合わせが異なるSRAMを全て手動で設計する必要があり非常に時間がかかる．また，FPGAではメモリは予め用意された組込みメモリ以外利用することはできない．そこで，1R1WのSRAMの多重化を行い多ポートSRAMを短時間で設計することが出来る手法が用いられる．以下でリードポートの多重化，ライトポートの多重化手法について述べる．

3.1.1　N-リードマルチポートメモリ

図3.3に多重化を用いた2リード1ライトのSRAMのブロック図を示す．リードポートの多重化では，ライトポートへの入力信号は利用する2

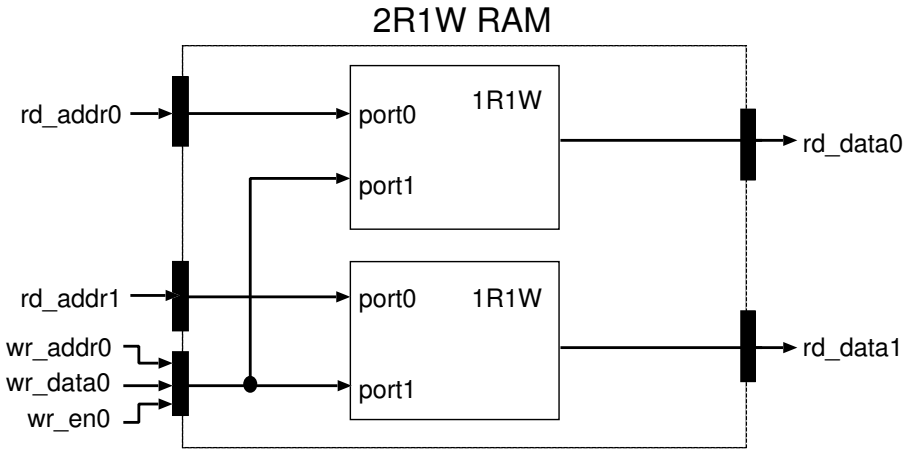


図 3.3: リードポートの多重化

ポート SRAM で共有するように繋ぎ，リードポートはそれぞれで異なるように繋ぐことで多重化を行う．よって，二つの SRAM は常に同一のデータを保持することになる．読み出し時には，それぞれの SRAM にリードアドレス信号を送ることで同時に二つのデータを読み出すことが出来る．

3.1.2 N-write マルチポートメモリ

図 3.4 に多重化を用いた 1 リード 2 ライトの SRAM のブロック図を示す．ライトポートの多重化もリードポートの多重化と同じように実装することが出来る．しかし，各アドレスの最新の情報を持つバンクを識別するための MRU メモリ（図 3.4 中の ram_select_vector），および最新のデータを持つバンクからデータを引き出すためのセレクタが追加される．そ

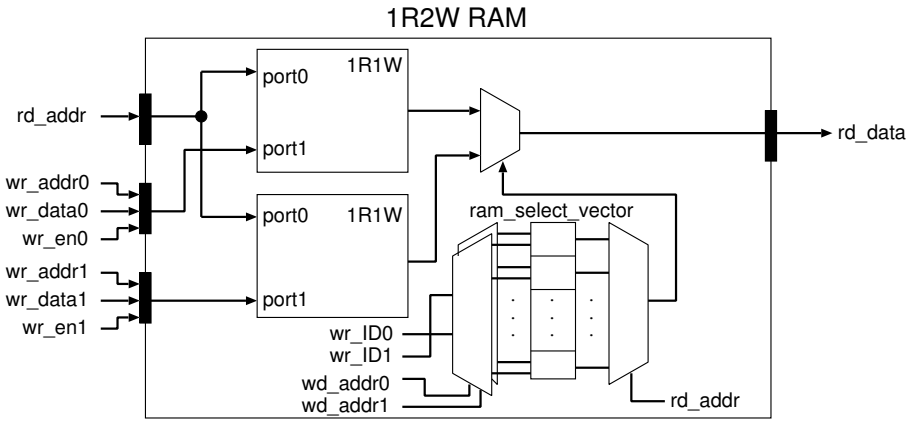


図 3.4: ライトポートの多重化

のため，ライトポートの多重化はリードポートの多重化と比べてコストが高い．

3.1.3 N-read，N-write マルチポートメモリ

図 3.5 に多重化を用いた 2 リード 2 ライトの SRAM のブロック図を示す．多重化に必要な SRAM の個数はリードポートとライトポートの数の積に比例するため，レジスタファイルのような多ポートの SRAM を設計しようとする場合，容量オーバーヘッドが非常に大きい．そこで，本稿はライトポート多重化時に発生するハードウェアオーバーヘッドと多重化による容量オーバーヘッドを削減するためにバンクメモリを用いる．

文献 [2] はライトポート多重化時に発生するハードウェアオーバーヘッドを削減するために XOR 演算を用いている．しかし，容量オーバーヘッ

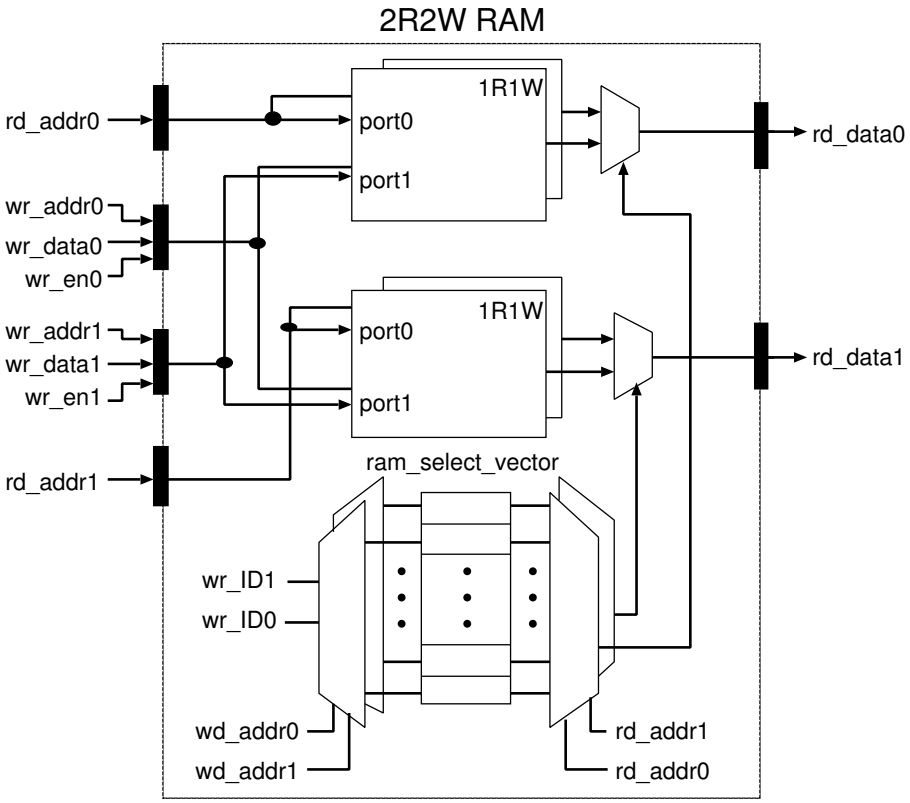


図 3.5: リードポート及びライトポートの多重化

ドを削減することはできない．

3.2 バンクメモリ

図 3.6 右は，データを 4 つの領域に分割する 4 バンクのメモリを示している．図 3.6 左の Index5 へのアクセスは，図 3.6 右の Bank1 の Index1 へのアクセスに対応する．つまり，図 3.6 左の Index の数字を二進数で表したときの下位 2bit がバンク番号を表し，上位 1bit が Index を表すことになる．5 を二進数で表すと 101 となり下位 2bit は 1，上位 1bit は 1 となるので，Bank1 の Index1 へのアクセスとなる．ここで，図 3.6 左はライトポートを 4 つ持つマルチポートメモリであり，図 3.6 右は各バンクがそれぞれライトポートを 1 つ持つとする．図 3.6 左のメモリの Index0，4 には同時に書込可能だが，図 3.6 右の場合 Index0，4 へのアクセスはどちらとも Bank0 へのアクセスになるため，バンクコンフリクトが起こり同時に書込が出来ない．

3.2.1 バンクコンフリクト

バンク型レジスタファイルを最も単純に実装する場合，同時にリネームされた命令の書込先にそれぞれ異なるバンクを割り当てるような実装になるが，実際には命令の実行サイクル数が異なっていたり，先行する

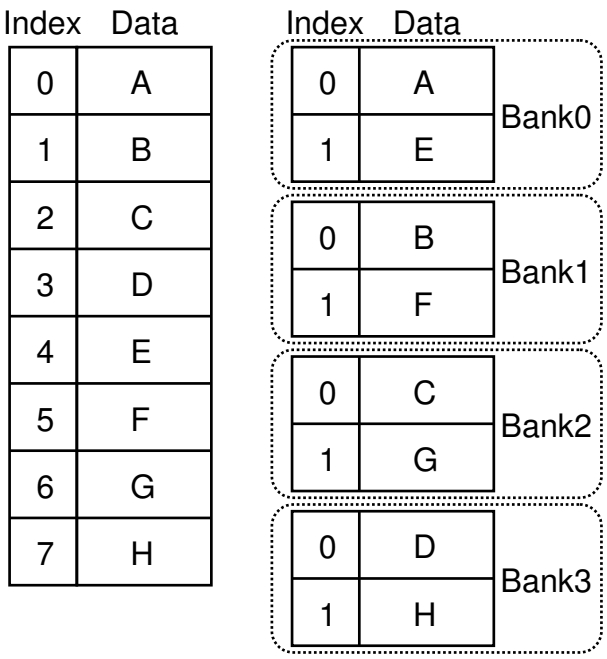


図 3.6: 4 バンクメモリ

命令との依存関係により実行タイミングがずれるため，バンクコンフリクトの発生回数を大きく低減することはできない．これを図 3.7 を用いて説明する．この例では命令の実行サイクル数は全て 1 サイクルとし，同時に 4 命令処理できるプロセッサを使用する．プロセッサは 0〜3 番の命令をリネームし，書込先にそれぞれ異なるバンクを割り当てる．同様に次サイクルは，4〜7 番の命令をリネームし，書込先にそれぞれ異なるバンクを割り当てる．命令間の依存関係を解析すると，図 3.7 中央のように各命令はパイプラインステージを流れていくことがわかるが，ここで 1 番の命令と 5 番の命令に注目すると，同じサイクルでレジスタに書込を行っ

ていることが分かる．2 番と 6 番の命令も同様である．このようにリネーム時に各命令の書込先にそれぞれ異なるバンクを割り当てたとしてもバンクコンフリクトを避けることは出来ない．

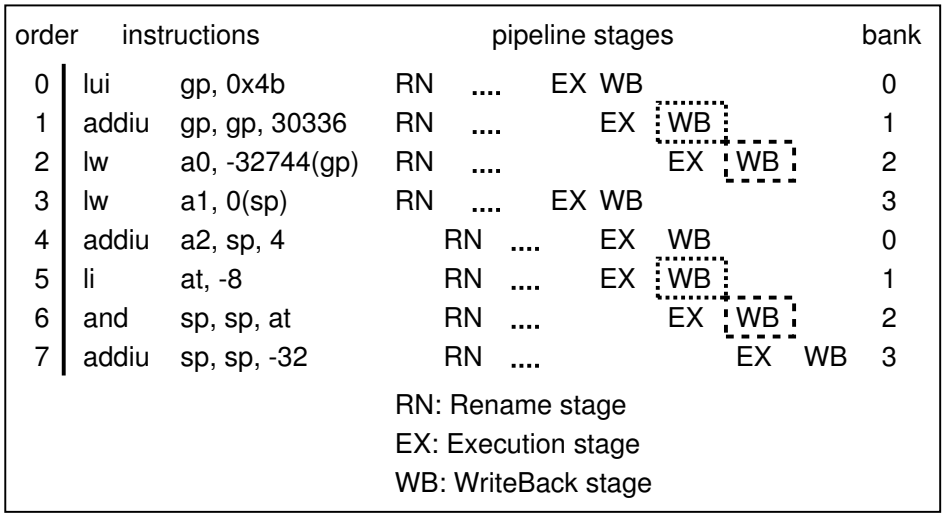


図 3.7: バンクコンフリクトの発生メカニズム

4　関連研究

従来より，レジスタファイルの大容量・多ポート化による回路面積・消費電力・アクセス時間の増加の問題への対処を目的として，様々な研究が行われている．

例えば，マルチバンク化 [3], [4], [5] や，レジスタ・キャッシュ[6], [7], [8]，クラスタ型マイクロアーキテクチャ[9], [10] などがよく知られる手法である．

文献 [3] は，ライトポートのみをマルチバンク化した例であり，バンクコンフリクトの発生を減らす工夫として，リネーミング時に直ちに物理レジスタ番号を割り当てず，依存性タグを割り当てておき，ライトバック時にバンクコンフリクトが発生しないように物理レジスタの割り当てを行う．このため，依存性タグと物理レジスタの対応をとるテーブルが必要となる．文献 [4] の方式ではリード，ライトポートともにマルチバンク化が可能であるが物理レジスタへのアクセスにアクセス・キューが追加要素として必要である．

レジスタ・キャッシュは，ミス時のペナルティが大きいことが問題であったが，文献 [6] では物理レジスタ番号の割り当て順に着目しヒット率を向上させている．文献 [7] では，ミスを仮定したパイプライン構成を取るこ

とで，IPC の低下を抑えつつ，面積，消費電力を削減することに成功している．しかし，アクセス時間の短縮は目的としていないので，レジスタアクセスには複数サイクルかかる．また，文献 [8] ではレジスタ・キャッシュとマルチバンク化を用いた手法を提案しているが，ライト・バッファの面積に占める割合が高すぎることが問題となっている．

本稿で提案するレジスタファイル構成は，1R1W の SRAM の多重化によって作成されるレジスタファイルを対象としている点で先行研究とは異なる．書込予測器は複雑な回路を必要とせず，バンクコンフリクトの発生を抑制することができ，性能低下を抑えることが可能である．マルチバンク化と書込予測を組み合わせることで，レジスタファイルの回路面積を大きく削減することができる．

5　バンク競合低減手法の提案

本章では，まず本研究で提案するレジスタ書込ポート予測を実装するためのマイクロアーキテクチャの拡張について説明した後，提案手法である書込ポート予測機能について説明する．

5.1　マイクロアーキテクチャの拡張

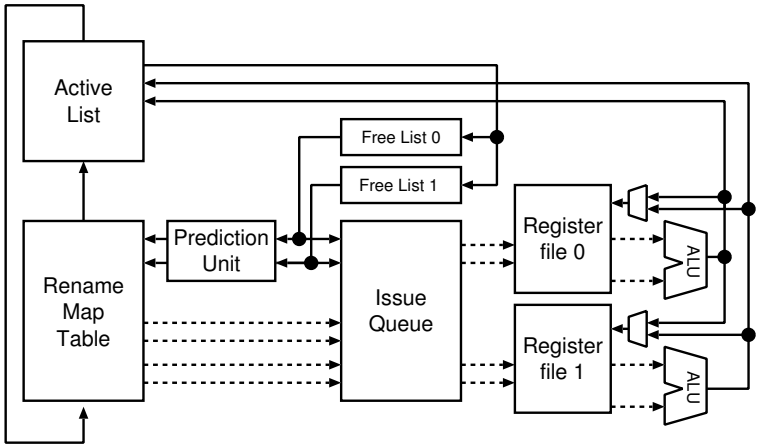


図 5.8: マイクロアーキテクチャの拡張

図 5.8 に提案するレジスタファイル構成を実現するための拡張を行ったマイクロアーキテクチャのブロック図を示す．図 5.8 と図 2.2 を比べると，従来のレジスタファイルを 2 バンクに分割していること，同様にフリーリストを分割していること，フリーリストと RMT の間に PredictionUnit が追加されていることがわかる．本拡張では各物理レジスタ番号は各レ

ジスタバンクに対応するようにバンク分けを行う．例えば，偶数番号はレジスタバンク 0 に，奇数番号はレジスタバンク 1 に対応するようにバンク分けを行う．また，リネーム時に物理レジスタの割り当てを予測を用いて行えるように，予測器を追加する．

従来型のプロセッサの場合，フリーリストのエントリ数はレジスタファイルのエントリ数と RMT のエントリ数の差である．しかし，本拡張において分割されたそれぞれのフリーリストのエントリ数を従来と同じように決めてしまうと，フリーリストのエントリが破壊されてしまう．これを図 5.9 を用いて説明する．簡単のため，レジスタファイルの総エントリ数は 16，バンク数は 2，RMT のエントリ数は 4，偶数番号の物理レジスタ番号を Free List 0 が，奇数番号の物理レジスタ番号を Free List 1 が管理すると仮定する．

従来型のプロセッサのフリーリストのエントリ数の考え方を適用するとフリーリストの総エントリ数は $16 - 4 = 12$ エントリとなり，2 つに分割するのでさらに $12/2 = 6$ となるのでそれぞれのフリーリストのエントリ数は 6 となる．図 5.9 は現在の RMT と AMT の状態を示しており，デスティネーションレジスタ R1 を使用していた命令がコミットされ AMT の該当するエントリを更新し，古い物理レジスタ番号の P1 を解放し，Free

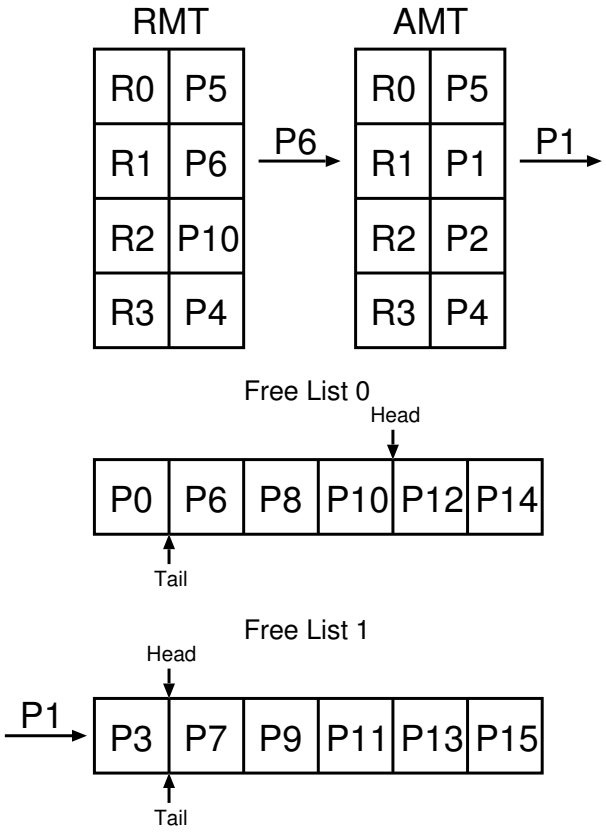


図 5.9: Free List の問題点

List 1 に戻そうとしている場面である．

このときの Free List 1 の Head Pointer と Tail Pointer は同じ地点を指し示しており，この状態で P1 を書き込むと Tail Pointer が Head Pointer を追い越してしまい P7 のエントリが破壊される．これは Free List 0 が管理する物理レジスタ番号によって Free List 1 が管理する物理レジスタ番号が解放されるため，Free List 1 のエントリ数が不足してしまうからである．これを防ぐためにフリーリストのエントリ数をバンク化したレジスタファイルのエントリ数と同じ数にまで拡張する．

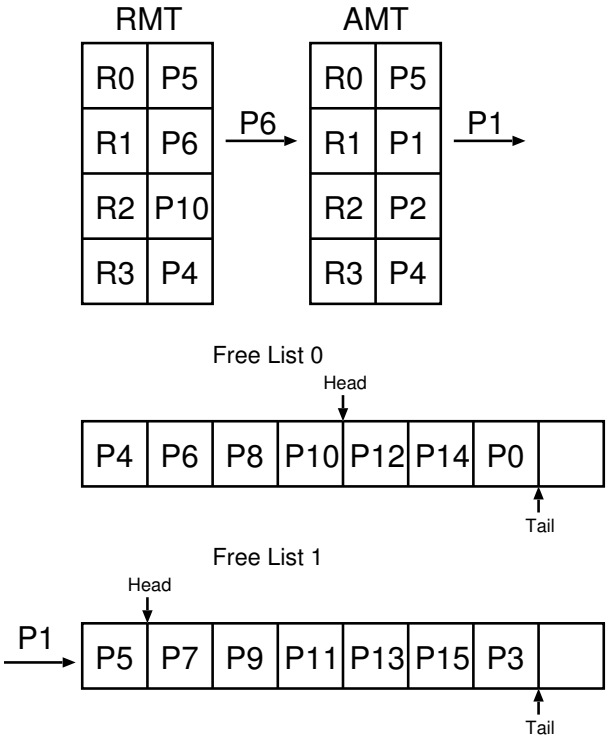


図 5.10: Free List の拡張

図 5.10 はフリーリストのエントリ数をバンク化したレジスタファイルのエントリ数と同じ 8 に拡張したフリーリストを示している．P1 の物理レジスタ番号を書き込むエントリが残されているため他のエントリのデータが破壊されることはなくなる．また，この拡張により Tail Pointer の初期位置が変化する．

従来型のプロセッサの場合，分岐予測ミスやオーバーフローなどの例外が発生した場合，アーキテクチャステートを復帰させるために AMT の内容を RMT にコピーし，同時にフリーリストの Head Pointer が指す位置を Tail Pointer が指す位置に合わせるだけでフリーリストの回復ができる．しかし，本実装では Tail Pointer の初期位置が変化したことにより，Head Pointer が戻るべき位置が特定できずアーキテクチャステートの復帰ができなくなってしまう．図 5.10 の AMT の内容を RMT にコピーした場合，HeadPointer が戻る位置を Tail Pointer が現在指している位置に合わせたとすると，次に読み出すデータが不定値となり，フリーリストの内容が破壊されてしまう．

そこで，AMT を用いないアーキテクチャステートの復帰方法を採用する．図 5.11 のように Active List に命令のデスティネーションレジスタ番号，以前に割り当てられていた物理レジスタ番号，現在割り当てられて

三重大学大学院　工学研究科

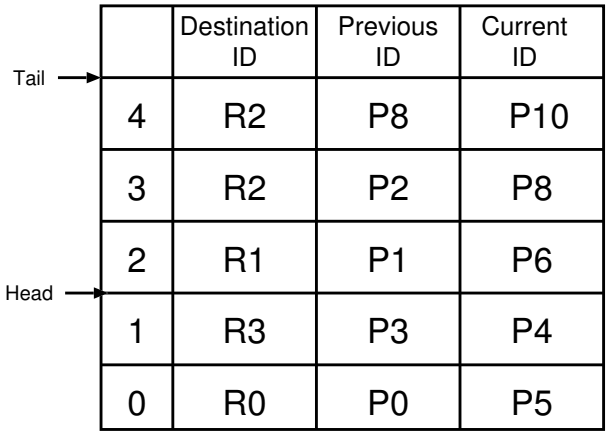


図 5.11: Active List

いる物理レジスタ番号を登録することにより，アーキテクチャステートを復帰させることが出来る．命令をコミットするときは Active List の Head Pointer から順番にデータを取り出し，以前割り当てられていた物理レジスタ番号を解放しフリーリストに戻す．アーキテクチャステートを復帰する場合は Tail Pointer からデータを読み出し，現在割り当てられている物理レジスタ番号をフリーリストに戻し Head Pointer を戻す．加えて，以前割り当てられていた物理レジスタ番号を RMT に登録する．

5.2　レジスタ書込ポート予測の詳細

第 3.2.1 章に示したように，従来のバンクメモリはバンクコンフリクトを起こし性能を低下させている．そこで本稿は，バンクコンフリクトを避けるためにレジスタ書込ポート予測 [11]，[12] を提案する．レジスタ書込予測は，同時にレジスタファイルに書込を起こす可能性が高い命令列を予測し，予測した命令の書込先をそれぞれ異なるバンクに割り当てることでバンクコンフリクトを回避する．ハードウェアの複雑性が高まることを避けるため，予測の対象となる命令を ADD 命令や SUB 命令などのシンプル命令や分岐命令のみをターゲットにする．アルゴリズム 1 と図 5.12 を用いて予測アルゴリズムを説明する．

提案するアルゴリズムは以下の 3 つのステップで構成されている．last_assigned_bank は 0 で初期化する．last_assigned_bank は 0，1，2，3，0 という様に循環リストのように値が遷移する．

Assign physical register

プロセッサはデスティネーションレジスタに last_assigned_bank を用いて物理レジスタ (書込先バンク) を割り当てる．もしある命令 A が先行命令 B に対して依存している場合，命令 A の実行は命令 B の実行後となる．よって，次のサイクルにリネームされる命令列と

バンクコンフリクトを起こす可能性がある．よって，物理レジスタの割り当ては，バンクコンフリクトを避けるために，どの命令が依存によって遅延し，その命令の書込先に割り当てられたバンクはどこなのかという情報を記憶しておかなければならない．そこで，提案するアルゴリズムは遅延する命令から書込先を割り当てることによって複雑性を削減している．

Detect dependency

リネームによって偽の依存関係は取り除かれるが，真の依存関係は取り除くことが出来ない．そこで，真の依存関係を持つ命令の書込先に割り当てたバンクを記憶することで，バンクコンフリクトの回避を狙う．ここでは，ハードウェアの複雑性を削減するために初めに発見した真の依存関係を持つ命令のみに着目することにした．

Propagate

記憶したバンク番号をインクリメントし，保存する．次のサイクルから物理レジスタの割り当ては，保存したバンク番号から行う．

図 5.12 を用いて提案するアルゴリズムを具体的に説明する．ここでも命令の実行サイクル数は全て 1 サイクルとし，同時に 4 命令処理で

Algorithm 1 レジスタ書込予測のアルゴリズム

initialization
$last_assigned_number \leftarrow 0$
1:Assign physical register
for $i = 0$ to $RENAME_WIDTH - 1$ **do**
  $t \leftarrow last_assigned_bank$
  $DestinationRegister[i] \leftarrow pop(FreeList[t++])$
  if $t >= thenumberofbank$ **then**
    $t \leftarrow 0$
  end if
end for
2:Detect dependency
for $i = 0$ to $RENAME_WIDTH - 1$ **do**
  for $j = 0$ to i **do**
    if $detectdependency$ **then**
      $depend_bank_number \leftarrow i$
      $break$
    end if
  end for
end for
3:Propagate
if $detectdependency$ **then**
  $last_assigned_bank \leftarrow depend_assigned_number + 1$
end if

きるプロセッサを使用する．プロセッサは 0〜3 番の命令をリネームし，last_assinged_bank をインクリメントしながら書込先にそれぞれ異なるバンクを割り当てる．各命令の書込先バンクは図 5.12 右のようになる．この処理は Assign physical register に該当する．同時に予測器は命令間の依存関係を解析する．この場合，命令 0(lui) と命令 1(addiu) が gp レジスタについて真の依存関係を持っている．また，命令 1(addiu) と命令 2(lw) も gp レジスタについて真の依存関係を持つ．よって，命令 0(lui) と命令 3(lw) が同時に実行され同時に書込を行う．それぞれの書込先バンクは 0 番と 3 番なので，バンクコンフリクトは発生しない．初めに見つけた真の依存関係を持つ命令 1 に割り当てたバンク番号 (ここでは 1) を depend_bank_number に保存する．この処理は Detect dependency に該当する．最後に depend_bank_number をインクリメントし，last_assigned_number に保存する．この処理は Propagate に該当する．同様に次サイクルは 4〜7 番の命令をリネームし，last_assinged_bank をインクリメントしながら書込先バンクを割り当てる．各命令の書込先バンクは図 5.12 右のようになる．この場合，真の依存によって遅延した命令 1，4，5 が同時に実行され書込を行う．各命令の書込先バンクは，それぞれ 1，2，3 であるため，バンクコンフリクトは起こらない．次のサイクルは命令 2，6 が同時に書

込を行う．各命令の書込先バンクは，それぞれ，2，0 であり，今回もバンクコンフリクトは起こらない．最後に命令 7 が書込を行う．よって，提案するアルゴリズムを利用し，同時にレジスタファイルに書込を行う可能性が高い命令を予測することでバンクコンフリクトを回避することが出来る．

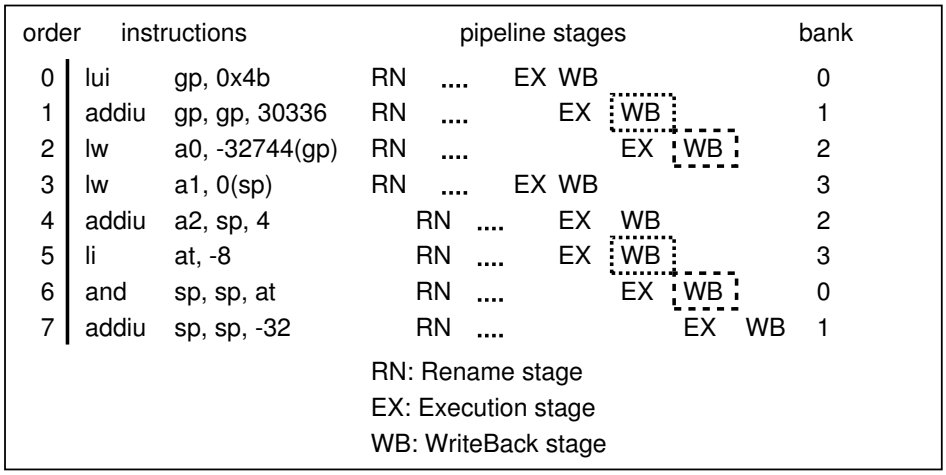


図 5.12: 命令列の例と提案アルゴリズムの動作

6　評価

6.1　評価環境

本章では，提案するレジスタ書込ポート予測の有効性を示すため，FabScalar[13]上に予測機構を実装し，性能評価，アクセスタイム評価，面積評価及び電力評価を行う．FabScalar とは任意のスーパースカラコアを生成することが出来るツールセットである．評価には，SPEC2000INT より 164.gzip，176.gcc，181.mcf，197.parser，254.gap，256.bzip，300.twolf を使用し，初めの 20 億命令をスキップしその後の 1 億命令を評価した．表 6.1 にプロセッサの構成を，表 6.2 に使用した EDA/CAD ツールを示す．

表 6.1: プロセッサの構成

Data path width	32 bit
Fetch, Dispatch, Issue, Commit width	4
Branch prediction	16K entry, bimodal
BTB	1024sets
Issue Queue size	32
Register File	96 entry
L1 I-Cache	32KB, 16B/line, 4way 1cycle latency
L1 D-Cache	32KB, 16B/line, 4way 2cycle latency

表 6.2: 使用した EDA/CAD ツール

Functional verifiation	Cadence NC-Verilog 12.20
Synthesis	Synopsys Design Compiler 2013.03-SP2
Power estimation	Synopsys PrimeTime PX D-2010.06
Technology	Rohm CMOS 0.18 μ m
Standardcell library	Kyoto University standard cell library

6.2 性能評価

図 6.13 は，それぞれマルチポートレジスタファイル (Multi-port)，予測機構なしの 4 バンクレジスタファイル (4 bank without prediction)，予測機構ありの 4 バンクレジスタファイル (4 bank with prediction) を用いた性能評価を示している．各プログラムの実行時間は，マルチポートレジスタファイルを使用したときの実行時間を 1 として正規化されている．提案するレジスタ書込ポート予測を用いたとき，164.gzip，176.gcc，181.mcf，197.parser，300.twolf はバンクコンフリクトの発生回数が減ったため，予測を用いない 4 バンクレジスタファイルと比較すると，実行時間を削減することが出来ている．しかし，254.gap，256.bzip2 はバンクコンフリクトの発生回数が増加しているため実行時間が伸びてしまっている．これはプログラムの命令間に強い依存関係があり，予測が機能せずバンクコンフリクトの発生回数をむしろ増やしてしまったためと考察できる．提

案する予測機構を用いた場合と用いない場合を比較すると，予測を用いた方が，実行時間を最大 3.9%，平均 0.8%削減できることがわかった．

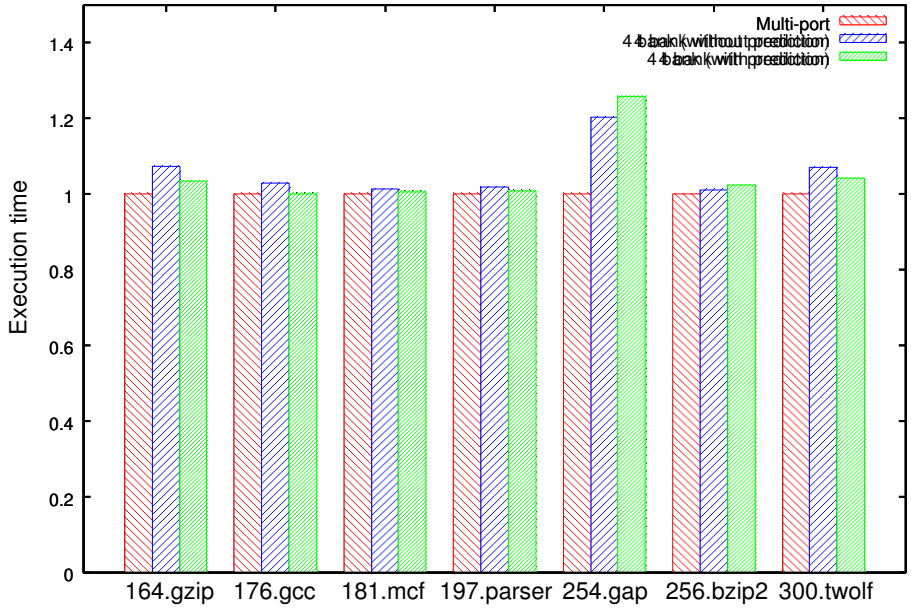


図 6.13: 性能評価

6.3 アクセスタイム評価

表 6.3 はアクセスタイム評価を示している．評価には表 6.2 に示したように，Synopsys Design Compiler を用いて静的なタイミング解析を行った．評価によると，4 バンクレジスタファイルは SRAM 多重化を用いたマルチポートレジスタファイルよりもアクセスタイムを 8.3%削減できることがわかった．4 バンクレジスタファイルを用いることで SRAM のライトポート多重化をキャンセルし，配線が簡単になることでアクセスタ

イムが短縮したと考察できる．

表 6.3: アクセスタイム評価

マルチポート	2.46ns
4 バンク	2.25ns

6.4 面積評価

表 6.4 は面積評価を示している．表 6.2 に示したテクノロジーを用いて評価を行った．評価によると，4 バンクレジスタファイルは SRAM 多重化を用いたマルチポートレジスタファイルよりも面積を 77.6%削減できることがわかった．4 バンクレジスタファイルはデータ領域を 4 つに分割するため，原理的に面積を 4 分の 1 にすることができる．また，ライトポートの多重化には第 3.1.2 項で示したように追加の回路が必要になるが，これらの回路もライトポートの多重化をキャンセルすることで削減できる．

表 6.4: 面積評価

マルチポート	2,764,611 μm^2
4 バンク	619,261 μm^2

6.5　電力評価

提案手法の消費電力を評価するために，表 6.2 に示した Synopsys Prime-Time PX を用いて電力予測を行った．

表 6.5，表 6.6，表 6.7 はそれぞれ，RAM 部を除いたプロセッサ全体の NAND ゲート換算のゲート数，RAM 部を除いたプロセッサ全体の消費電力，レジスタファイル以外の RAM 部を除いたプロセッサ全体の消費電力を示している．評価結果によると，提案する予測機構を実装することでゲート数は 0.83%，消費電力は 1.94% 増加した．これは予測に用いる情報を保存するバッファ及び，物理レジスタの割り当てを行う回路に変化がみられるためだと考察できる．従来の単純な物理レジスタの割り当ては第 3.2.1 項に示したように，リネーム時にそれぞれ異なる書込先を割り当てるが，書込先は一意に決定することが出来る．これを図 3.7 を用いて説明する．命令順が 0(lui) と 4(addiu) の命令の書込先は共に 0 番のバンクであり，命令順 1(addiu) と 5(li) は 1 番，2(lw) と 6(and) は 2 番，3(lw) と 7(addiu) は 3 番である．つまり命令順の番号を 4 で割った余りの番号に一意に決定する．しかし，書込予測を用いることで，図 5.12 のように書込先が変化する．よって，書込先の割り当てを行う配線がクロスバとなるため消費電力が増加する．RAM 部のうちレジスタファイルのみを加

えたプロセッサの消費電力評価は，通常のプロセッサと比べ 11.11%削減することができた．

表 6.5: RAM 部を除いたプロセッサ全体の回路規模

通常のプロセッサ	152,236 μm^2
提案手法を実装したプロセッサ	153,505 μm^2

表 6.6: RAM 部を除いたプロセッサ全体の消費電力

通常のプロセッサ	8.76 mW
提案手法を実装したプロセッサ	8.93 mW

表 6.7: レジスタファイル以外の RAM 部を除いたプロセッサ全体の消費電力

通常のプロセッサ	18.73 mW
提案手法を実装したプロセッサ	16.65 mW

7　結論

本稿では，レジスタファイルの小面積化を目的として，レジスタ書込ポート予測を用いたバンクレジスタファイルを提案した．レジスタ書込ポート予測は同時にレジスタファイルに書込を行う可能性の高い命令列を検出し，それぞれの命令の書込先を異なるバンクに振り分けることでバンクコンフリクトの発生を防ぎ，性能低下を抑える働きがある．評価結果によると，書込予測を使用しないバンクレジスタファイルよりも予測を用いた方がベンチマークプログラムの実行時間を最大 3.9%，平均 0.8%減少させることに成功した．また，予測機構実装によりプロセッサの消費電力を 11.11%削減し，レジスタファイルの面積を 77.6%，アクセスタイムを 8.3%削減することができた．今後の展望として，他手法との比較や書込予測のアルゴリズムをさらに発展させ，さらに性能低下を抑える方法を検討していく予定である．

謝辞

本研究を行うにあたり，多数のご指導を頂きました近藤利夫教授，深澤研究員，並びに佐々木敬泰助教に深く感謝いたします．また，計算機アーキテクチャ研究室院生・学生のメンバーには常に刺激的な議論を頂き，精神的にも支えられました．また，本研究は日本学術振興会の科学研究費補助金，Synopsys 社 CAD ツールによる東京大学 VDEC，Rohm 社 VDEC，凸版印刷社の支援により実施されたことを並びに感謝します．

参考文献

[1] Brandon H. Dwiel, Niket Kumar Choudhary and Eric Rotenberg:FPGA modeling of diverse superscalar processors, ISPASS 2012, pp. 188-199.2012.

[2] Charles Eric LaForest, Ming G. Liu, Emma Rae Rapati and J. Gregory Steffan:Multi-Ported Memories for FPGAs via XOR, FPGA ' 12, pp. 209-218, 2012.

[3] Il Park, Michael D. Powell and T. N. Vijaykumar:Reducing Register Ports for Higher Speed and Lower Energy, MICRO-35. 2002.

[4] Hironaka, T. et al.:Superscalar processor with multi-bank register file, Innovative Architecture for Future Generation High Performance processors and Systems, 2005.

[5] J. L. Cruz, et al.:Multiple-Banked Register File Archi-tectures, In Proc. the 30th ISCA, pp.62-71, June 2003.

[6] R. Kobayashi, D. Horibe and T. Shimada:High Accuracy of Register Cache with Ordering of Physical Register Numver, HOKKE-2006,

2006.

[7] Shioya, R., Horio, K., Goshima, M. and Sakai, S.:Register Cache System Not for Latency Reducing Purpose, 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 301-312, 2010.

[8] J. Yamada, N. Kurata, R. Shioya, M. Goshima and S. Sakai:Multi-bank Register File in the Register Cache System, 2014-ARC-212, pp. 1-14, 2014.

[9] G.S. Shohi, et al.:Multi-scalar processors, In Proc. the 22th ISCA, 1995.

[10] R.E. Kessler.:The Alpha 21264 Microprocessor, IEEE Micro, Vol.19, No.2, pp.24-36, Apr. 1999.

[11] 川島弘晃、他:マルチバンク化と書込予測を用いた小面積レジスタファイルの提案. 情報処理学会研究報告, Aug.2015

[12] Hiroaki Kawashima, et al.:Register Port Prediction for a Banked Register File, The Third International Symposium on Computing and Networking. Dec.2015.

[13] Niket Kumar Choudhary, et al.:FabScalar: Composing synthesizable RTL designs arbitrary cores within a canonical superscalar template, ISCA 2011, pp. 11-22, 2011.