特別研究報告書

スーパースカラにおける
高速な動的命令スケジューリング方式の
IPC の評価

指導教官　　富田 眞治 教授

京都大学工学部情報学科

西野 賢悟

平成 13 年 2 月 13 日

スーパースカラにおける高速な動的命令スケジューリング方式のIPCの評価

西野 賢悟

内容梗概

プロセッサの性能は、クロック速度とIPC(instructions per cycle)の積によって算出される。スーパースカラのIPCを向上させる最も直接的な方法は、命令発行幅とウィンドウサイズを増やすことである。初期のスーパースカラにおいては、トランジスタ数が許す範囲で命令発行幅とウィンドウサイズを増やすことにより、大幅にIPCを向上させてきた。

しかし現在では、クロック速度が命令発行幅とウィンドウサイズを制限する主因となりつつある。命令発行幅とウィンドウサイズを増やしても単純にIPCが増加するわけではなく、徒に増加させればかえって全体の性能を悪化させることになる。

スーパースカラは、動的命令スケジューリングのため、命令の実行に必要なデータの有効性を追跡する*wakeup*と呼ぶロジックを持つ。従来の*wakeup*は、データに割り当てられたタグによる連想処理に基づくもので、RAMを読み出した結果でCAMをアクセスするという構造を持ち、LSIの微細化、パイプラインの深化に伴っていっそうクリティカルになっていくと予測されている。また、動的命令スケジューリングには他にも*rename*と*select*というロジックがある。このうち*rename*は他のロジックとパイプライン化することが可能であるが、*wakeup*と*select*はパイプライン化が不可能である。このそれぞれに1サイクルをかけると、IPCは30%も 低下し、クロック速度向上に見合わない可能性が高い。

本稿ではまず*wakeup*を高速化する方式について述べる。この高速な方式は、タグに基づく連想処理ではなく、命令間の依存関係を直接的に表現するテーブルを用いるもので、単にRAMを読み出すことで*wakeup*を実現することができる。そして、実際に実装する方法と、その更新処理について述べる。

更に本稿では、このロジックの遅延を、テーブルを縮小することによりIPCに対するペナルティに転化する手法を2通り示す。縮小したテーブル内で*wakeup*を行える場合は普通にそのまま行えばよいが、そうでない場合は他の物に頼るのである。1つ目の方法は、delay方式といい、もう1つテーブルを用意し、そ

ちらでは 1 サイクル余分にかけて *wakeup* を行うというものである。2 つ目の方法は、stall 方式といい、*wakeup* を行えない命令をキューに入れずにデコードの段階で止めておくというものである。

だが、他にもペナルティは存在する。命令ウィンドウを複数の命令キューによって実装している場合は、命令キュー間で消費の歩調を揃える必要がある。そのため、無駄に消費されるエントリが発生する。また、このテーブルではエントリのリセットを行う必要があるため、解放直後のサイクルには当該エントリは使用できないことがあり、エントリの使用効率は悪化する。

実在する.18μm CMOS プロセスのデザイン・ルールに基づいてこれらのロジックを設計し、回路の面積を求め、Hspice によって遅延を測定した。また、シミュレータにこの機構を実装し、SPEC95 の整数系ベンチマークプログラムを実行させて、IPC のペナルティを測定した。その結果、delay では命令キューサイズの 4 分の 1、また stall では 2 分の 1 のサイズのテーブルを実装すれば、IPC のペナルティは 3% 以下となることが分かった。その他のペナルティ要因はほとんど無視できるほどである。また、総合すると、3%以下の IPC ペナルティを代償に、2GHz を越える最高動作周波数を達成できることが分かった。

IPC evaluation of a high-speed dynamic instruction scheduling method for superscalar

Kengo NISHINO

Abstract

The performance of the processor is calculated by the product of clock speed and IPC. The most immediate method that improves IPC of a superscalar is to increase issue width and window size. As for early superscalar, the number of the transistors they improved IPC drastically by increasing issue width and window size in the range that the number of the transistors is forgiven.

But now, clock speed becomes the primary cause that limits issue width and window size. Even if they are increased, IPC doesn't always increase simply. it makes the whole performance worse all the more if they are increased in vain.

A superscalar has *wakeup* logic, which manages availability of the data for dynamic instruction scheduling. The usual *wakeup* logic is based on association of the tag allocated to the data. The delay time of *wakeup* consists of read delay time of a RAM and matches access delay time of a CAM. Since the delays of these memories are dominated by the wire delay, it will be more critical with smaller feature sizes and deeper pipelines. Besides, there are both *rename* and *select* logics in dynamic instruction scheduling. *rename* logic can be pipelined with other logics, but *wakeup* logic and *select* logic cannot. If we spend a cycle each of them, IPC drops 30%. It seems to be worthless clock speed improvement.

This paper describes a high-speed dynamic instruction scheduling scheme. This high-speed scheme is not based on association of the tags, but a matrix, which directly represents the dependence among instructions. The scheme realizes *wakeup* by just reading a small RAM. Then, this paper describes how to implement this table, and how its renewal is done.

In addition, this paper also describes two schemes, which changes the delay of the logic into IPC penalty by shrinking the table. One of them is named "delay". At this scheme we prepare one more full-size table. When use this table, spend one more cycle to *wakeup* . Another is named "stall". Instructions that cannot *wakeup* stall in there decode term.

But there are other penalties. If instruction window is mounted by more

than one instruction queue, make the step of the consumption equal among the queues.

Therefore, entries consumed uselessly. Moreover, because this table needs reset before its entry used, The cycle when the entry released, sometimes the entry cannot be used. It decreases efficiency of using entries.

We actually designed the logic guided by a design rule of a real .18μm CMOS process, measured the areas, and calculated the delays by Hspcie. And we also implemented simulator this scheme, and evaluated the IPC penalty of the scheme by running SPEC95 CINT benchmark programs. The evaluation result shows that "delay" scheme needs only a quarter size table to achieve IPC penalty less than 3%. "Stall" scheme needs half size table. Other penalties are very small. Synthetic evaluation result shows that this scheme achieves over 2GHz clock speed with the IPC penalty less than 3%.

スーパースカラにおける高速な動的命令スケジューリング方式の IPC の評価

目次

第 1 章　はじめに

プロセッサの性能は、クロック速度と IPC (instructions per cycle) との積によって算出することができる。スーパースカラの IPC を向上させる最も直接的な方法は、命令発行幅 (IW:Issue Width) とウィンドウ・サイズ (WS) を増やすことである. 実際 初期のスーパースカラは、トランジスタ数が許す範囲で IW, WS を増やすことにより、大幅に IPC を向上させてきた.

しかし現在では、LSI の微細化にともなって、トランジスタ数ではなく、クロック速度が IW, WS を制限する主因となりつつある。IW, WS を増やしても単純に IPC が向上する訳ではないので、徒に増加させれば、かえって全体の性能を悪化させることになる。

スーパースカラの構成要素のうち、$wakeup$ と呼ぶロジックが、将来クロック速度を制限するものの 1 つになると予測されている [4]。$wakeup$ は、動的命令スケジューリングのために、命令の発行に必要なデータの有効性を追跡するロジックである。

従来の $wakeup$ は、各命令が使用するデータに割り当てられたタグによる連想処理に基づくもので、RAM を読み出した結果で CAM をアクセスするという構造を持つ。これらのメモリは、配線遅延に支配されるため、LSI の微細化の恩恵を受けにくい。また $wakeup$ は、他の多くの構成要素とは異なり、複数のパイプライン・ステージに分割することができない。以上の理由により $wakeup$ は、LSI の微細化、パイプラインの深化にともなっていっそうクリティカルになっていくと予測されるのである。

このような背景から我々は、$wakeup$ を高速化する全く新しい動的命令スケジューリング方式を提案する。本方式では、タグによる連想処理ではなく、命令間の依存関係を直接的に表現する行列を用いて $wakeup$ を実現する。その結果、4b×4word、1-read IW-write という、小型の RAM を読み出す程度の遅延で $wakeup$ を実行することができる。本稿では、この方式を更に高速化する手法と、定量評価の結果を示す。

以下、まず 2 章では従来の動的命令スケジューリング方式の一般的な構成法を紹介し、3 章で提案する方式について詳しく述べる。そしてそして 4 章で IPC 測定のためにシミュレータに対して実装したことを述べ、5 章で、実際に行なった測定の結果を評価する。最後に 6 章でまとめをする。

第 2 章　従来の動的命令スケジューリング方式

スーパースカラを構成するの基本構造のうち，演算器それ自体以外のほとんど全ての遅延は IW, WS の増加関数で与えられる。そのような構造には，キャッシュ，命令フェッチ・ロジック，レジスタ・ファイル、オペランド・バイパス、そして、本稿の主眼である動的命令スケジューリングを行うロジックなどがある。

ただしそれらの遅延の増大が、直接システムのクロック速度の低下につながるわけではない。いくつかの処理に対しては、パイプライニングやクラスタリングなどの技術によって、1 サイクルに終えなければならない処理の遅延を大幅に短縮できるからである。

例えば、命令フェッチやレジスタ・リネーミングなど、命令パイプラインの実行ステージより前にある処理の遅延は、パイプライン化によって分岐予測ミス・ペナルティに転化することができる。最近では、AMD Athlon や Intel Pentium III、4 などのように、キャッシュやレジスタへのアクセスに対してもパイプライン化が施されるようになってきている。

また、DEC 21264 に採用されているように、演算器やレジスタ・ファイルをクラスタリングすることによって、レジスタ・ファイルのポート数の削減、オペランド・バイパスの配線長の短縮が可能である [3]。

これらの技術は、ロジックの遅延を、一部の命令の実行レイテンシや、何らかのペナルティに転化するものである。したがって、これらの技術が有効であるためには、クロック速度の向上に対して IPC の悪化の度合いが十分に小さい必要がある。

しかし動的命令スケジューリングを行うロジックに対しては、このような技術は効果的ではない。本章では、その理由について詳しく述べる。以下まず 2.1 節においてスーパースカラの動的命令スケジューリングの原理についてまとめ、2.2 節でスケジューリングの処理と命令パイプラインの関係について説明する。

2.1　従来の動的命令スケジューリング方式

Out-of-order スーパースカラは、論理的なレジスタとは別に、各命令の実行結果を一時的に保存するバッファを用いる。このバッファの構成方式には、リオーダ・バッファを用いる方式と、物理レジスタを用いる方式がある。本稿ではこの違いは重要ではないので、これらを単にバッファと呼ぶことにする。

スーパースカラにおける動的命令スケジューリングは、このバッファのエントリを用いて、局所的にデータ駆動型の計算を行うこととみなすことができる。命令ウィンドウ内で、命令 I_p が生産するデータを命令 I_c が消費する場合を考えよう。バッファのエントリを介して I_p から I_c にデータが渡されることに着目すると、スケジューリングの処理は以下のように説明できる：

(1) *rename*　命令がフェッチされると、論理レジスタ番号から**タグ**への変換が行われる。

I_p には、バッファの 1 エントリが割り当てられ、エントリはデータが『ない』状態に初期化される。このエントリの ID がタグである。I_p に割り当てられたタグを特に tagD ということにする。

I_c は、左/右のソースの論理レジスタ番号から、依存する I_p に割り当てられた tagD を得る。これを tagL/R ということにする。I_c は、tagL/R で示されるエントリにデータが書き込まれるのを待つ。

(2) *wakeup*　I_p の実行にともなって、I_c が実行可能になることを検出する。

I_p が実行されると、その結果は tagD で示されるエントリに書き込まれ、エントリはデータが『ある』状態に遷移する。

I_c は、tagL/R で示されるエントリにデータが『ある』のを見て、発行可能になる。

(3) *select*　発行可能な命令から、実際に発行するものを選択し、発行する。

2.2　命令スケジューリングのパイプライン化

次に，*rename*，*wakeup*，*select* の各処理をパイプライン化することを考えよう。命令パイプライン中のステージの違いから，*rename* と (*wakeup*+*select*) とに分けて考える必要がある。

rename は，必要ならば，パイプライン化することでクリティカル・パスから外すことができる。実際現存するスーパースカラでは，*rename* の遅延のため、デコード・ステージに複数サイクルを充てることが普通である。ただしもちろん、その分だけ分岐予測ミス・ペナルティが増加することになる。

wakeup と *select* は、*rename* とは異なり、パイプライン化することができない。図1 に、*wakeup* と *select* のそれぞれに 1 サイクルかけた場合の命令パイプラインの様子を示す。この場合、`add` の結果を消費する `sub` は、先行する命令に引き続くサイクルに実行することができない。このことは、レイテンシが

1 サイクルである演算器からはオペランド・バイパスを行わないことと等価である。詳細は 5 章で述べるが、それによる IPC の悪化は 30%にもなり、クロック速度の向上に見合わない可能性が高い。

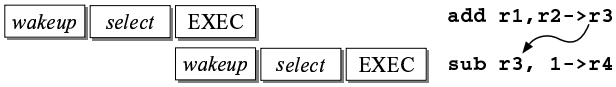

図 1: *wakeup* と *select* のパイプライン化

2.3 Superscalar の *wakeup*

図 2 に、従来方式の命令ウィンドウのブロック図を示す [4]。*wakeup* は、実行される I_p の tagD を上部の RAM から読み出し、それをキーとして下部の CAM にアクセスするという処理によって実現される。

CAM の入力側は tagD をキーとする IW 本の比較入力ポートである。入力された tagD と一致する tagL/R を検出し、tagL/R で示されるバッファのエントリ

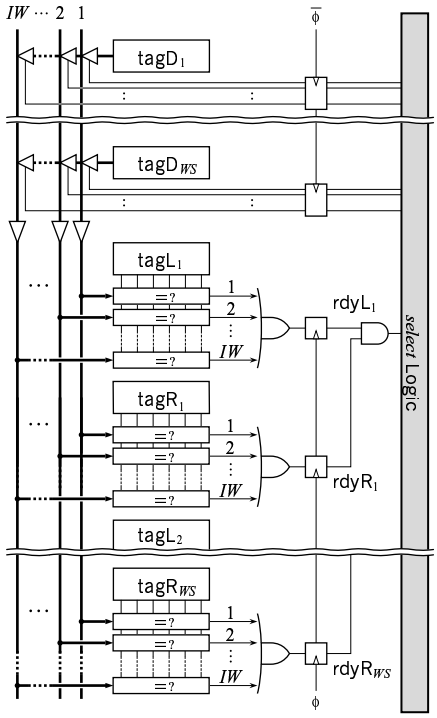

図 2: 従来方式の命令ウィンドウ

Instruction window logic of a superscalar

にデータが『ある』ことを示すフラグ rdyL/R をセットする。出力側では、rdyL/R を記憶する $2 \cdot WS$ 個のセルの出力線が直接的に $select$ に接続されている。

命令ウィンドウは、実際には、演算器（のクラス）ごとに設けられたリザベーション・ステーション、あるいは、命令キューによって実現されることが多い。例えば MIPS R10000 では、ロード/ストア、整数演算、浮動小数点演算のそれぞれに 16 エントリの命令キューを用意している [5]。しかし、複数のキューによって実現される場合にも、それらの基本的な構成は図2 に示したものと変わらない。ただし、命令キューの命令発行幅を $IW_q(< IW)$、サイズを $WS_q(< WS)$ とすると、各キューにおける RAM、CAM のワード数は WS から WS_q に、また、RAM の読み出しポート数は IW から IW_q に、それぞれ縮小することができる。しかし CAM の比較入力ポート数は、IW のまま縮小することができない。

前述したように、$wakeup$ と $select$ と合わせて 1 サイクルで実行する必要がある。これらのうち、$select$ の遅延は専らゲート遅延からなるため、LSI の微細化に伴って順調に短縮されていくと予測される。一方、$wakeup$ の遅延は RAM と CAM のワード線、ビット線などの配線遅延からなるため、LSI の微細化の恩恵を受けにくい。以上の理由により $wakeup$ は、LSI の微細化、パイプラインの深化にともなっていっそうクリティカルになっていくと予測されるのである [4]。

第 3 章　高速な動的命令スケジューリング方式

提案するスケジューリング方式は、従来のタグによる連想処理に基づくものとは根本的に異なり、各命令間のデータ依存関係を直接的に表すデータ構造を用いてスケジューリングを行う。本章では，提案方式について詳しく述べる。

3.1　DMT

提案方式では，命令ウィンドウ中の各命令間のデータ依存関係を直接的に表す**依存行列テーブル** (dependence matrix table:**DMT**) が，スケジューリングにおける中心的な役割を果たす。

3.1.1 DMT の概要

図 3 に，DMT の概念図を示す。DMT は、基本的には、rdyL/R 用に各 1 つづつの WS 行 WS 列の RAM である（ただし、対角要素は使用しない)。それぞれの RAM の各行は I_p に、各列は I_c に対応する。各要素は、対応する I_p と I_c の間の依存関係を表す。すなわち、p 行 c 列の要素は、命令ウィンドウ内のエントリ ID=p の I_p の実行結果を ID=c の I_c が消費するなら “1”、そうでなければ “0” とする。図 3 の例は、連続する 4 つの命令が ID=1、2、3、4 のエントリに順に格納された場合を表している。図の例では、ID=1 の命令が生産する r1 を ID=2、3 の命令がそれぞれの左オペランドとして消費している。したがって DMT では、1 行目の 2、3 列目が “1” となる。その他の要素も同様に求められる。

DMT の更新は、従来のスーパースカラの $rename$ ステージにおいて実行しておくことができる。それについては次節で詳しく述べることとし、本節では提案手法のポイントとなる $wakeup$ の処理について述べる。

$wakeup$ ステージにおいては、実行される最大 IW 個の I_p に対応する IW 行の OR を求めれば、セットすべき rdyL/R を表す行ベクトルを求めることができる。例えば、図 3 に示した状態でエントリ ID=1 の命令が発行された場合を考えよう。この命令の実行結果は、ID=2、3 の命令が左オペランドとして消費する。自明ではあるが、DMT の第 1 行は ID=1 の命令が実行される時にセットすべき rdyL/R を表している。実際に ID=1 の命令が実行されると、ID=2、3 の

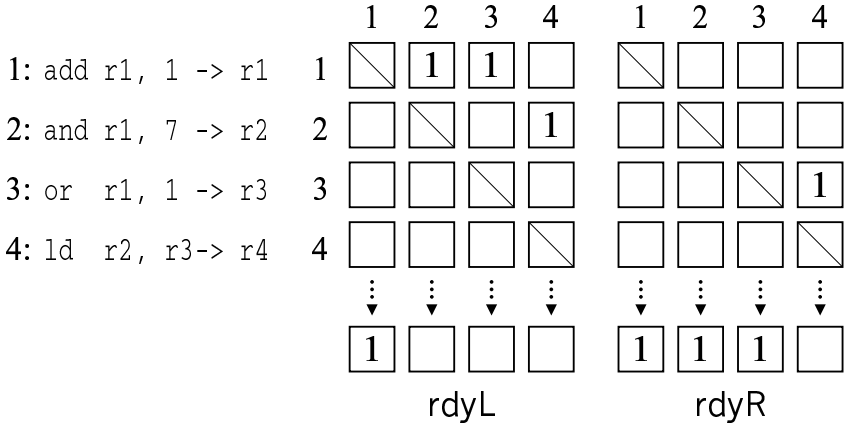

図 3: 依存行列テーブル (dependence matrix table:DMT)

Dependence matrix table:DMT

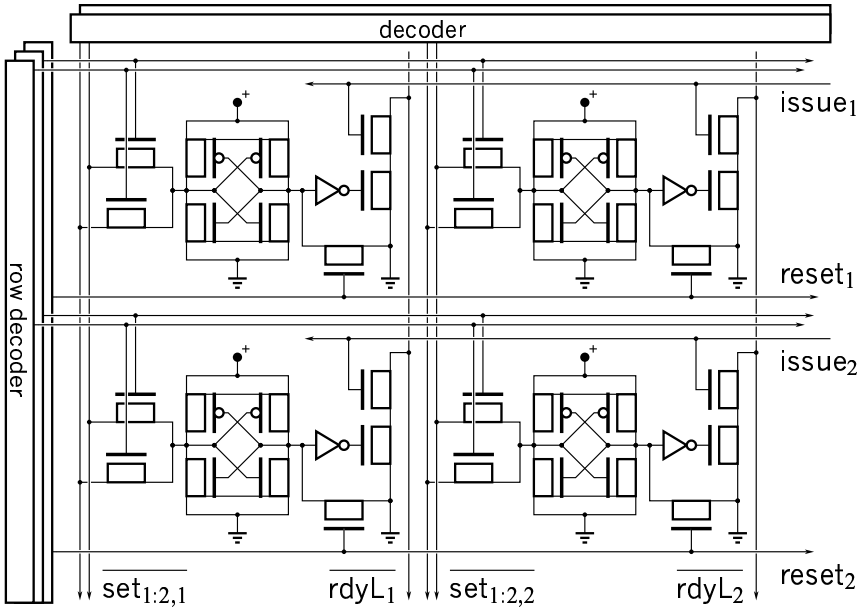

図 4: 依存行列テーブルのロジック（rdyL 用）

rdyL がセットされ、次のサイクルにはその 2 つの命令が実行可能になる。実際にその ID=2、3 の命令が選択され同時に実行される場合には、第 2 行と第 3 行の OR を求めればよい。結果、ID=4 の命令の rdyL/R がそれぞれによってセットされる。ただし実際には、OR を求めると言っても、各列で “1” である要素ははたかだか 1 つである；なぜなら、1 つのソース・オペランドを生産する I_p はただ 1 つだからである。

命令ウィンドウが q本の命令キューによって実装される場合には、生産側と消費側に対応して、$q \times q$ 個の DMT を用意すればよい。それぞれの DMT は、WS b×WS word から、WS_q b×WS_q word に縮小される。

3.1.2 DMT の実装

DMT を実装するにあたっては、単に 1-read の RAM を用いればよい。複数行の OR を求めるための特別なロジックは必要ない。

図 4 に、DMT の回路図を示す。同図には、左上の 2 行 2 列分のセルが示してある。各セルの中央にある 4T セルの左側は書き込みポートであり、次節で詳しく述べる。*wakeup* に関連するのは、4T セルの右側にある読み出しポートである。図からも明らかなように、この読み出しポート部は、single-bitline の 1-read の RAM と構造上は全く変わらない。

ただし、通常の RAM では同時にはたかだか 1 つのワード線しかアサートさ

れないのに対して、DMT では毎サイクル実行される I_p に対応する（最大）IW 本のワード線 issue が同時にアサートされる点が異なる。各ビット線 $\overline{\text{rdyL}}$ には、アサートされた issue に対応する（最大）IW 個のセルが接続され、いずれかのセルの出力が low であれば pull-down される。すなわち、単に複数のワード線を同時にアサートすることによって、対応する行の OR を読み出すことができる。

3.2 DMT の更新

次に，DMT の更新処理について述べよう。DMT の更新は、ちょうど *rename* と同時に、*rename* と同形のロジックによって実現される。そこで、以下まず 3.2.1 節で *rename* についてまとめ、3.2.2 節で DMT の更新について述べる。

3.2.1 *rename*

rename では、前述したように、まず I_p に tagD を割り当て、次に I_c に対し tagL/R を求める。

tagL/R を求める上では、論理レジスタ番号からタグへの写像を記録するレジスタ・マップ・テーブル (**RMT**) が中心的な役割を果たす。RMT は、論理レジスタ番号をアドレス、タグを内容とする、$2 \cdot IW$-read IW-write の RAM である。

I_p は、tagD を割り当てられた後、デスティネーションの論理レジスタ番号をアドレスとして、今割り当てられた tagD を RMT に書き込む。一方 I_c は、基本的には、左/右ソースの論理レジスタ番号をアドレスとして RMT を読み出すことによって tagL/R を得る。

ただし、同時にデコードされる（最大 IW 個の）命令間に依存がある場合には、RMT からは『古い』tagL/R が得られるので、調整が必要である。そのため、比較器のアレイによって論理レジスタ番号の一致比較を行い、同時にデコードされる命令間の依存を検出する。依存が検出された場合には、RMT から読み出される『古い』tagL/R の代わりに、RMT に書き込まれようとしている『新しい』tagD を用いればよい。

3.2.2 DMT の更新

では、DMT の更新処理について述べよう。今デコードされている I_c が ID=c のエントリに格納されるとしよう。DMT の更新処理は、以下の 2 つのフェーズからなる：すなわち、1. I_c の依存元の I_p が格納されているエントリの ID=p

を求める、2. DMT の p 行 c 列をセットする の 2 つである。それぞれのフェーズの処理は、具体的には以下のとおりである：

1. I_c に依存する I_p のエントリ ID は、上述した $rename$ とほぼ同じ方法によって求めることができる。上記の説明において、『タグ』を『I_p のエントリ ID』と読み換えればよい。RMT の位置には、論理アドレス番号をアドレスとし、I_p のエントリ ID を内容とする、$2 \cdot IW$-read IW-write の RAM が用いられる。この RAM を**生産者テーブル** (producer table:**PT**) と呼ぶことにする。
同時にデコードされる命令間の依存を検出するための比較器のアレイは、$rename$ と共用することができる。したがって、提案方式のために別途必要となるのは、主に、PT である。
2. DMT に p をアドレスとして与え、c をデコードしたものをセットすればよい。この際には、上書きするのではなく、セットする点に注意する必要がある。図 4 に示した DMT の回路図を再び参照されたい。同図中、4T セルの左側がセットを行うためのポートである。同図は、$IW = 2$ の場合を示しており、書き込み用の nMOS トランジスタが各セルに 2 つづつ用意されている。通常の RAM セルでは 2 本の相補的なビット線によって書き込みを行うが、DMT のセルではセット用のビット線は原理的に 1 本しか用いない。セットする時には、c をデコードし、更に反転したものをビット線に入力する。各セルにおいて、接続されたビット線が low であればセットが行われるが、high であってもリセットされることはない。その代わり、エントリ（行）は使用に先だってリセットする必要がある。リセット用ポートは、同図では、4T セルの右下に示してある。

使用に先だってエントリのリセットを行う必要があるため、パイプラインの構成によっては、エントリが解放された直後のサイクルには当該エントリを使用することはできないことがある。そのためエントリの使用効率は悪化するが、第 5 章で示すように、その影響はわずかである。

3.2.3 更新処理の遅延

では次に、更新処理の遅延について考察しよう。DMT の更新は $rename$ と同時に行われるから、$rename$ と比較して、その遅延がクロック速度に影響する可能性があるのは、1. PT の遅延 と、2. c のデコード である。以下、それぞれについて述べる：

1. 物理レジスタの本数は WS の倍程度とすることが普通であるから、PT の内容であるエントリ ID は、RMT の内容であるタグより 1b 程度短い。したがっ

て、PT の遅延は RMT のそれより小さく、そちらがクリティカルになることはない。

2. DMT のセットは、*rename* で得られたタグを含む、デコードされた命令の情報の命令ウィンドウへ書き込みと同時に行われる。この書き込みの遅延は、通常、ウィンドウを構成する RAM の行デコーダとワード線の遅延に支配される。c のデコードはこの行デコーダと同じ回路で実現できるから、その遅延が表面化する可能性は低い。

以上から、この方式における更新処理がクロック速度を低下させる可能性は低いと言える。

3.3 DMT の縮小

依存する I_p と I_c の間の距離は短い場合が多く，32 命令以下の場合が 90%程度以上を占めることが分かっている [6]。本節では，この性質を利用して，前節で述べた *wakeup* を更に高速化する手法について述べる。

前節で述べた DMT は $(WS-1)$ b$\times WS$ word であったが、このビット数を $w(0 \leq w < WS-1)$ b に削減することによって *wakeup* の遅延を更に短縮することを考える。この w を **DMT の幅**と呼ぶことにする。依存する命令間の距離が DMT の幅以下の場合には、DMT によって rdyL/R を更新する。幅を越えていた場合には、クロック速度に影響を及ぼさない別の方法を用い、IPC に対するペナルティに転化するのである。依存する命令間の距離は短い場合が多いので、ペナルティの影響は小さいと予想される。

命令間の距離が DMT の幅を越える場合への対応としては、以下の 2 つの方法が考えられる：

delay 幅 $w(w < WS-1)$ である縮小した、幅 $WS-1$ である DMT を用意し、1 サイクル余分にかけて *wakeup* を行なう。

rdyL/R の更新が 1 サイクル遅れるため、I_c は I_p に引き続くサイクルには発行されない。

stall 縮小したードの段階でフロント・エンドをストールさせる。

I_p の実行が終了した後にデコードを再開すれば、rdyL/R は 1 に初期化される。

DMT の縮小の方法 図 5 に、DMT の縮小の様子を示す。左は元々の、右が縮小後の DMT である。同図では、$WS=8$、$w=4$ である。元々の DMT から

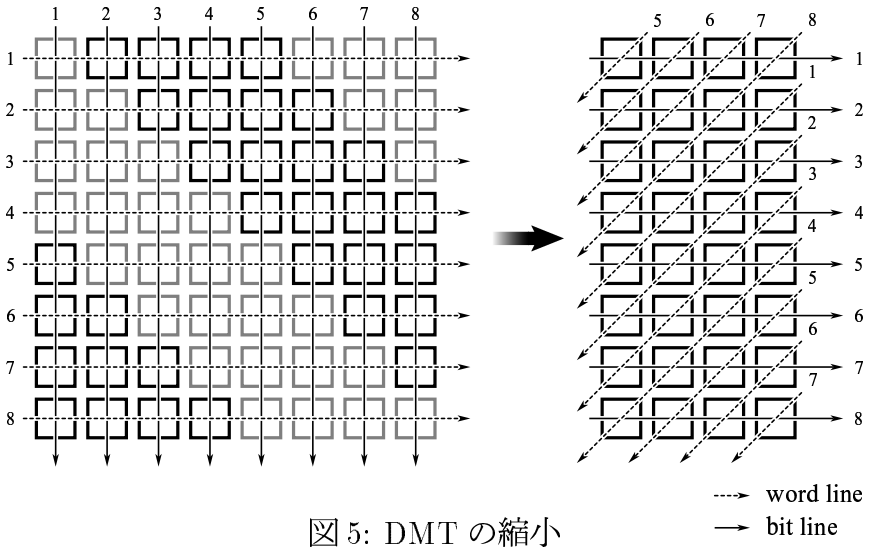

図 5: DMT の縮小

削除されるセルは、左図中で薄く示した。必要なセルを（矩形領域に）集める方法には任意性があるが、よりクリティカルであるビット線の長さを短縮することを優先して、図 5 右のようにするとよいであろう。

図 5 右から明らかなように、縮小された DMT のワード線、ビット線は、それぞれ w 個のセルにしか接続されておらず、それぞれの長さは WS とは無関係となっている。したがって $wakeup$ の遅延は、WS とは独立に、w によって決まる。

ウィンドウ・エントリの使用に対する制限　上では、命令間の距離という言葉を曖昧に用いていたが、DMT によって rdyL/R が更新できるのは、正確には、命令間の距離ではなく、ウィンドウ内のエントリ間の距離が DMT の幅以下の場合である。したがって、命令流上での距離とウィンドウ内での距離が（ある程度）一致するように制御する必要がある。その方法としては、命令ウィンドウのエントリをサイクリックに使用する、あるいは、コンパクションを行うなどが考えられる。前者は比較的容易であるが、後者にはかなりの量のハードウェアが必要である。

このことは特に、命令ウィンドウを複数の命令キューによって構成する場合に問題となる。サイクリックに使用する場合には、異なる命令キュー間では、パディングによってエントリの消費の歩調を揃える必要がある。例えば R10000 の命令キューの構成では、各キューのサイズは同じであるから、例えばあるサイクルに 2 つの整数命令が整数命令キューに格納されるとすると、他のキューで

も 2 つのエントリを無駄に消費する必要がある。

パディングの方法　ここで、パディングの方法を 2 通り述べる。

order　命令オーダーで自分より後の命令は必ず自分と同じ段か、それ以降のエントリに入る、とするものである。DMT で指す事のできる宛先は、自キューの 1 つ先のエントリから w先のエントリまでと、他キューの同じ段のエントリから $(w-1)$ 先のエントリまでである。

pack　order よりもキューの使用効率を上げるように、サイクルごとに最高でも IW_q 段のエントリのみを使用し、そこに詰め込むというものである。DMT で指す事のできる宛先は、IW_q=2 の場合は自キューの 1 つ先のエントリから w 先のエントリまでと、他キューの 1 つ前のエントリから $(w-2)$ 先のエントリまでである。IW_q=4 の場合は他キューの 3 つ前のエントリから $(w-2)$ 先のエントリまでとなる。

Integer 命令を Int、Load/Store 命令を L/S と略することにする。また、各命令の区別のために番号をつける。例えば、IW_q =2 で、命令を Int_0、Int_1、$\mathrm{L/S}_0$、$\mathrm{L/S}_1$、$\mathrm{L/S}_2$、$\mathrm{L/S}_3$、Int_2、Int_3 という順で fetch した場合は、パディングを行いながらこの 8 命令を命令キューに格納していくと、図 6のようにエントリを使用する。order の (empty) となっているエントリが無駄に消費されているわけである。また、同図には、w=4 の場合、Int_1 が DMT によって $wakeup$ できるエントリを、矢印で示してある。

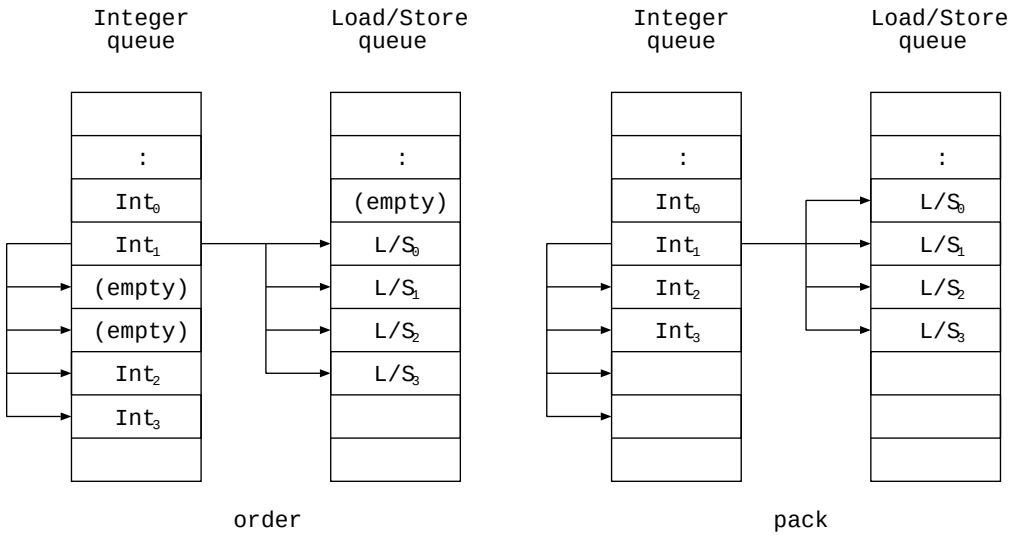

図 6: パディングの方法

第 4 章　シミュレータへの実装

本節では、IPC の測定をするために用いたシミュレータとそれに施した変更について説明する。

4.1 SimpleScalar

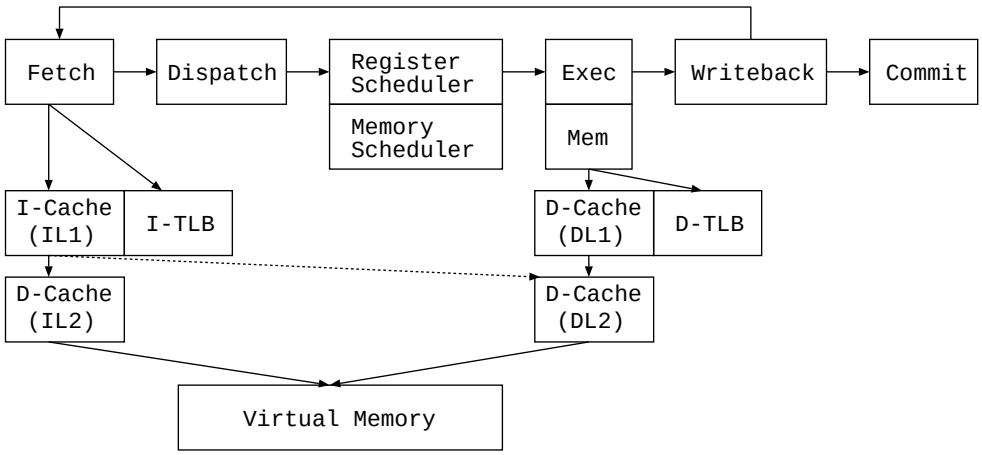

図 7: Simplescalar sim-outorder の構成図

IPC の測定のためのシミュレータは、SimpleScalar ツールセット (Ver.2.0)[2] の sim-outorder を元にした。sim-outorder は、MIPS アーキテクチャを拡張した SimpleScalar アーキテクチャを採用しており、レジスタ更新ユニット (RUU:register upgrade unit) に基づいて、動的命令スケジューリングを行なうアウトオブオーダ・スーパースカラプロセッサモデルである。構成図 [1] を図 7に示す。

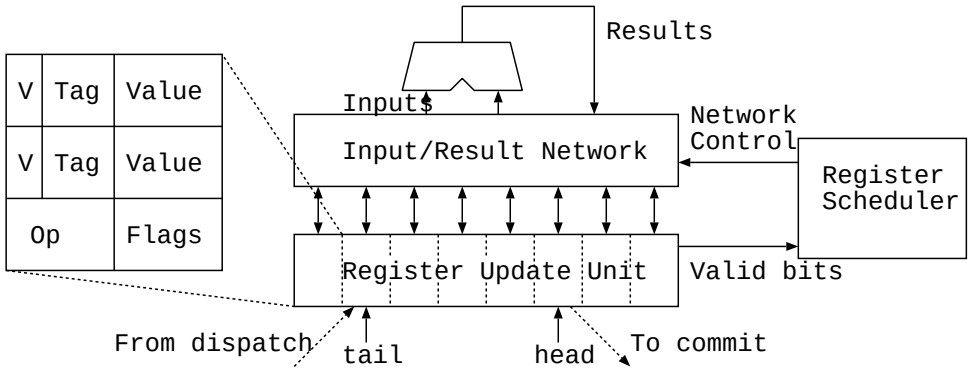

図 8: Register Update Unit(RUU) の概略図

図 8に RUU の概略図を示す。RUU はレジスタリネーミングを行い、依存する命令間の結果を保存する。リオーダ・バッファとリザベーションステーションの両方をあわせた役割をする。

RUU はサイクリックキューとなっており、命令エントリは後述する ruu_dispatch() において追加され、ruu_commit() において削除される。エントリには命令、フラグ、使用するレジスタの値とタグ、有効ビット等の情報が格納される。

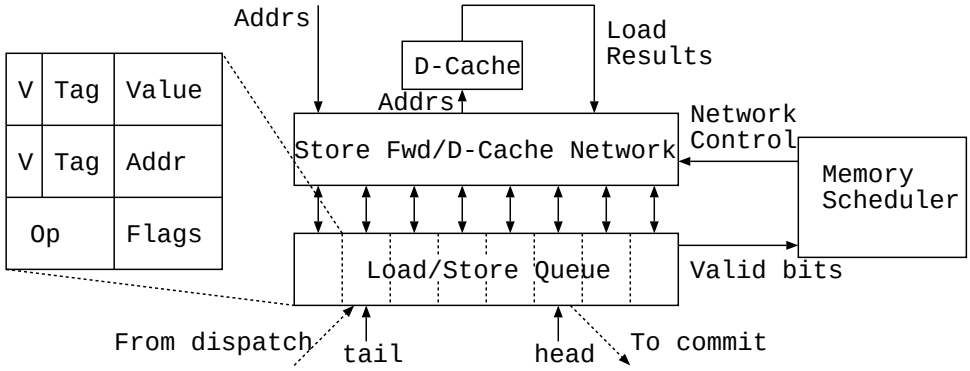

図 9: Load/Store queue(LSQ) の概略図

図 9に Load/Store queue(LSQ) の概略図を示す。ロードストア命令は、インオーダに発行する必要があるため、RUU とは別に LSQ を実装しており、整数および浮動小数点演算命令とは別格扱いをする。LSQ の構造は RUU と似ているが、レジスタの値の代わりにアドレスが格納されるようになっている。

シミュレータのメインループは以下のようになっている。

```
for(;;){
  ruu_commit();
  ruu_writeback();
  lsq_refresh();
  ruu_issue();
  ruu_dispatch();
  ruu_fetch();
}
```

この 1 ループが 1 マシンサイクルに該当する。パイプラインの最終ステージから逆順に巡ることにより、各パイプラインステージを 1 マシンサイクルあたり

1 度ずつだけ実行しつつ、ステージ間の依存関係を正しく処理できている。

以下では、各ルーチンの概要を説明する。

命令フェッチステージは ruu_fetch() に実装されている。命令キャッシュから命令をフェッチして命令フェッチキュー (instruction fetch queue:IFQ) に送り込む。

ruu_dispatch() において、IFQ の先頭の命令から順に decode と rename、そして RUU/LSQ へのエントリ追加を行なう。ロード・ストア命令はアドレス計算部とメモリアクセス部に分割され、前者は RUU に add 命令として、後者は LSQ にメモリアクセス命令としてそれぞれ登録される。そして、RUU 側から LSQ 側へリンクが貼られる。add 命令によりアドレスが計算されると、その値はメモリアクセス命令のアドレスオペランドに入り、使用可能となるのである。また、分岐においてはレジスタファイルのコピーをとっておく。

命令発行ステージは ruu_issue() と lsq_refresh() に実装されている。ここでは、wakeup と select を、レジスタ依存関係およびメモリ依存関係をもとに行なう。使用するオペランドがすべて揃った時点で命令は発行可能となり、ready_queue に送られる。ready_queue では fetch されたオーダーに従ってソートされ、その先頭から順に、実行装置が空いていれば発行していく。すなわち *select* の機構がここに実装されている。ロード命令はストアバッファに値がある時はそれをバイパスしてやる事はできるが、そうでなければ実際にメモリシステムから読み出す必要がある。その管理を lsq_refresh() が行なっている。

実行ステージも ruu_issue() に実装されている。実行装置が使用可能であれば、実際に命令が発行され、各命令の所要サイクル後に ruu_writeback() にて結果の書き戻しを行なえるようにする。

ruu_writeback() においては、実行が終了した命令の、実行結果の書き戻しを行なう。実行結果を待っている命令に対して値を与え、その命令が発行可能となったならば ready_queue に送る。また、分岐予測ミスの検知もこのルーチンで行う。分岐予測ミスが発生したと分かれば、RUU のエントリの分岐命令以降の間違って発行された分の命令を捨て、ruu_dispatch() で取っておいたレジスタファイルのコピーを書き戻す。

ruu_commit() においては、RUU の先頭から順番に、命令のインオーダ・コミットを行う。ストア命令によるデータキャッシュの更新もここで行う。

4.2 キューと DMT の実装

マシンコンフィグとして R10000 のそれを用いるのだが、オリジナルの SimpleScalar sim-outorder では、整数、浮動小数点、ロードストアという複数のキューにあたるものは存在しない。そのため、以下に示すような変更を加え、仮想的な複数のキューと DMT の動作を実装した。

まず、キューをサイクリックに使用するために、キュー内での head と tail を示すものが必要である。tail にあたるものとして、各命令系列ごとにシリアルカウンタを用意する。また、RUU のエントリにシリアル番号を保存するための領域を追加しておく。RUU に命令を追加するときに、命令にシリアル番号をふっていき、追加された部分に保存しておく。RUU には commit されるまでの命令が保存されているが、そのうちで未発行であるものが各命令キューに入っているものにあたり、そのうちで最小であるシリアル番号が head にあたる。head と tail の差をとれば、各キューの使用エントリ数が分かる。

tail の変更は ruu_dispatch() で命令を追加した時と、および分岐予測ミス発生時の wrong path 切り捨て時にのみ起こる。前者は追加時にカウンタから RUU のエントリへのコピーをしたのち、当該カウンタのインクリメントを行えばよいが、後者を簡単に実現するために、RUU に追加した領域をさらに拡張し、ほかのキューのカウンタも同時に保存する。実際に分岐予測ミスが判明して RUU からの wrong path 切り捨てが終わった後で、カウンタを書き戻すのだが、当該分岐命令の追加エントリに保存されているカウンタのコピーは、この分岐命令を命令キューに格納する前のものであるので、書き戻した後で、分岐命令の格納されるキュー、すなわち整数のキューのカウンタを 1 進めておく必要がある。

head の変更は、ruu_issue() において命令を発行し、キューのエントリを消去した後に起こる。実装では、そのサイクルでの発行終了後に RUU を頭からなぞり、各キューの head を算出する。ここで注意しなければならないのは、ここで算出した head は次のサイクルで有効となることである。

ここまでを例によって説明する。最初は、図 10(a) のように head=tail(=カウンタ)=0 である。図 10左の 3 命令を追加したとしよう。最初の命令はシリアル番号 0、後続は 1 と 2 となる。エントリの右側が与えられたシリアル番号である。追加後は tail は 3 となる。図 10(b) の状態となり、ここでは使用エントリ数は 3-0=3 である。まず、ここから最初の div r1,2 が発行されたとすると、図

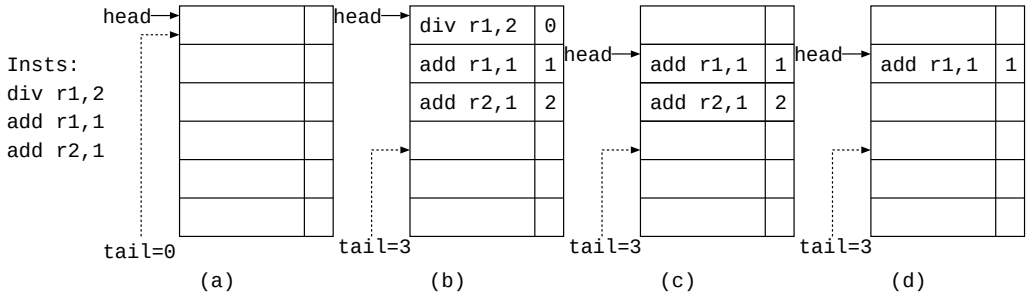

図 10: 命令キューの動作例

10(c) の状態となり、未発行命令のうちで最小の番号は 1 であるので、使用エントリ数は 3-1=2 となる。これに引き続いて add r2,1 が発行されても、図 10(d) のように head、tail はともに不変である。

命令キューのエントリのリセットに 1 サイクルを要する場合は、算出した head は次のサイクルではなく、さらにその次のサイクルにて有効となるようにすればよい。

パディングを行う場合は、各キューの head が共通のものとなる。order の場合は、ruu_dispatch() において当該キューのカウンタをインクリメントした後で他のキューのカウンタで差が 2 以上開いているものを、その差が 1 になるようにすることによって実現できる。図 11 にその様子を示す。左の状態に対して Integer の命令を追加した場合、まず Integer の tail が 1 進む。この時点で FP の tail は Integer のそれとの差が 2 になっているので、これを差が 1 になるように、FP の tail も 1 進める。結果として、図 11 の右の状態となり、FP の命令キューには命令の格納されていないエントリ (empty) が追加されたことになる。

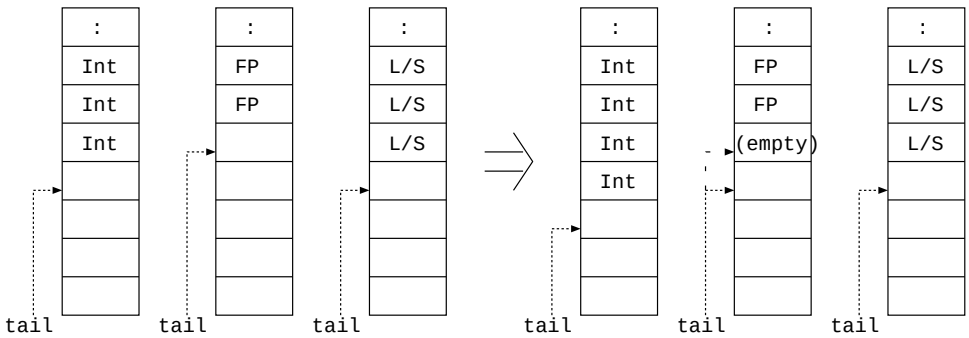

図 11: order におけるパディングの様子

また、pack の場合は 1 サイクルごとにカウンタを最大のものに歩調を合わせ

れば良い。これは、サイクルの終わり、ruu_fetch() よりも後で行なっている。

DMT の縮小に対しては、以下のようにした。delay については、ruu_writeback() において結果の書き戻しを行う際に I_c と I_p の間のシリアル番号を比較し、DMT の幅以内である場合はここで普通に書き戻しを行うが、幅を超えている場合はこのサイクルの終わり、ruu_fetch() よりも後で書き戻しを行うようにすることで、1 サイクルの遅延を実現している。

stall については、ruu_dispatch() において RUU のエントリに命令を追加する際に、I_c と I_p の間のシリアル番号を比較し、DMT の幅を超えていた場合はここで ruu_dispatch() を中止し、ruu_fetch() も行わないようにすることで実現している。IFQ の先頭が当該命令のままであるので、I_p の実行が終了するまでは、ずっと ruu_dispatch() および ruu_fetch() は実行されなくなる。

4.3 実証

これまで述べてきた実装方法で、それぞれの機構が正しく動作することを実証するため、簡単なテストプログラムを SimpleScalar 附属のアセンブラで作成した．その内訳は以下のとおりである．

```
add r1,1      div r1,2     div r1,2
add r2,1      add r1,1     add r1,1
add r1,1      add r2,1     add r2,1
add r2,1          :            :
add r1,1      add r15,1    add r15,1
add r2,1      add r16,1    add r16,1
    :         add r17,1    ld  r17,(adr)
                  :        ld  r18,(adr)
                               :

  (a)           (b)          (c)
```

図 12: テストプログラム

delay と stall ：図 12(a) のように、R1 への add と R2 への add を繰り返すプログラムを組んだ．DMT の幅が 2 以上である場合は、レイテンシなしに 2 つ先の命令に結果を渡すことが可能であり、1 サイクルに 2 命令を発行できるが、DMT の幅が 1 である場合は、delay の場合はあるサイクルに 2 命令実行した次のサイクルは、後続の命令を発行できない．stall の場合は、1 サイクルに 1 命令のみ発行できる．

リセット　：図12(b)のようなプログラムを組んだ．先頭のdivはすぐに発行されるが、計算に時間がかかるため、直後のadd r1,1はずっと発行されないまま命令キューに残る．それより後の依存関係のないaddはr1へのaddより先に発行されていく．結果、命令キューは使い尽くされる．divが終了し、add r1,1が発行されると、resetなしの場合は次のサイクルで命令キューに後続の命令add r17,1が格納されるが、resetありの場合はadd r1,1が発行された2サイクル後にならないと格納できない．

パディング　：図12(c)のようなプログラムを組んだ．整数の命令キューが使い尽くされる所まではリセットと同じである．その後、ロード命令はパディングを行なわない場合はWS個格納できるが、パディングを行う場合はr1へのaddが発行されるまでは1個か、多くても2個しか格納できない．

以上のすべてについて、命令の発行と命令キューの様子をモニタリングできるようにシミュレータを改造して、WS=16、$\mathrm{IW}_q = 2$のコンフィグにおいて実行した結果、それぞれ述べたとおりの結果を得た．

第5章　評価

従来の方式と高速な方式との定量的な比較を行う。比較項目は、3.3節で述べたDMTの縮小に対するペナルティと，回路の遅延である。前者は5.1節で、後者は5.2節で、それぞれ述べる．

5.1　IPCの評価

測定条件　SimpleScalarツールセット(ver.2.0)に対して、前節で述べた実装を行い、SPECベンチマークを用いてDMTの幅に対するIPCの変化を測定した。

ベース・モデルとしては、MIPS R10000[5]のマシン構成を用いた。R10000は、ロード/ストア、整数演算、浮動小数点演算のそれぞれに命令キューを持ち、$(IW_q, IW, WS_q, TW) = (2, 4, 16, 6)$である。1次キャッシュは、命令/データ、それぞれ、容量32KB、ライン・サイズ32Bである。2次キャッシュは命令/データ共有で、容量1MB、ライン・サイズ64Bである。レイテンシは、1次は1サイクル、2次は6サイクルである。2次キャッシュ・ミス時は、最初のデータが得られるまでが18サイクルで、後続データへのアクセスには2サイクルが必要である。メモリアクセス命令の分割が行なわれるのと、命令セットの相違との

表1: 各命令のレイテンシ

ユニット	レイテンシ	リピート・レート	命令
Integer ALU 1/2	1	1	add,subtract,logical,move Hi/Lo,trap
Integer ALU 1	1	1	integer branches
Integer ALU 1	1	1	shift
Integer ALU 1	1	1	conditional move
Integer ALU 2	10	10	multiply
Integer ALU 2	67	67	divide
FP adder	2	1	add,subtract,compare
FP multiplier	2	1	multiply
FP divide	12	14	32-bit divide
FP divide	19	21	64-bit divide
FP square root	18	20	32-bit square root
FP square root	33	35	64-bit square root
adress calculator	1	1	memory adress calculation
memory-port	1	1	load value
memory-port	-	1	store value

ユニット	個数
Integer ALU 1	1
Integer ALU 2	1
FP adder	1
FP multiplier	1
address calclator	2
memory-port	2

表2: 各ユニットの個数(R10K×1)

プログラム	入力セット	実行命令数
099.go	9 9	132M
124.m88ksim	dcrand.big	120M
126.gcc	genrecog.i	122M
129.compress	10000 q 2131	35M
130.li	train.lsp	183M
132.ijpeg	vigo.ppm -GO	26M
134.perl	primes.in	10M
147.vortex	persons.250	157M

表3: ベンチマークプログラム

ため、各命令の所要サイクルは表1のように、また、各演算ユニットの個数は表2のように設定した。ロードストア命令のアドレス計算は、add 命令として Integer ALU を使用しているのを、アドレス計算専用の address calculator を使用するように変更した。

分岐予測には、ツールセットに用意されている 2b 飽和型カウンタによるもの (bimod) を用いた。これを、構成 R10K×1 と呼ぶことにする。また、キャッシュ – メモリ以外のすべての資源を 2 倍、すなわち、$(IW_q, IW, WS_q, TW) = (4, 8, 32, 7)$ としたものも合わせて測定した。これを構成 R10K×2 と呼ぶ。

また、使用した SPEC ベンチマークプログラムとその入力セットを表3に示

表 4: リセットとパディングによる IPC の低下率 (%)

		099	124	126	129	130	132	134	147	av.
R10K×1	reset	0.23	0.01	0.12	0.24	0.19	0.75	0.23	0.19	0.25
	padding	0.18	0.04	0.31	0.08	1.12	0.44	0.38	1.23	0.47
	both	0.46	0.10	0.53	0.26	1.54	0.96	0.29	1.75	0.74
R10K×2	reset	0.23	0.02	0.10	0.16	0.12	0.77	0.01	0.01	0.18
	padding	0.18	0.02	0.19	0.09	0.09	1.01	0.24	0.38	0.28
	both	0.33	0.05	0.30	0.25	0.31	0.76	0.34	0.38	0.34

す。入力セットは、現実的な時間でシミュレートが終了するように調整選択してある。これらの 8 つのプログラムはすべてウイスコンシン大学で配布している、SimpleScalar アーキテクチャをターゲットとしてコンパイル済みのバイナリを使用した。

測定結果　まず、DMT の縮小の前に、3.2.2 節で述べた DMT エントリのリセットとパディングの影響を調べる。表 4 に、構成 R10K×1 の従来方式に対して、リセット、パディング、および、その両方を組み込んだ場合の IPC の低下率を示す。リセットとパディングによる IPC の低下率は 1%程度以下である。後述する DMT の縮小による IPC の低下率は数%程度はあるから、リセットとパディングによる影響はそれに比べてほとんど無視できる。これより、キューの使用率はそれほど高くはないことが予想される。

次に、DMT の縮小を行った場合の、2 通りのパディング方法による、従来方式に対する IPC の低下率の違いを計測した。これは、R10K×1 において、129.compress と 134.perl についてのみ実行してみた。この結果を図 13 に示す。なお高速な方式には、リセットを組み込んである。また、DMT の幅が WS_q の場合でもパディングを行っている。

この結果からは、どちらの方式でもほとんど変わらないが、少しだけ order の方が値が良いことがわかる。先に述べた、キューの使用率はそれほど高くはないということがここでも言える。

最後に、DMT の縮小を行った場合の、従来方式に対する IPC の低下率を図

=

図 13: パディング方法の違いによる DMT の幅に対する IPC の比

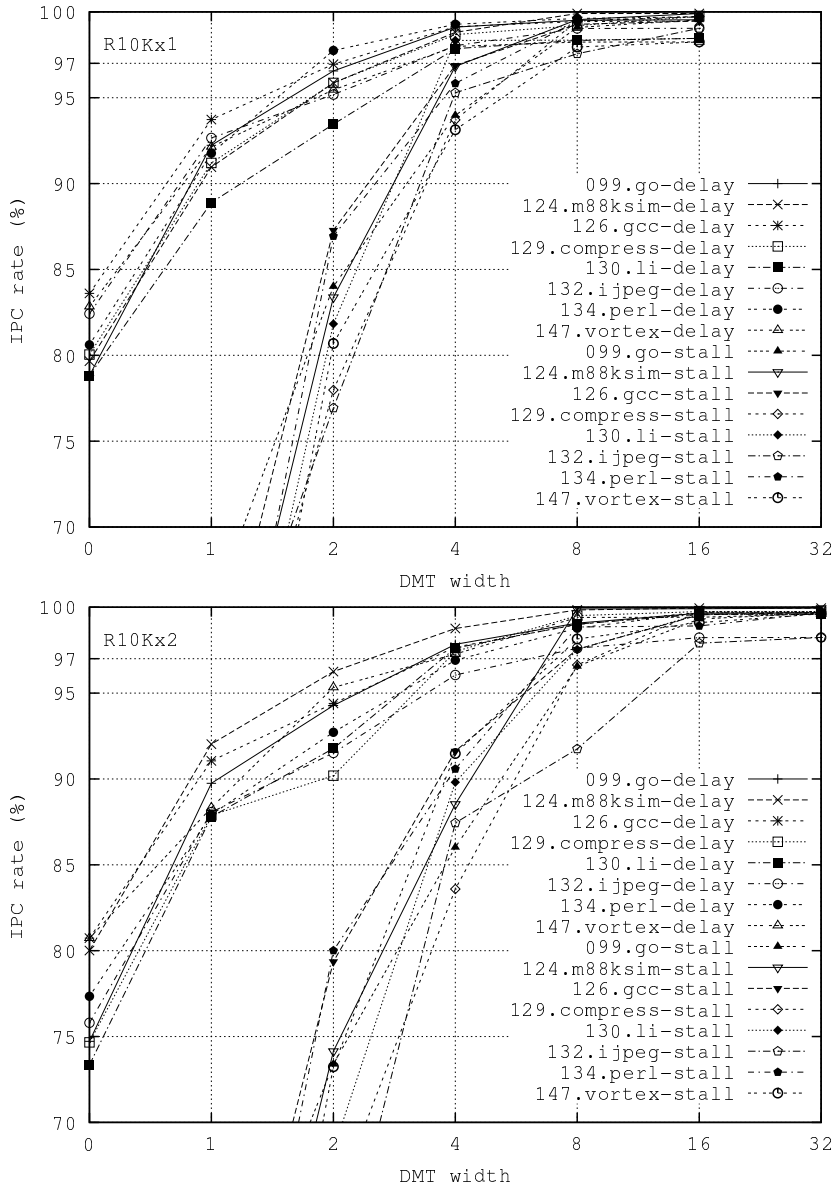

図14: DMT の幅に対する IPC の比

14 に示す。なお、ここでもリセットおよびパディングを組み込んでいる。

この結果からは、DMT の幅は、delay ならば WS_q の 1/4、stall でも 1/2 あれば、従来方式に対する IPC の低下は 3%以下に抑えられることが分かる。

ちなみに、DMT の幅を 0 とした場合の delay の値は、ちょうど、2.2 節で述べた、*wakeup* と *select* にそれぞれ 1 サイクルかけ、ALU からはバイパスを行わない場合に相当する。その場合の IPC は低下は、30%程度になる。

stall で、DMT の幅 0 の delay よりも IPC が低下する構成には意味がないた

表 5: *wakeup*+*select* の遅延 (ps)

Delay of *wakeup*+*select*(ps)

		R10K×1		R10K×2	
		wakeup	*wakeup*+*select*	*wakeup*	*wakeup*+*select*
DMT	1	65.0(13.4)	339.3(44.7)	65.0(9.6)	411.8(40.2)
	2	75.1(15.5)	349.4(46.0)	75.1(11.1)	421.9(41.2)
	4	100.7(20.7)	375.0(49.4)	100.7(14.8)	447.5(43.7)
	8	149.0(30.7)	423.3(55.7)	149.0(22.0)	495.8(48.4)
	16	242.0(49.8)	516.3(68.0)	242.0(35.7)	588.8(57.4)
	32			429.4(63.3)	776.2(75.7)
RAM+CAM		485.5(100.0)	759.8(100.0)	678.2(100.0)	1025.0(100.0)

め、グラフには示されていない。

5.2 回路の評価

我々は、富士通株式会社から提供された、.18μm CMOS プロセスのデザイン・ルールに基づいて、従来方式と高速な方式の主要な回路のレイアウト設計をスクラッチから行った。得られたレイアウトから、回路面積を求めた。また、プロセス・パラメータに基づいて RC データを抽出し、Hspice シミュレーションによって遅延を求めた、以下、その結果についてまとめる.

5.2.1 評価結果

各回路の評価でも, IPC の評価でも用いたのと同じ、構成 R10K×1 と R10K×2 を対象とした。それぞれの (IW_q, IW, WS_q, TW) は、前述の通り、$(2,4,16,6)$、および、$(4,8,32,7)$ である. DMT に対しては, 幅が 1, 2, 4, 8, 16, 32 の 6 種を評価した。

遅延 表 5 に、*wakeup* と *select* の遅延をまとめる。表中、() 内の数値は、従来方式に対する遅延の比 (%) を表す。

構成 R10K×1 の場合、提案方式の *wakeup*+*select* の遅延は、従来方式のそれの 68.0% となる。この場合、提案方式の動作周波数の上限は 1.93GHz となる。DMT の幅を 4 まで縮小すると、前節の結果から、delay 方式ならば IPC は 3%程度低下するが、*wakeup*+*select* の遅延は 375.0ps、動作周波数の上限は 2.7GHz に達する。

同様に R10K×2 の場合では、高速な方式の *wakeup+select* の遅延は、従来方式のそれの 75.7%となる。提案方式の動作周波数の上限は 1.29GHz となる。DMT の幅を 8 まで縮小すると、*wakeup+select* の遅延は 495.8ps、動作周波数の上限 2.02GHz となる。この代償はやはり IPC の 3%程度の低下ですむのである。

第 6 章　おわりに

本稿では，高速な方式：命令間の依存関係を直接的に表すテーブル DMT を用いた動的命令スケジューリング方式について述べた。本方式では、小容量の RAM を読み出すだけで *wakeup* を実現することができる。

富士通株式会社から提供された .18μm CMOS プロセスのデザイン・ルールに基づいてレイアウト設計を行い、Hspice を用いて遅延を計測した。その結果、MIPS R10000 と同様の構成において、動的命令スケジューリング・ロジックの遅延は 517.2ps となり、従来方式の 71.3%にまで削減された。その場合の動作周波数の上限は 1.93GHz となり、動的命令スケジューリングがクリティカルとなる可能性は極めて低いと言える。

そして本稿では、DMT を縮小することによってその遅延を更に短縮する手法についても述べた。この手法において *wakeup* の遅延を IPC に対するペナルティに転化することができる。これらの手法をシミュレータに実装し、SPEC ベンチマークプログラムを実行することにより IPC に対するペナルティを測定した。すると、DMT を 4b×4word にまで縮小しても、IPC の悪化は 3%以下であることが分かった。R10000 の 2 倍の計算資源を持つ仮想的な 8-way の構成においても、DMT の縮小によって、IPC の 3%以下の低下を代償に 2.02GHz の動作周波数を達成することができる。

以上から、スーパースカラにおいて、その動的命令スケジューリング・ロジックがクロック速度を規制することは将来に渡ってなくなり、そのために必要な代償もわずかである、と言うことができる。

ただ、今回は SPEC95Int の 8 プログラムについてのみのシミュレーションに留まったが、今後は SPEC95FP のプログラムについても同様に実行していく予定である。

謝辞

本研究の機会を与えて頂いた，本研究室の富田眞治教授に深甚な謝意を表します.

また，本研究に関して適宜御指導，御鞭撻を賜った森眞一郎助教授，中島康彦助教授，北村俊明助教授，五島正裕助手に深く感謝致します.

さらに，共同研究者である縣 亮慶氏、Nguyen Hai Ha 氏をはじめとして，日頃様々な角度から助力して下さった京都大学大学院情報学研究科通信情報システム専攻富田研究室の諸兄に心より感謝致します.

参考文献

[1] Burger, D., Austin, T. M. and Bennett, S.: Evaluating Future Microprocessors: The SimpleScalar ToolSet, Technical Report CS-TR-1342, Univ. of Wisconsin-Madison (1997).

[2] Burger, D. et al.: SimpleScalar Tools Home Page, http://www.cs.wisc.edu/~mscalar/simplescalar.html.

[3] Keller, J.: The 21264: A Superscalar Alpha Processor with Out-of-Order Execution, *9th Annual Microprocessor Forum* (1996).

[4] Palacharla, S., Jouppi, N. P. and Smith, J. E.: Quantifying the Complexity of Superscalar Processors, Technical report, Univ. of Wisconsin-Madison (1996).

[5] Yeager, K. C.: The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, No. 4, pp. 28–40 (1996).

[6] 五島正裕, グェンハイハー, 森眞一郎, 富田眞治: Dual-Flow: 制御駆動とデータ駆動を融合したプロセッサ・アーキテクチャ, 情処研報 98–ARC–130, pp. 115–120 (1998).