修士論文

依存行列に基づく
Out-of-Order 命令スケジューリング方式

指導教官　　富田 眞治 教授

京都大学大学院情報学研究科
修士課程通信情報システム専攻

西野 賢悟

平成 15 年 2 月 7 日

依存行列に基づく Out-of-Order 命令スケジューリング方式

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。スーパースケーラのIPCを向上させる最も直接的な方法は、命令発行幅 (同時に演算器に発行できる命令数) とウィンドウサイズ (デコードが完了して発行待ち状態の命令を貯めておける場所の大きさ) を増やすことである。初期のスーパースケーラにおいては、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。

スーパースケーラは、Out-of-Order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ている。本稿ではwakeupを高速化する方式について述べる。本方式は、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行列を用いるもので、単にRAMを読み出すことでwakeupを実現することができる。

一方、行列を用いた類似の方式がDEC Alpha 21264などで採用されている。これらの違いを明らかにし、さらに、このロジックの遅延をIPCに対するペナルティに転化し、高速化する手法を示す。富士通株式会社から提供された0.18μm 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、シミュレーションによって、ペナルティを測定した。その結果、1%程度のペナルティを代償に、我々の方式の回路遅延は、21264の方式の1/2程度以下であることが分かった。また、命令発行幅とウィンドウ・サイズを無限大にした場合でも行列のサイズは16 ~ 64程度あればよく、提案方式によって、命令スケジューリングの遅延はウィンドウ・サイズとほぼ独立にできることが分かった。

A Dependence Matrix based Out-of-Order Instruction Scheduling Scheme

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 match 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. This paper describes a high-speed dynamic instruction scheduling scheme. This 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.

A similar scheme based on matrices, however, has already adopted in DEC Alpha 21264. This paper describes differences of them, and a scheme which changes the delay of the logic into IPC penalty. We actually designed the logic guided by a design rule of a real 0.18μm CMOS process, measured the areas, and calculated the delays by Hspcie. And we also evaluated the penalty of the scheme by simulation. The evaluation result shows that the circuit delay of our scheme is less than half of that of the scheme adopted in 21264, with the IPC penalty about 1%. The result also shows that the matrices size of 16 to 64 is enough even for the processor which have an infinity of the issue width and the window size, and that our scheme let the delay of instruction scheduling almost independent of the window size.

依存行列に基づく Out-of-Order 命令スケジューリング方式

目次

第 1 章　はじめに　1

第 2 章　Out-of-Order 命令スケジューリング　3

2.1　Out-of-Order 命令スケジューリング の原理 3

2.2　連想方式 4

2.2.1　連想方式の原理 4

2.2.2　連想方式の wakeup ロジック 6

2.2.3　命令ウィンドウの分散化 9

2.2.4　Wakeup の定義 10

2.3　関連研究 11

第 3 章　依存行列による方式　15

3.1　間接方式 15

3.1.1　間接方式の原理 15

3.1.2　間接方式のロジック 17

3.1.3　連想方式との関係 18

3.2　直接方式 19

3.2.1　直接方式 の概要 19

3.2.2　依存行列 の実装 21

3.2.3　依存行列 の更新 22

3.2.4　更新処理の遅延 23

3.3　依存行列 と投機 24

第 4 章　依存行列の改良　25

4.1　直接方式における依存行列の単一化 25

4.1.1　ウィンドウ・エントリと物理レジスタの寿命 25

4.1.2　行列の単一化 26

4.2　依存行列 の高速化 28

4.2.1　依存行列 の分散化 28

4.2.2　L-1 依存行列の狭幅化 31

第 5 章　評価 **34**
5.1　IPC の評価 34
5.1.1　SinpleScalar 34
5.1.2　IPC の評価 39
5.2　回路の評価 42
5.2.1　評価方法 42
5.2.2　回路の説明 43
5.2.3　回路面積の評価 44
5.2.4　回路遅延の評価 46
第 6 章　おわりに **48**
謝辞 **50**
参考文献 **51**

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) の積によって算出される。スーパースケーラの 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: Window Size) を増やすことである。実際 初期のスーパースケーラは、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 が向上するわけではないので、いたずらに増加させれば、かえって全体の性能を悪化させることになる。

スーパースケーラの構成要素のうち、**wakeup** と呼ぶロジックの遅延が、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。wakeup は、Out-of-Order 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 の遅延は、IW,WS の増大、LSI の微細化、パイプラインの深化にともなって、スーパースケーラのクロック・スピードを制限する主要因の 1 つとなると予測されるのである。

このような背景から我々は、wakeup を高速化する新しい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。

一方、DEC Alpha 21264 では、行列を用いた類似の方法がすでに採用されている。両者では、行列を用いて wakeup を実現するという点で一致しているが、依存関係の表現の方法が異なる。本稿では、従来の方法と、2 つの依存行列による方法について述べ、さらにこの依存行列を用いた方式を高速化する手法と、定量評価の結果を示す。

以下、まず 2 章では従来の Out-of-Order 命令スケジューリング 方式の一般的な構成法と関連研究を紹介し、3 章で依存行列を用いた、我々の方式と DEC Alpha 21264 の方式の両方について詳しく述べる。4 章では依存行列をさらに改良する方式を述べ、そして 5 章で、各方式の定量的評価を行う。

第 2 章　Out-of-Order 命令スケジューリング

スーパースケー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 Out-of-Order 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 [1, 2] 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。たとえば、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 .7emIII、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 [1]。

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。本稿で述べる依存行列による方式も、後述するとおり、これにあてはまる。

以下まず 2.1 節において Out-of-Order 命令スケジューリングの原理についてまとめ、2.2.1 項において従来の方式である連想方式について説明する。2.2.2 項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。2.2.3 項では命令ウィンドウの分散化について説明し、2.2.4 項で wakeup の定義を明らかにする。また、2.3 節では関連研究の紹介を行う。

2.1　Out-of-Order 命令スケジューリング の原理

Out-of-Order スーパースケーラ・プロセッサ は、論理的なレジスタとは別に、各命令の Out-of-Order な実行結果を保存するための物理レジスタを必要と

する。物理レジスタは、リオーダ・バッファ、あるいは、物理レジスタ・ファイルなどによって実現される。

Out-of-Order 命令スケジューリング は、この物理レジスタを用いたデータ駆動型の計算とみなすことができる。すなわち、命令ウィンドウ中で『眠っている』命令は、ソースに割り当てられた物理レジスタにデータが書き込まれることによって『起こされ』、プログラム・オーダとは独立に実行を開始することができる。

2.2 連想方式

2.2.1 連想方式の原理

従来から用いられてきた連想方式では、この物理レジスタの番号に基づいて wakeup を実現する。連想方式では、物理レジスタの番号はタグと呼ばれ、その連想処理によって wakeup すべき命令を検索する。

Out-of-Order 命令スケジューリング の処理は、以下で述べる 5 つのフェーズに従って進む；ただし、命令 I_l/I_rの結果を、命令 I_cがそれぞれ左/右のソースとして消費する場合を考える[1)]。$I_l/I_r/I_c$は、それぞれ、命令ウィンドウの $l/r/c$ 番エントリに格納され、また、I_l/I_rには、その結果の書き込み先として物理レジスタの L/R 番が割り当てられるとする：

(1)Rename

フェッチされた命令に対して、レジスタ・リネーミングが施される：

デスティネーション
まず、デスティネーションに対して、物理レジスタの 1 つが割り当てられる。この物理レジスタの番号を tagD と呼ぶ。

ソース
同時に、左/右のソースに現在割り当てられている tagD が求められる。これを、tagD と区別して、tagL/R と呼ぶ。
$I_l/I_r/I_c$に関しては、$\mathrm{tagL}[c] = \mathrm{tagD}[l] = L$、$\mathrm{tagR}[c] = \mathrm{tagD}[r] = R$ となる。

(2)Dispatch

その後命令は、ウィンドウに格納される。

[1)] 簡単のため各命令は左/右 2 つのソースを持つとしたが、3 つ以上のソースを持つ場合にも容易に拡張できる。

I_l/I_rがまだ実行されていない場合 I_cは、I_l/I_rの実行を待って、ウィンドウ内で『眠る』ことになる。

(3)Wakeup

連想方式の wakeup は、タグに基づく連想検索によって行われる。後述する select によって I_l/I_rの発行が許諾されると、tagD$[l/r]$ から L/Rが読み出され、ウィンドウ内のすべての tagL/R からの連想検索が行われる。tagL/R$[c]$ の内容が L/Rと一致するので、左/右のソースが利用可能であることを表すフラグ rdyL/R$[c]$ がセットされる。

(4)Select

rdyL/R が共にセットされている命令が発行可能な命令である。Select では、発行要求が調停され、選択された命令の発行が許諾される。

(5)Issue

発行を許諾された命令の情報が命令ウィンドウから読み出され、実行ユニットに送られる。

wakeup と select のループ

Wakeup から select へは、rdyL/R を介した依存が存在する。また、I_cに対する wakeup が開始できるのは、select が I_l/I_rの実行を決定してからである。すなわち、wakeup と select は、緊密なフィードバック・ループを形成している。以下では Out-of-Order 命令スケジューリング ・ロジックのクリティカル・パスについて考えるが、そこではこのループが特に重要である。

このフィードバックループが存在するため wakeup と select はパイプライン化できない。現在、スーパースケーラのパイプラインはますます深くなる傾向にある。ため、このループ部分はパイプラインが深くなると相対的にクリティカルになる。

これらのうち wakeup には、LSI の微細化の恩恵を受けにくいという問題もある。select の遅延はもっぱらゲート遅延からなるため、微細化にともなって順調にスケーリングされる。一方 wakeup の遅延は主に RAM と CAM のワード線、ビット線、マッチ線などの配線の遅延からなるため、スケーリングされにくい。したがって、LSI が微細化されるにつれ、wakeup の遅延は相対的にクリティカルになる。

2.2.2 連想方式の wakeup ロジック

図 1 に、連想方式の wakeup ロジックのブロック図を示す。連想方式の wakeup ロジックは、tagD を格納する RAM 部と、tagL/R をキー、rdyL/R を内容とする CAM 部からなる。同図中、*IW* は命令発行幅 (Issue Width) を、*WS* はウィンドウ・サイズを表す。

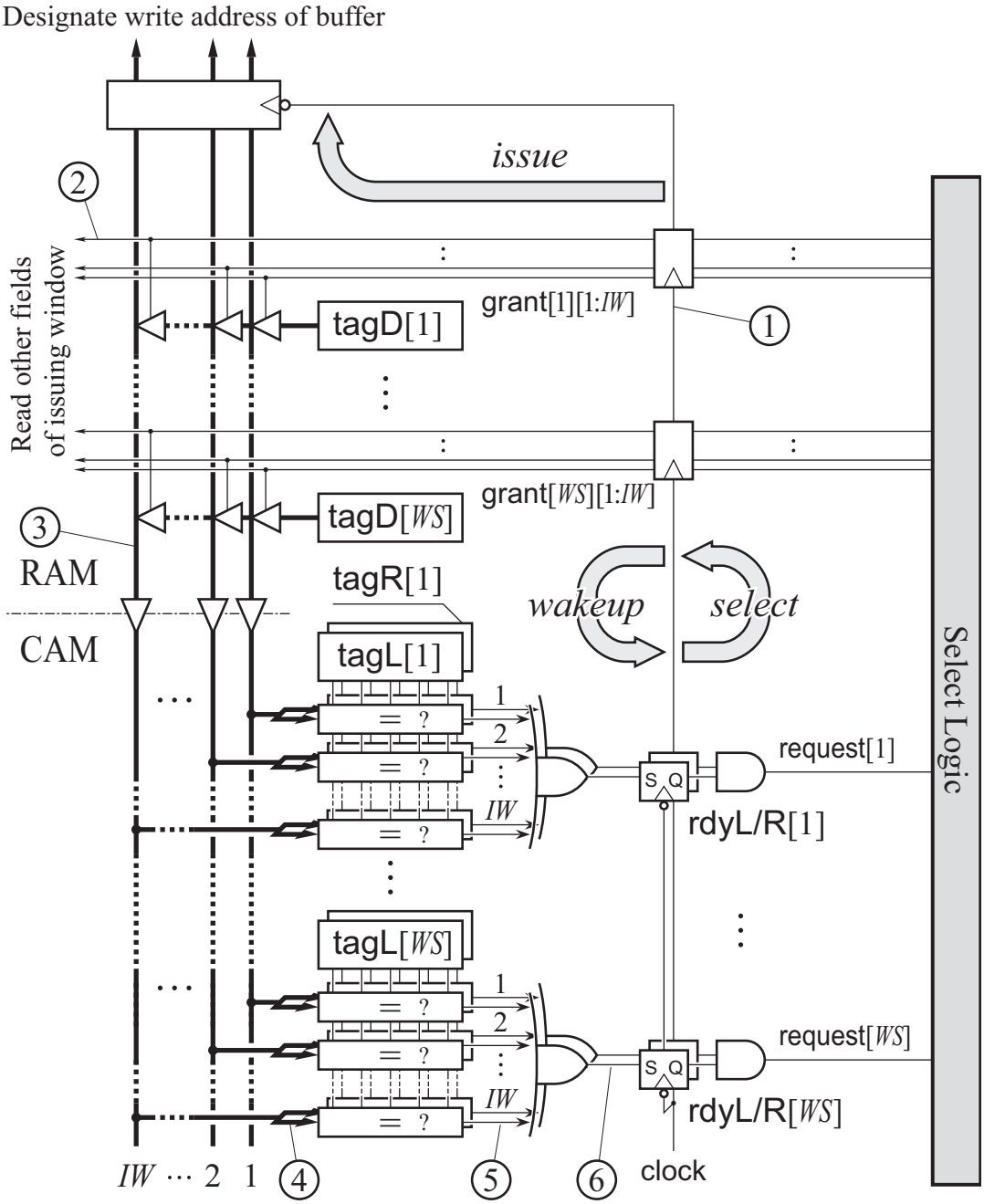


図 1: 連想方式の wakeup ロジック

RAM 部からは、tagD が読み出される。この RAM 部は、通常の RAM とは異なり、行デコーダを持たない。読み出しワードラインには、パイプライン・ラッチを介して、select ロジックからの発行許諾信号 grant が直接接続される。読み出される tagD は、1 書き込むべき物理レジスタを指示するために上部のパイプライン・ラッチに、2 wakeup のために下部の CAM 部に、それぞれ送られる。tagD の読み出しは、したがって、issue と wakeup の両方に含まれることになる。なお、上部のパイプライン・ラッチに送られた tagD は、適当なサイクル数だけ遅延され、物理レジスタの書き込みアドレスとして使用される。

CAM 部では、比較入力ポートに入力された tagD と一致する tagL/R が連想検索され、対応する rdyL/R がセットされる。CAM 部の出力側では、rdyL/R を記憶するレジスタの出力信号が、組合せ回路的に select の入力 request に接続されている。

Select ロジックは、WS 個の request を調停し、wakeup ロジックに対して IW 組の grant をアサートする。このように、wakeup と select は、緊密なフィードバック・ループを形成している。

ロジックの動作

図 2 (L-1) に、wakeup、select、issue の命令パイプラインでの位置づけとその動作の様子を示す。同図は、MIPS R10000 のパイプライン構成に準ずる [3]。図中、Exec は実行を、RF→と→RF は物理レジスタ・ファイルに対する読み出しと書き戻しを表す。同図は、タイミングが最もクリティカル、すなわち、レイテンシが 1 である命令 I_p の次のサイクルで I_c が実行される場合を示している。

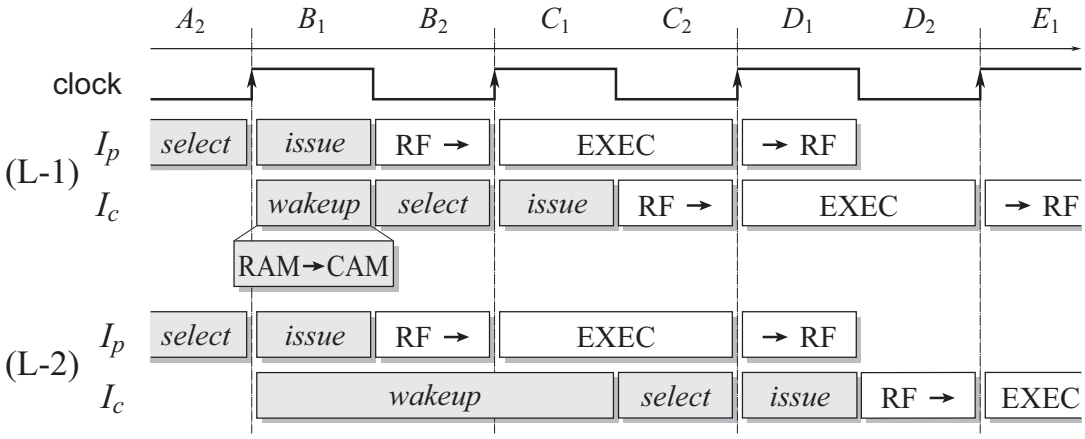


図 2: 命令パイプラインにおける wakeup、select、issue

I_p が生成したデータは、オペランド・バイパスを通って I_c の実行に使用される。

ロジックの動作は、以下のようにまとめられる：

A_2　Select ロジックが I_p を選択する。

B_1　grant[p] がアサートされ、I_p に対する issue と同時に、I_c に対する wakeup が行われる。RAM 部から読み出された tagD[p] は、wakeup のために CAM 部に入力され、rdyL/R[c] がセットされる。

B_2　request[c] がアサートされ、I_c も選択の対象となる。実際に I_c が選択されると、C_1 では grant[c] がアサートされることになる。

ロジックのパイプライン化

命令スケジューリングの 5 つのフェーズのうち、rename、dispatch、および、issue は、必要であれば、パイプライン化を施すことによってシステムのクリティカル・パスから外すことができる。パイプライン化の代償は、分岐予測ミス・ペナルティの増加であり、通常受け入れられる。実際 現存するスーパースケーラでは、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。

一方、wakeup と select は、実際上パイプライン化できない。図 2 (L-2) に、wakeup に 1 サイクル余分にかけた場合の命令パイプラインの様子を示す。Select から wakeup へのフィードバックにより、(A_2) I_p に対する select が終わった後にしか、(B_1) I_c に対する wakeup は開始できない。その結果、I_c の発行は 1 サイクル遅れ、I_p と back-to-back に実行できなくなる。

同図では、I_p が生成したデータは、オペランド・バイパスを通る必要はなく、レジスタ・ファイルを介して I_c の実行に使用される。すなわち、wakeup と select に 1 サイクルより多くを割り当てることは、IPC の観点からは、レイテンシが 1 である演算器– 通常の構成では ALU からのオペランド・バイパスを取り除くことと等価である。5 章で述べるが、それによる IPC の悪化は最大 15%程度にもなり、動作周波数の向上に見合わない可能性が高い。したがって、レイテンシが 1 であるパスに関しては、wakeup と select からなるループ一周の遅延は 1 サイクル以内でなければならないと結論づけられる。逆に言えば、このループ一周の遅延がクリティカルである場合には、それがシステムの動作周波数を決定することになる。

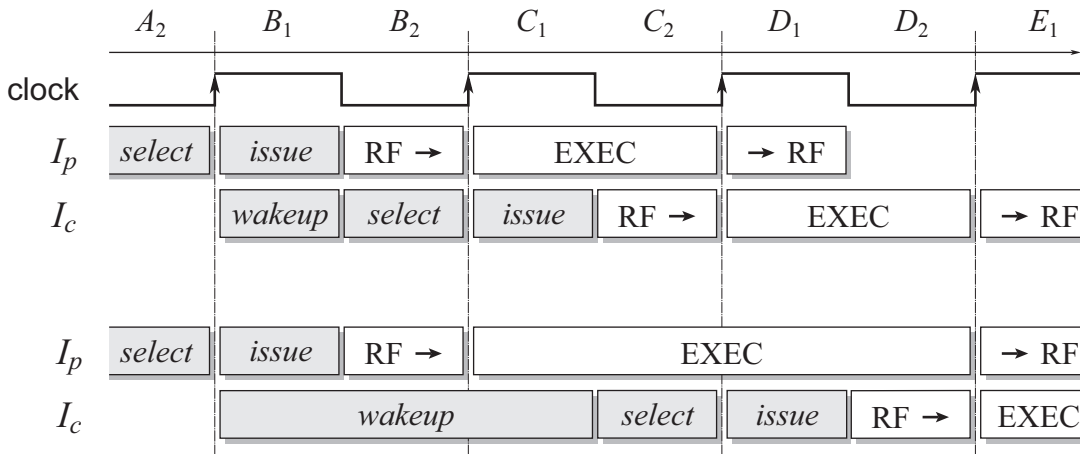


図 3: レイテンシによる wakeup と select のパイプライン化

2.2.3 命令ウィンドウの分散化

前述したように、命令ウィンドウは、集中化された単一のロジックとして実装されるのではなく、演算器のクラスごとに設けられたリザベーション・ステーションや命令キューとして、分散実装されることが多い。ロジックの遅延を考えるにあたってはこの分散化が重要である。このような分散化は、IPC に対する影響も小さいうえに、遅延を大幅に短縮できるからである。分散化には、(1) 構成要素のパラメータの縮小と、(2) パスのレイテンシに合わせた最適化、の 2 つの効果がある。以下、それぞれについて説明する。

(1) パラメータの縮小

複数の命令キューに分散化した場合にも、各キューのロジックの構成自体は図 1 に示したものと基本的には変わらない。キューの命令発行幅を IWq、サイズを WSq とすると、図中の IW,WS を IWq、WSq に読み替えればよい。ただし、CAM の比較入力ポート数だけは、IW から IWq にすることはできない。RAM から読み出された tagD は、別のキューの CAM にも送る必要がある。逆に、各キューの CAM の比較入力ポート数は、別のキューからの tagD を受け入れるため、IWq とするのではなく、IW のままとするのが理想である。実際には、IWq よりやや大きい数に制限し、IPC とのトレードオフとすることが多い。

(2) パスのレイテンシに合わせた最適化

前述したように、レイテンシ 1 のパスでは wakeup と select は合わせて 1 サイクルで実行する必要がある。逆に、これ以外のパスでは、図 3 の下のように、wakeup と select を適当にパイプライン化してもよい。この例では、命令のレイ

テンシが 2 であり、wakeup に 1.5 サイクルをかけている。このような場合は、クリティカル・パスについて議論からは除外してもよい。

たとえば、MIPS R10000[3] は、整数演算、ロード/ストア (**LS**)、浮動小数点 (**FP**) 演算のそれぞれに命令キューを用意している。レイテンシ 1 のパスは (1) 整数から整数、(2) 整数から LS の 2 つである。したがって、クリティカル・パスについての議論では、特に整数キューのみを考慮すればよい。R10000 の整数キューの wakeup と select ロジックも、基本的にはこれまで述べてきたとおりである。そのパラメータは、(IWq, IW, WSq, TW) = (2,4,16,6) である。TW はタグのビット幅である。ただし、ソース・オペランドは、L/R の 2 つではなく、A、B、C の 3 つある。A、B に対しては、自身から 2 つ、ロード命令から 1 つの、C に対しては、それらに加えて、FP 比較命令から 1 つの tagD を受け入れる。したがって、RAM は 2-read 2-write、6b×16word、CAM のキー部は 2-write、6b×48word である。CAM の比較入力ポート数は 3 ~ 4、比較器の数は 160 個にものぼる。

従来の wakeup では、このような RAM と CAM を約 0.5 サイクルの間に逐次的にアクセスしなければならない。詳細は 5.2節で述べるが、この程度の IWq、WSq でも、その遅延はかなりクリティカルである。

2.2.4 Wakeup の定義

次章からは、行列を用いた 2 つの方式について述べる。両方式の比較のためには、wakeup の処理を正確に定義する必要がある。

Wakeup を、連想方式の CAM 部だけと定義している文献も多い [4, 2, 5, 6] が、本稿では、本章で述べてきた通り、RAM 部と CAM 部、を wakeup と定義している。

本稿の定義は、以下の理由により、より妥当である ; システムの動作周波数を決定する上で重要であるのは、前述のループ一周の遅延である。そのうち select の処理は wakeup の方式とはほぼ独立であるが、残りの部分、すなわち、RAM 部と CAM 部の形式は、後述するように、互いに強く依存しており、どちらか一方だけを取り上げることは意味がない。また、5 章で述べるように、RAM 部の遅延は CAM 部の遅延の 1/2 程度もあり、決して無視することはできない。

以上の理由から、wakeup は、IW,WS の増大、パイプラインの深化、LSI の微細化にともなって、スーパースケーラのクロック速度を制限する主要因の 1

つとなると考えられるのである。次章以降では述べる依存行列による方法は、この wakeup を置き換えるものである。

2.3 関連研究

スコアボーディングと Tomasulo アルゴリズム

Out-of-Order 命令スケジューリングの歴史は、スコアボーディング [7] と **Tomasulo アルゴリズム** [8] まで遡ることができる。それ以来 wakeup は、基本的には、タグによる連想検索によって行われてきた。ただし、2.1 節で触れたように、タグの選び方にはいくつかのバリエーションがある。タグは個々のデータに付される ID であるから、バックエンドに同時に存在するデータを一意に特定できるものであれば、その役割を果たすことができる。タグという術語を最初に用いた Tomasulo アルゴリズムでは、基本的には、各命令が格納されたリザベーション・ステーションのエントリの番号をタグとしている。Control Data CDC 6600 のスコアボーディング [7] では、出力依存を解消しないため、データを生成する機能ユニットの番号をタグとして用いることができた。投機的実行を行う最近のスーパースケーラでは、投機的な実行結果を格納するバッファのエントリの ID をタグとして用いることが多い。90 年代後半に入ると、Out-of-Order 命令スケジューリング・ロジックの複雑さがクロック速度に与える影響が懸念されるようになり、複雑を軽減する研究が次第に注目されるようになってきた。たとえば Henry らは、回路上の工夫によって、大規模な命令ウィンドウを検討している [9]。一方、マイクロアーキテクチャ・レベルの改良としては、連想検索の対象を制限したり、連想検索をクリティカル・パスから外すアプローチと、連想検索を完全に省略するアプローチがある。

連想検索の対象を制限する方法

Palacharla らは、**Dependence-Based 命令ウィンドウ**を提案している [2]。この方式では、命令ウィンドウを、演算器クラスタに対応して、複数の小規模な FIFO に分散化する。依存関係にある命令は同じ FIFO に入れられるので、wakeup される命令は各 FIFO の先頭に限定することができる。この方式の潜在的な問題点は各 FIFO への命令の分配 (steering) の方法にある。分配はヒューリスティクスに基づいて行われ [2, 10, 11] るため、性能はその精度に依存する。あまり複雑なヒューリスティクスを用いると、クロック速度に対して悪影響を与えか

ねない。

連想検索をクリティカル・パスから外す方法

Canal らは、**First-Use** 法と、**Distance** 法を提案している [12]。First-Use 法では、命令ウィンドウは、データの揃っている命令が格納される Ready Queue と、物理レジスタの各エントリを参照する最も古い命令が登録されている First-Use Table から構成される。命令が実行されると First-Use Table が参照され、登録されている命令が Ready Queue へと移される。この方式は、連想検索を不要としているが、従来方式に対する IPC の低下が大きい。性能低下を補うためには、I-Buffer と呼ぶ連想メモリを必要としている。そのため、連想検索が削除されているとは言い難い。加えて Ready Queue ではポインタをたどって該当する命令を検索するので、プロセッサの動作速度に対する影響も大きいと考えられる。Distance 法の命令ウィンドウは、データが利用可能になる時刻を管理する Register Availabel Table、実行レイテンシが未定の命令を蓄えている Wait Queue、命令を VLIW 形式にして保持している Issue Queue から構成される。命令は Issue Queue から発行されるので、連想検索を必要としない。しかし、Wait Queue は、連想メモリで構成されているため、やはり連想検索を取り除いているとは言い難い。

Dualflow

連想検索が必要となる本質的な原因は、命令間の依存関係が暗示的に与えられるためである。したがって、連想検索を排除する鍵は、命令間の依存関係を明示的に示すことにある。**Dualflow** [13, 14] は、制御駆動とデータ駆動を融合した命令セット・アーキテクチャである。Dualflow は、命令セット・アーキテクチャのレベルから命令間の依存関係を明示的に示すことによって、マイクロアーキテクチャでの連想検索を省略することを目的に考案された。通常の制御駆動アーキテクチャがレジスタを介して間接的にデータの授受を行うのに対して、Dualflow は、レジスタを定義せず、生産側の命令が消費側の命令を明示的に指定することで直接的にデータの授受を行う。消費側の命令は、生産側の命令のフィールドに埋め込まれた数個のポインタによって指定される。Dualflow の初期の研究では、各命令に埋め込まれた消費側命令へのポインタをほぼそのままの形で保持する、ポインタ形式の命令ウィンドウを仮定していた [13]。ポインタ形式では、従来のスーパースケーラ では CAM で構成されるテーブル rdyL/R

を RAM に置き換えることができる。wakeup では、まず RAM からポインタを読み出し、それをアドレスとして rdyL/R を格納する RAM にアクセスする。Dualflow は、しかし、命令セット・アーキテクチャの制限が強すぎるため、無用な命令によってコード・サイズが爆発するという問題点があった。この問題のため、Dualflow それ自体は成功しなかったが、その後の研究に影響を与え、依存行列による方式を生み出すこととなった。

EDF 命令ウィンドウ

佐藤らは、本稿で述べるのと同様に、スーパースケーラ のフロントエンドにおいて命令間の依存関係を明示的な形式に変換しておく方法を提案している [15]。彼らが用いる **EDF (explicit data forwarding)** 命令ウィンドウは、Dualflow の初期のポインタ形式と基本的に同じものである。ポインタ形式には、しかし、以下の問題点がある：

1. メリットが小さい。rdyL/R を格納するテーブルのアクセス部分が、CAM の比較器から、RAM の行デコーダに置き換わるのだが、その遅延の差はそれほど大きくはない。

2. 保持できるポインタの数の制限が、IPC とハードウェアの複雑さの間にトレードオフを生じる。ポインタの数に対する制限は、Dualflow ではコンパイラによって解決されるのに対して、ではハードウェアによって解決されなければならない。そのため現状では、IPC とハードウェアの複雑さの間に厳しいトレードオフが生じている。特に、投機失敗時の状態回復処理が難しくなる。

依存行列

ポインタ形式においては、読み出されたポインタで rdyL/R を格納する RAM にアクセスする際に、その RAM の行デコーダにおいてポインタのデコードが行われる。このデコード処理をフロントエンドで済ましておくという着想から、依存行列 は元々Dualflow 向けのロジックとして考案された [16]。前述したポインタ形式の 2 つの問題点は、依存行列 ではほぼ完全に解消される。上述したの既存方式に対する提案方式の定性的な優位点は、以下のようにまとめられる：

1. 構造がシンプルである。提案方式では、通常の RAM とほとんど変わらない依存行列のみによってを実現することができる。

2. IPC に対するペナルティはごく小さい。

- 依存行列 の狭幅化を行わなければ、IPC に対するペナルティはほとんどない。依存行列 の狭幅化は、遅延とペナルティとのトレードオフを導入するが、その関係は厳しくない。
- 投機失敗時の状態回復処理が容易であり、依存行列に関連して新たなペナルティが生じることはない。

これらの優位点は、命令間のデータ依存関係を行列で表すという表現方法そのものによるものであると考えられる。

また現在、当研究室では、依存行列と DEC Alpha 21264 のクラスタリングを組み合わせた場合の研究が進められている。クラスタリングのペナルティと依存行列のペナルティが隠蔽し合うことによる結果が期待されている。

第 3 章　依存行列による方式

依存行列による方式は、前章で述べた従来方式とは根本的に異なり、タグを用いずに wakeup を行う。提案する方式では、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 が、スケジューリングにおける中心的な役割を果たす。また DEC alpha 21264 で採用されている方式では、命令が参照する物理レジスタの利用可能性を表すベクトル **prrdy** を介して、生産者から prrdy へ、prrdy から消費者への依存を表す 2 種類の依存行列を用いる。前者を直接方式、後者を間接方式と呼ぶことにし、3.1 節で間接方式、3.2 節で直接方式についてそれぞれ述べる。3.3 節では、依存行列を用いた場合の、投機実行との関係について述べる。

3.1　間接方式

間接方式は、grant から request を求めるにあたって、割り当てられた物理レジスタが利用可能かどうかを表すベクトル prrdy を経由する。grant から prrdy、prrdy から request を求めるには、それぞれ、IR、RI の 2 つの行列を用いる。

3.1.1　間接方式の原理

IR/RI は、それぞれ、命令とそのデスティネーション/ソースとなる物理レジスタとの関係を表す。したがって、物理レジスタの数を NR とすると、それぞれ、WS 行 $\times NR$ 列である。理由は後述するが、$NR = 2 \cdot WS$ 程度とすることが普通である。

図 4 に、間接方式の概念図を示す。同図は、2.2.1 節と同様に、ウィンドウ・エントリ l/r 番 に格納された命令 I_l/I_r が生産する結果を、物理レジスタ L/R 番 を介して、ウィンドウ・エントリ c 番 に格納された命令 I_c が消費する場合を示す。行列 IR の l/r 行では、それぞれ、L/R 列の 1 要素のみが 1、それ以外は 0 となっている。一方、行列 RI の c 行では、L/R 列の 2 要素が 1、それ以外は 0 となっている。

命令の発行が許諾されるとまず、行列 IR に従って、grant から prrdy を求める。発行が許諾された命令に対応するすべての行を列ごとに OR した得られる行ベクトルが、セットすべき prrdy を表す。図 4 では、IR の l/r 行によって、prrdy$[L/R]$ がそれぞれセットされる。

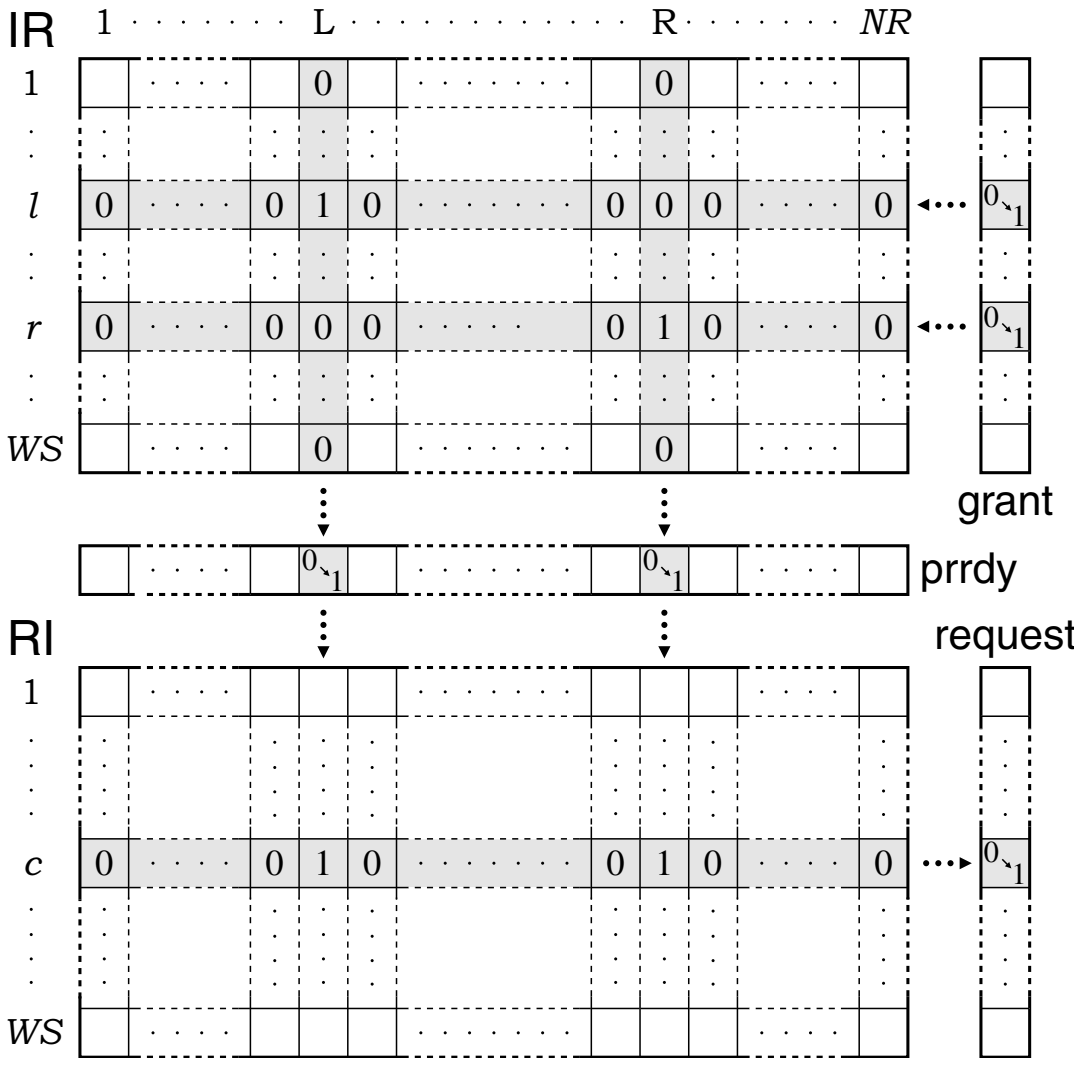


図 4: 間接方式の概念図

次いで、行列 RI に従って、prrdy から request を求める。c 行では L/R 列が 1 であるので、物理レジスタ L/R 番 が共に利用可能であれば、I_c が発行可能である。すなわち、prrdy$[L/R]$ が共に 1 であれば、request$[c]$ を 1 とする。

3.1.2 間接方式のロジック

図 5 に、間接方式のロジックを示す。ロジックは、図 4 と 1 対 1 に対応するアレイによって実装される。

IR

IR に対応する部分は、NR b$\times WS$ word、IW-write、1-read の RAM の読み出しによって実現される。ただし、連想方式の RAM 部と同様に、行デコーダは必要なく、grant が直接ワードラインに接続される。

また、通常の RAM とは異なり、発行が許諾される命令に対応して複数のワードラインが同時にアサートされる。そのため読み出しポートは、必然的にシングル・ビットラインとなる。ダブル・ビットラインとしても、1 であるセルがある行では両方のビットラインがプルダウンされてしまい、いずれ差動出力を得ることはできないからである。そのため、シングル・エンドのセンスアンプを用いる必要がある。

RI

RI は、各行に対応する WS 個の wired-AND として実現される。request は、予め high にプリチャージされ、接続された NR 個のプルダウン・スタックによってプルダウンされる。図 5 の場合、c 行 X 列のセルの値は 0 であるため、$\overline{\text{prrdy}[X]}$ の値に関わらずプルダウン・スタックは ON にならず、request の値に影響を及ぼさない。L/R 列では、セルの値が 1 であるため、$\overline{\text{prrdy}[L/R]}$ のどちらか一方が high であると request$[c]$ はプルダウンされる。逆に、$\overline{\text{prrdy}[L/R]}$ が共に low であれば、request$[c]$ は high に保たれる。

なお、図 5 の例では、IR と RI はまったく同じ構造のセルを用いて実現されている。

prrdy

RI において、request$[c]$ が high に保たれるには、$\overline{\text{prrdy}[L]}$と$\overline{\text{prrdy}[R]}$が同時にアサートされる必要がある。一方で、grant$[l]$ と grant$[r]$ は、一般には異なるサイクルでアサートされる。そのため prrdy の位置には、単なるパイプライン・ラッチではなく、1b のレジスタを置く必要がある。

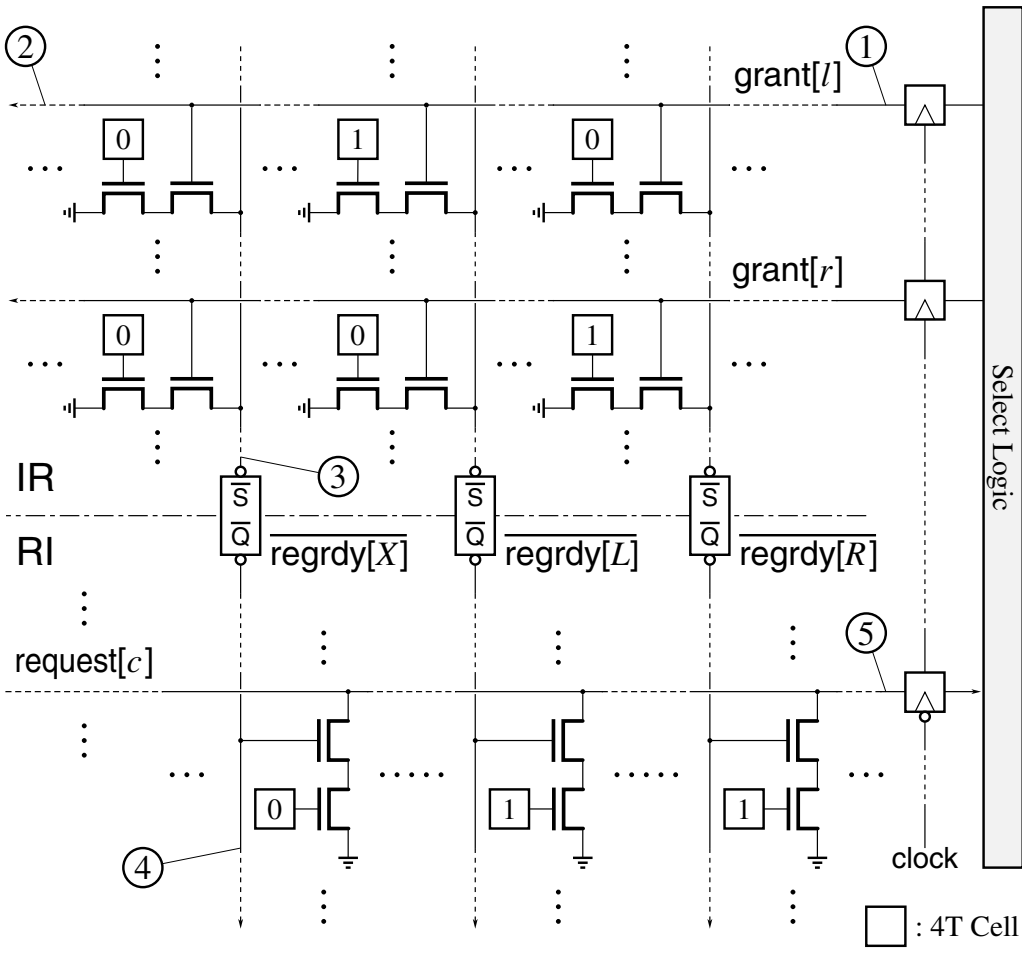


図 5: 間接方式のロジック

その一方で、RI と select ロジックの間には、連想方式における rdyL/R (図 1) のようなレジスタは必要なく、パイプライン・ラッチを置けばよい。

3.1.3 連想方式との関係

間接方式における IR/RI の各行は、基本的には、前章で述べた連想方式における物理レジスタ番号、すなわち、タグをデコードしたものである。IR/RI は、連想方式の RAM 部/CAM 部と 1 対 1 に対応している。

連想方式では、IW 個のタグに対する連想処理を、それぞれ IW 個の部分回路で行っていた。一方間接方式では、タグをデコードすることによって、1 個の回路でまとめて処理している。間接方式のロジックは、連想方式のロジックに対して、以下のように簡単化されている：

RAM 部と IR 読み出しポート数が IW 本から 1 本に削減される。

CAM 部と RI　一致比較が単純な和積に変わると共に、入力ポート数が IW 本から 1 本に削減される。

しかしその一方で、ビット幅は $\lceil \log_2 NR \rceil$ b から NR b へと大幅に増加している。これら得失が遅延に与える影響については、5 章で詳しく述べる。

3.2　直接方式

3.2.1　直接方式 の概要

図 6 に、スケジューリング・ロジックにおける依存行列 の位置を示す。図 1 に示した従来方式のロジックと比較されたい。提案方式では、wakeup、すなわち、従来方式における RAM と、CAM のキー部が、依存行列 に置き換わる。依存行列 は、select ロジックからの発行許諾信号 (grant) を受けて rdyL/R の入力を生成する。また、間接方式が IR、RI の 2 種類を逐次的にアクセスしていたのに対して、直接方式では直接からを求める。

依存行列を用いた方式では、wakeup にはタグを用いない。タグ[1)]が命令の実行結果を格納するバッファのエントリ ID である場合には、従来方式と同様タグを読み出す必要があるが、それは issue に含まれ、wakeup には含まれない (図 6)。前述したように、issue は、wakeup とは異なり、パイプライン化可能である。

依存行列 の定義

図 7 に、依存行列 の概念図を示す。ここでいう依存行列は、命令間のデータ依存関係を表す WS x WS の行列で (対角要素は使用しない)、左/右のソース・オペランドに対して 1 つずつ用意される。エントリ ID $= p$ の命令が生産するデータを、ID $= c$ の命令がその左 (右) オペランドとして消費する場合、左 (右) オペランド用の行列の c 行 p 列の要素は '1'、それ以外は '0' とする。

図 7 に示す例では、連続する 4 つの命令が ID $= 1$ のエントリから順に格納された場合を表している。この場合、たとえば、ID $= 2$ の命令が左オペランドとして消費する\$1 を生産するのは ID $= 1$ の命令であるから、左の行列の 2 行 1 列の要素が '1' となる。そのほかの要素も同様に求められる。

[1)] この場合、タグというネーミングはもはや適当ではないが、説明の連続性のためあえてそのままとする。

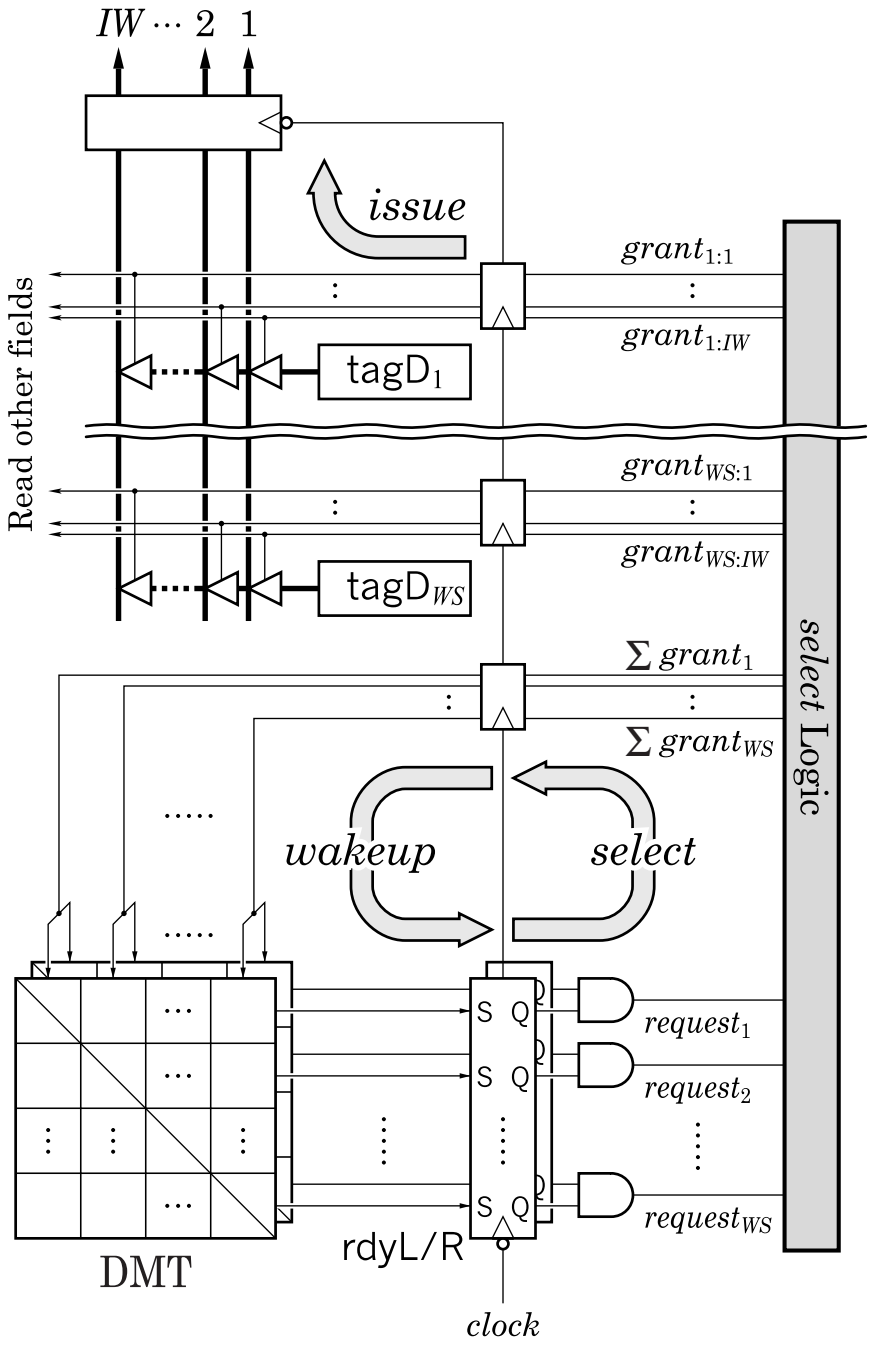


図 6: スケジューリング・ロジックにおける依存行列 の位置づけ

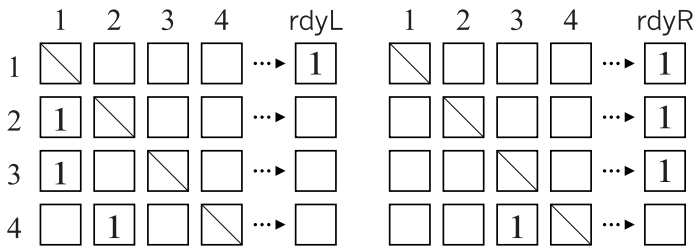


図 7: 依存行列

依存行列 による **wakeup**

wakeup においては、発行される IW 個の命令に対応する IW 行 の bitwise-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。

図 7 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。依存行列 の第 1 行 は、定義により、ID = 1 の命令が実行されるときにセットすべき 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 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 2 つの命令が選択され同時に実行される場合には、第 2、3 行の bitwise-OR を求めればよい。すると、ID = 4 の rdyL と rdyR をセットすればよいことが分かる。ただし実際には、bitwise-OR を求めるといっても、各行で '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命令はただ 1 つだからである。

次節からは、依存行列 のロジックとその動作について詳細に説明する。依存行列 の動作としては、wakeup、更新のほかに、投機の失敗時の状態回復がある。3.2.2 節ではロジックの構成を示し、wakeup 時の動作を説明する。3.2.3 節では更新時の動作、3.3 節では投機失敗時の状態回復について述べる。

3.2.2 依存行列 の実装

前述したように、依存行列 の読み出しでは、原理的には複数行 の bitwise-OR を求める必要がある。しかし、そのための特別なロジックは必要なく、通常の 1-read の RAM をほぼそのまま用いることができる。

ただし、通常の RAM とは、以下の構造上と動作上の相違点がある：

構造上の相違点　書き込み側とは、行と列の関係が逆になっている。すなわち、ワード線 issue は縦方向に、ビット線$\overline{\text{rdyL}}$ は横方向に配線されている。

動作上の相違点　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、依存行列 では毎サイクル実行される最大 IW 個の命令に対応する最大 IW 本のワード線 issue が同時にアサートされる。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、その命令に対応する列の bitwise-OR を読み出すことができるのである。

なお、前述したように、各列行で “1” である要素ははたかだか 1 つであるから、各ビット線 $\overline{\text{rdyL}}$ の pull-down はたかだか 1 つのセルによって行われる。

なお依存行列 では、select ロジックからの入力信号が若干異なる。従来方式の wakeup、および、両方式の issue では、ポートを選択するため、select ロジックからは各命令が何番目に選ばれたかを示す信号が必要であった。一方依存行列 には、何番目に選ばれたかという信号は必要なく、選ばれたかどうかを表す信号が必要である。select の構成によっては、前者の OR をとる必要がある場合がある。

3.2.3 依存行列 の更新

次に、依存行列 の更新処理について述べる。依存行列 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、以下で rename について述べる。

rename

rename は、前述したように、tagD を割り当てる処理と、対応する tagL/R を求める処理からなる。割り当てるべき tagD は、たとえば直前のサイクルで求めておくことができるから、性能上重要であるのは tagL/R を求める処理である。

tagL/R を求めるには、論理レジスタ番号からタグへの写像を保持するレジスタ・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。

各命令に割り当てられた tagD は、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 RMT に書き込まれる。tagL/R 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得ることができる。

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、依存検出器が必要となる。依存検出器は、同時にデコードされる命令間で論理レジスタ番号の比較を行う一致比較器のアレイである。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。

依存行列 の更新

次に、依存行列 の更新処理について述べる。今デコードされているIcがID = cのエントリに格納されるとしよう。依存行列 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) Icの依存元の命令I_l/I_rが格納されているエントリのID = pを求める、(2) pをデコードしたものを依存行列 のc行に書き込む、の2つである。

(1) pは、renameとまったく同じ方法によって求めることができる。すなわち、上述のrenameの処理の説明において、『タグ』を『I_l/I_rのエントリID』と、『RMT』を『生産者テーブル (producer table:**PT**)』と読み換えればよい；ただしPTは、論理アドレス番号をアドレスとし、I_l/I_rのエントリIDを内容とする、2・IW-read IW-writeのRAMである。

依存行列 の更新処理においても、renameと同様の依存検出器が必要となるが、これはrename用のもの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。

3.2.4 更新処理の遅延

次に、更新処理の遅延について考察する。依存行列 の更新はrenameと同時に行われる。その遅延がクリティカルとなる可能性があるのは、(1) PTの遅延と、(2)pのデコード であるが、以下のように、双方ともrenameに比べて短いと考えられる：

1. バッファのエントリ数はWSの1 ~ 2倍程度とすることが普通である[1)]から、RMTの内容であるタグは、PTの内容であるエントリIDより0 ~ 1b程度長い。そのほかのパラメータは両者で同じであるから、PTの遅延はRMTの遅延と同じかより短い。

[1)] その不足によって命令フェッチが滞ることを防ぐには、バッファは最低 WS エントリ必要である。リネーミングのオーバヘッドのため、それでは不十分な場合がある。

2. 依存行列 への書き込みは、rename で得られた tagL/R 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よって決まる。p 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、提案方式における依存行列 の更新処理がクロック速度を低下させる可能性は低いといえる。

3.3 依存行列 と投機

最近のスーパースケーラでは、分岐予測による投機的実行が必須となっている。命令スケジューリングの方式にとっては、投機失敗時の処理が容易であることが重要である。

依存行列 における分岐予測ミス時の処理はごく簡単である。分岐予測ミス時には、rdyL/R を含む、無効化される命令に対応する行列をすべて '0' にリセットすればよい。図 7 の例において ID = 2 の分岐命令が予測ミスを起こした場合には、ID = 3 の命令が無効化される。依存行列 では、3 ~ 6 行列をリセットすればよい。

行列のリセットは命令ウィンドウのほかの情報の無効化と同程度のコストで実行できるから、投機失敗時に依存行列に関連して新たなペナルティが生じることはない。

第 4 章　依存行列の改良

4.1　直接方式における依存行列の単一化

行列の単一化について理解するためには、ウィンドウ・エントリと物理レジスタの寿命の違いについて理解しておく必要がある。そこで 4.1.1項でまず、それらについてまとめた後、4.1.2項で行列の単一化について述べる。

4.1.1　ウィンドウ・エントリと物理レジスタの寿命

ある命令に対して、それが格納されるウィンドウ・エントリと、その命令のデスティネーションとして割り当てられる物理レジスタは、異なる寿命を持つ。ウィンドウ・エントリと物理レジスタは、フェッチされた命令に対してほぼ同時に割り当てられるが、開放のタイミングが異なる。投機失敗時の回復の方式などにも依存するが [5]、基本的には、それぞれ以下のタイミングで開放される：

ウィンドウ・エントリ　命令の実行が終了すれば開放してよい。

物理レジスタ　命令の実行が終了した時点ではなく、以下のように、当該物理レジスタを参照する命令がウィンドウ内に存在しなくなるまで解放できない：

リオーダ・バッファ方式　当該エントリがバッファからリタイアする時まで

レジスタ・リネーミング方式　論理レジスタを上書きする次の命令が完了する時まで [3]

物理レジスタは、リオーダ・バッファ方式の場合、当該エントリがバッファからリタイアした時まで；レジスタ・リネーミング方式の場合、論理レジスタを上書きする次の命令が完了した時 [3] まで、開放することができない。

レジスタ・リネーミング方式における物理レジスタの解放のタイミング
以下に示す命令列を考えよう；ただし、L=P は、論理レジスタ L に物理レジスタ P が割り当てられている状態を示す：例として、PP は、論理レジスタ L に物理レジスタ P を割り当てる動作をする命令、CP1、CP2 はどちらも論理レジスタ L に割り当てられている物理レジスタ P を参照するような命令である。

```
PP  :     -> L=P
CP1 : L=P ->
CP2 : L=P ->
```

```
PQ  :      -> L=Q
CQ1 : L=Q ->
```

この場合、正確には、P を参照するすべての命令 CPx の実行が完了すればP は不要になる。

しかし実際には、そのこと自体を直接検出することは困難である。

そこで、L を再定義する PQ の実行の完了をもって、P が不要と判断する [3]。PQ の実行が完了すれば、PP が定義した古い論理レジスタ L の値が参照されることは最早なく、P は不要と判断できる。

この寿命の違いは、以下のように、ウィンドウ・エントリと物理レジスタは個別に管理されるべきであることを意味する。

ウィンドウ・エントリは、サイクル・タイムとのトレードオフの観点から言って、物理レジスタに比べ高価な資源である。ウィンドウ・エントリの数、すなわち、WS は、wakeup、select の遅延に直接影響するからである。

仮に、ウィンドウ・エントリと物理レジスタを 1 対 1 に対応させ、一つの機構によって一括管理するとすると、ウィンドウ・エントリにも物理レジスタの寿命が強制されることになる。すなわち、対応する物理レジスタの寿命が尽きるまで、物理レジスタより高価なウィンドウ・エントリまで再利用できなくなってしまう。

以上の理由から、ウィンドウ・エントリと物理レジスタを一括管理することは受け入れ難い。実際、現存するプロセッサでは、その管理コストにも関わらず、ウィンドウ・エントリと物理レジスタを個別に管理している。物理レジスタの数 NR は、$NR = 2 \cdot WS$ とすることが多い [3, 17]。

4.1.2 行列の単一化

さきほどの定義では、ソースごとに 1 つの行列を必要とした。 図?? の例では、左/右 2 つの行列が用いられている。一方間接方式では、ソースの数に関わらず、1 組の行列で済む。ソースの数は、RI の各行の 1 の数として表われる。図 4、図 5 の例では、物理レジスタ L/R 番 に対応して L/R 列の 2 要素がセットされている。そこで直接方式でも、間接方式と同様に、1 つの行列への単一化を試みる。

行列を単一化するには、基本的には、各行列を要素ごとに OR すればよい。図 7 の例では、単一化された行列の第 4 行では、第 2、3 列の 2 要素が 1 となる。

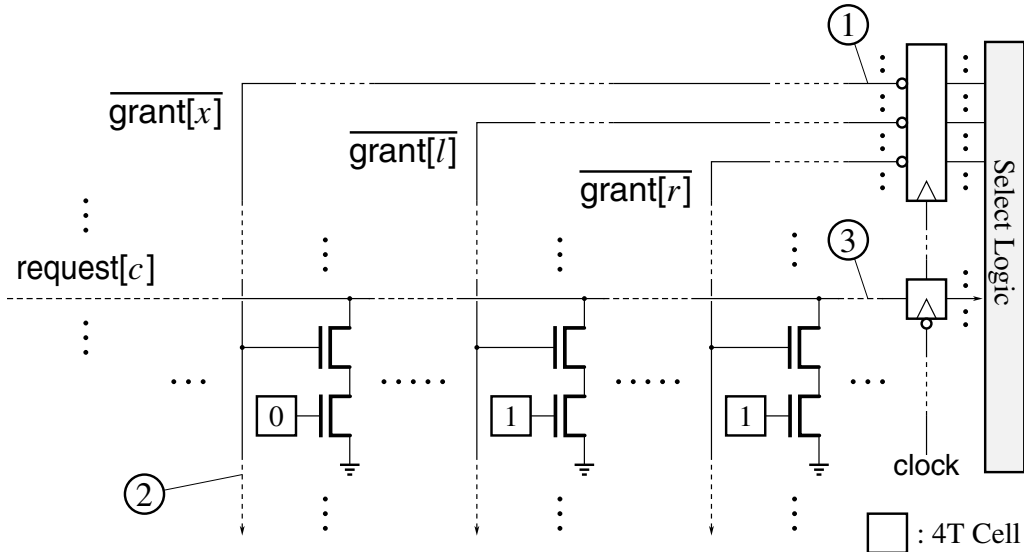


図 8: 直接方式のロジック

単一化された行列のロジックは、図 8 のようになる。図 5 に示した間接方式の RI 部と基本的には同じだが、アレイの幅は NR から WS へと大幅に削減される。削減の程度については、次節で詳しく述べる。

ただし、RI の場合と同様に、grant が異なるサイクルでアサートされることへの対処が必要である。それには、2 つの対処法が考えられる。1 つ目は、やはり RI と同様に、行列の前にレジスタを置くことである。しかしこの方法は、直接方式ではうまく行かない。以下、1 つ目の対処法がうまく行かない理由と、2 つ目の対処法について順に述べる。

対処法 **1**

間接方式の prrdy と同様、行列の前にレジスタ irdy を置くことにしよう。

その役割を果たすためには、irdy も、prrdy と同様に、それに依存する命令がウィンドウ内に存在している間セットされていなければならない。すなわち irdy は、前節の説明に従えば、ウィンドウ・エントリより長い、物理レジスタと同様の寿命を持つとすることができる[1)]。すなわち、格納された命令 Ilr の実行が終了した後、ウィンドウ・エントリ p 番は原理的には解放してよいが、irdy の p 番は解放できない。

そのため、解放されたウィンドウ・エントリ p 番 も、irdy の p 番が解放されるまで、再利用することはできない。したがって、ウィンドウ・エントリも、

[1)] 正確には、リオーダ・バッファ方式の物理レジスタと同じ寿命を持つ。

irdy と同じ物理レジスタの寿命を強制されることになる。このことは、前節で述べた通り、通常受け入れられない。

対処法 **2**

2 つ目の対処法として、grant された命令に対応する列をリセットすることが考えられる。例えば、grant$[l]$ と grant$[r]$ が、この順序で別のサイクルにアサートされた場合、以下のようになる：

1. grant$[l]$ がアサートされると、l 行のスタックによって request$[c]$ はプルダウンされる。
2. そのプリチャージ・フェーズで l 列をリセットする。以降では、grant$[l]$ に関わらず l 行のプルダウン・スタックは ON にならない。
3. したがって、grant$[r]$ がアサートされる時には、request$[c]$ は high に保たれる。

列をリセットするには、専用のポートを用意するのが最も低コストであろう。各セルに対しては、小型の nMOS ゲートを 1 つ追加するだけでよい。

なおこの方法は、行列上に保存された依存関係を発行時に破壊するため、投機失敗時の状態回復の方法が制限される。しかし、このことに依存しない効率のよい回復法も提案されている [5]。

4.2 依存行列 の高速化

本節では、依存行列 アクセス、特に wakeup のための依存行列の読み出しを高速化する方法について述べる。その 1 つ目は、2.2.3 節で述べたのと同様の命令ウィンドウの分散化である。2 つ目は、依存行列 を狭幅化することにより、そのレイテンシを IPC に対するペナルティに転化する方法である。両者はそれぞれ独立に適用することができるが、組み合わせることでさらに効果を発揮する。したがって、以下、まず 4.2.1 項で分散化について述べた後、4.2.2 項では、分散化されたそれをさらに狭幅化するという流れで説明を行う。

4.2.1 依存行列 の分散化

命令ウィンドウが、それぞれ発行幅 IWq、サイズ WSq の q 本の命令キューに分散化される場合には、依存行列 は q 個の WSq 行 WS 列の部分行列に分割される。R10000 は、整数、LS、FP の 3 つの命令キューを持ち、それらのパ

ラメータはそれぞれ (IWq,WSq) = (2,16) である。この場合依存行列 は、3 個の 16 行 48 列の部分行列に分割される。その様子を図 9 に示す。

従来方式と同様に、提案手法でも、分散化によって、(1) パラメータの縮小と、(2) レイテンシに合わせた最適化、の 2 つの効果を得ることができる。

1 パラメータの縮小

それぞれの部分行列では、書き込みポート数が IW から IWq に削減されるため、セル面積が減少し、読み出し遅延も短くなる。

また select ロジックは、従来手法と同様に、方式 $WSIW$ から 方式 $WS'IW'$ に縮小される。

ただし、方式 $WSIW$ とは、WS 個の命令から IW 個を選択することを表す。

2 レイテンシに合わせた最適化

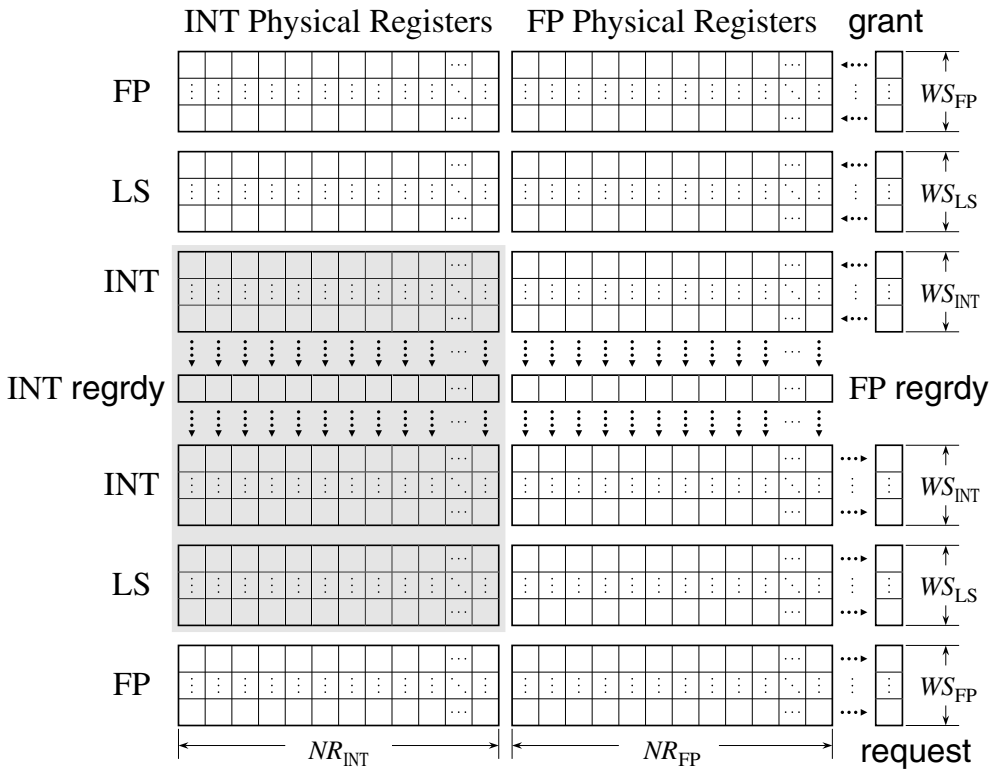
図中 9 個ある WSq 行 WSq 列の小行列は、整数、LS、FP 間のデータの依存を表す。SPARC[18] などのように、整数ー FP レジスタ間転送命令を持たないアーキテクチャでは、図中で破線で示した右上と左下の小行列は不要である。また、MIPS などのように転送命令を持つアーキテクチャであっても、その命令の終了まで後続の命令のフェッチ、デコードを中断することによって、省略することができる。その場合 rdyL/R は、register において '1' に初期化される (2.1 節参照)。いずれの小行列もこの手法によって省略することができるが、上記以外では IPC に与える影響が大きくなってしまうため、適用しない方がよい。

また、前述したように、wakeup と select を合わせて 1 サイクルで実行する必要があるのはレイテンシが 1 のパスだけであり、R10000 の構成では、整数から整数、整数から LS の 2 カ所だけがこの条件を満たす。したがって、図 9 で影を付けた部分の読み出しは select と合わせて 1 サイクルで実行する必要があるが、それ以外の部分の読み出しは適当にパイプライン化してよい。

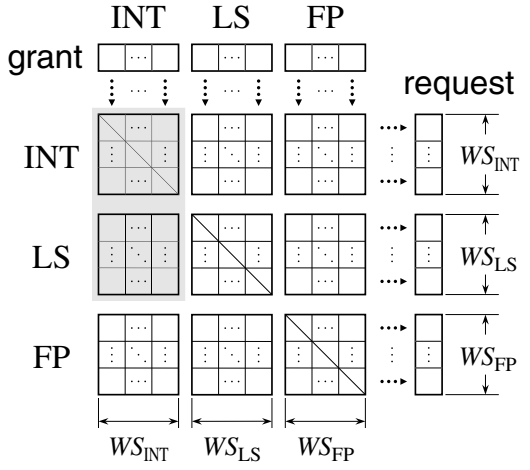
図 9 で影を付けた部分のみを取り出し、これを L-1 依存行列と呼ぶ。元の依存行列 を L-2 依存行列と呼ぶ。

L-1 依存行列は、元の依存行列 に比べて大幅に小型化されるため、読み出しの遅延も短縮される。

L-2 依存行列の読み出しは、対応する演算器のレイテンシに合わせて、select と合わせて 2 サイクル以上かけて実行すればよい。wakeup と select にそれぞれ 0.5 サイクルを充てるとすると、L-2 依存行列の読み出しには 1.5 サイクル、す



(a) Indirect Scheme



(b) Direct Scheme

: L-1 Matrix

図 9: 依存行列 の分散化

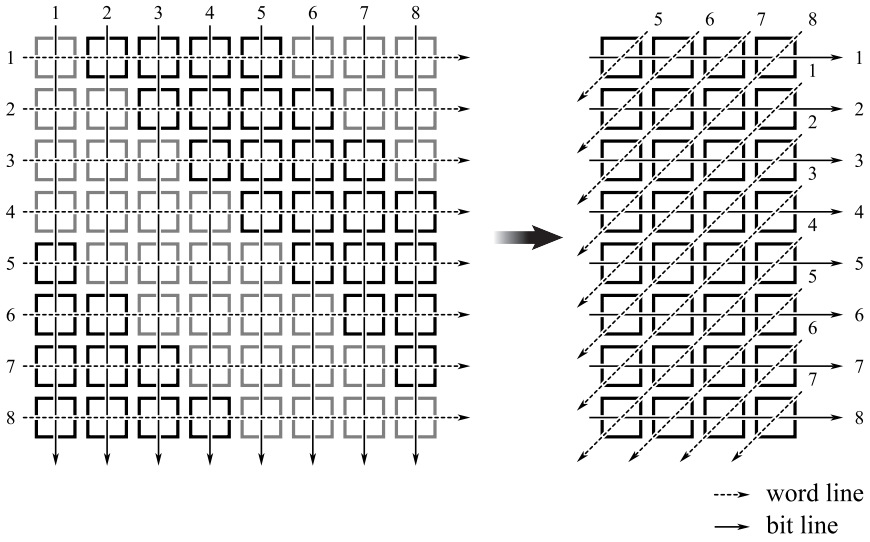


図 10: L-1 依存行列の狭幅化

なわち、L-1 依存行列の読み出しの 3 倍の時間をかけられる。したがって、L-2 依存行列の読み出しがクリティカルになる可能性は非常に低く、wakeup の遅延についての議論から除外することができる。

さらに、L-1 依存行列に対しては、以下に述べる高速化手法を適用することができる。

4.2.2 L-1 依存行列の狭幅化

依存する命令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 [19]。この性質を利用して、L-1 依存行列を狭幅化し、さらにその遅延を短縮することを考える。すなわち、各命令キューにおいて自命令の後の w $(1 \leq w \leq WS' - 1)$ 個の命令に対応するビットだけを L-1 依存行列に残し、そのほかのビットを L-2 依存行列 に移すのである。

図 10 に、WSq = 8、w = 4 の場合の L-1 依存行列の狭幅化の様子を示す。図には、整数から整数の部分のみを示したが、整数から LS の部分も同様である。左は狭幅化前の、右が狭幅化後の L-1 依存行列である。元の L-1 依存行列から削除されるセルは、左図中で薄く示した。必要なセルを矩形領域に集めるにはいくつかの方法が考えられるが、よりクリティカルであるビット線$\overline{\text{rdyL/R}}$ を短縮することを優先して、図 10 右のようにするとよいであろう。w を L-1 依存行列の幅と呼ぶことにする。

図 10 右から分かるように、狭幅化された L-1 依存行列のワード線 issue、ビッ

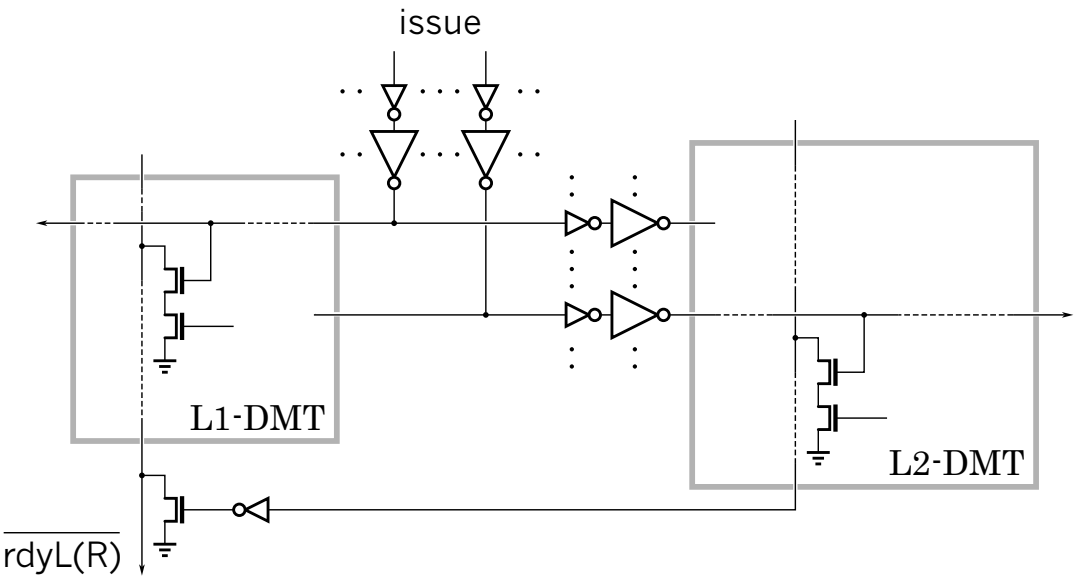


図 11: 依存行列 の 2 階層化

ト線$\overline{\mathrm{rdyL/R}}$ は、それぞれ w 個のセルにしか接続されていない。したがって L-1 依存行列の読み出し、すなわち、wakeup の遅延は、WSq とは独立に、w によって決めることができる。

さて、依存する命令間の距離が w 以下の場合には、狭幅化した L-1 依存行列によって rdyL/R を更新できる。w を超えていた場合には、rdyL/R は L-2 依存行列によって更新される。その場合には、更新が 1 サイクル遅れるため、依存先の命令は依存元の命令の次のサイクルには発行することができなくなる。しかし、依存する命令間の距離は短い場合が多いため、次章で示すように、その影響は十分に小さい。

L1/L2-依存行列間の接続

L-1 依存行列の読み出しでは、L-2 依存行列の影響を最小化することが望ましい。特に、L-1 依存行列を狭幅化した場合には、整数命令に対する発行信号 issue は、L-1/L-2 双方のワード線を駆動する必要があるので、注意する必要がある。図 11 に、両者の接続例を示す。なお、同図中には示していないが、L-2 依存行列のパスの適当な位置にはパイプライン・レジスタが挿入される。同図の構成では、L-2 依存行列が L-1 依存行列の読み出しの遅延に与える影響は以下の 2 点になる：

- L-2 依存行列のワード線ドライバのため、L-1 依存行列のワード線ドライバの負荷が増加する。

- L-2 依存行列の読み出し結果で L-1 依存行列のビット線をプルダウンするため、ビット線の容量が増加する。

いずれも、その影響はごくわずかである。

pace-keeping

狭幅化した L-1 依存行列によって rdyL/R が更新できるのは、実際には、命令間の距離ではなく、命令が格納されたウィンドウのエントリ間の距離が w 以下の場合である。したがって、L-1 依存行列の狭幅化が有効であるためには、命令流上での距離とエントリ間の距離がある程度一致するように制御する必要がある。

同一のキュー内の命令に対しては、エントリをサイクリックに利用することによって比較的容易にこの条件を満たすことができる。しかし異なるキュー間では、それに加えて、キュー間でのエントリの使用位置の制御が必要になる。R10000 の構成では、LS 命令を、依存元の整数命令が格納されるエントリの『横』以降のエントリに格納する必要がある。

この制御自体は容易であり、それがクロック・スピードやデコードの遅延に悪影響を及ぼす可能性は低いが、LS キュー・エントリの利用効率は悪化してしまう。命令間の依存関係を調べることでこの影響は緩和できるが、整数命令が格納されたエントリより『前』の LS キュー・エントリを、盲目的に pace-keeping によって埋めてしまっても、次章で示すように、その影響はごくわずかである。

第 5 章　評価

5.1　IPC の評価

シミュレータ上に行列とその狭幅化を実装することにより、依存行列の導入による IPC へのペナルティーを計測した。

5.1.1　SinpleScalar

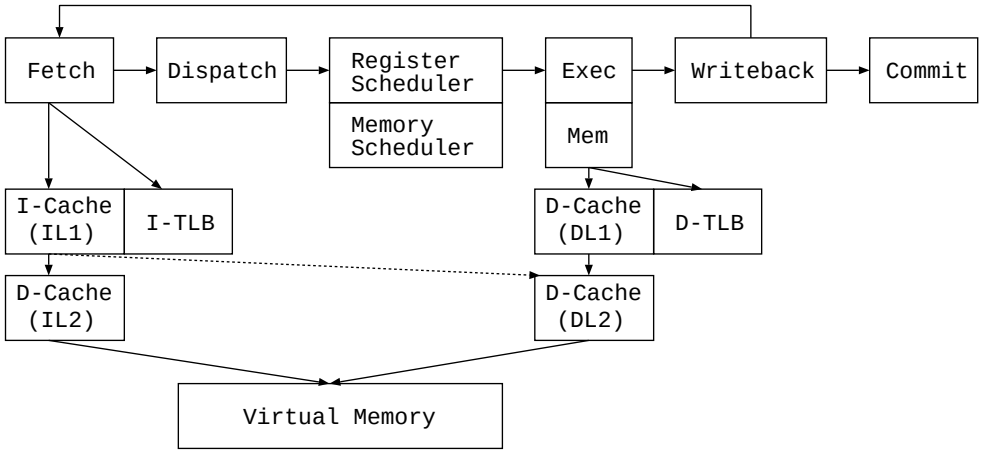


図 12: Simplescalar sim-outorder の構成図

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)[20] 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 [21] を図 12に示す。

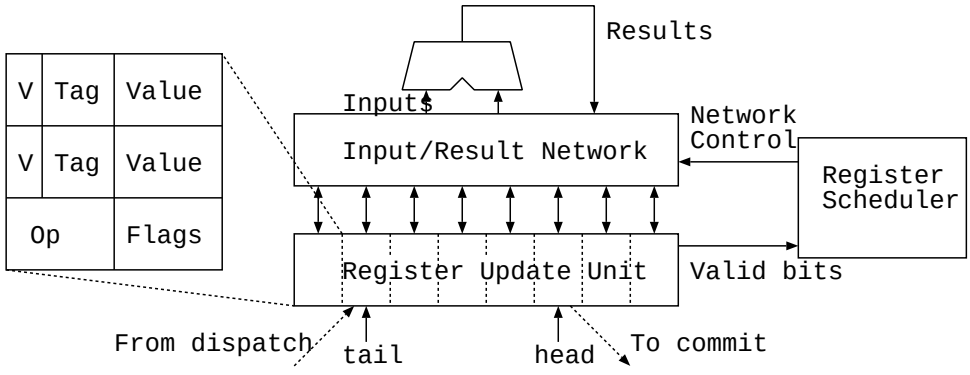


図 13: Register Update Unit(RUU) の概略図

図 13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。

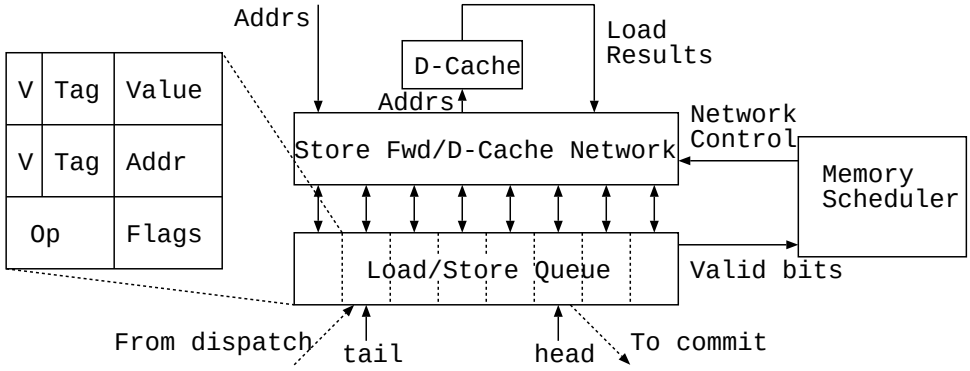


図 14: Load/Store queue(LSQ) の概略図

図 14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。

キューと行列の実装　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 15(a) のように head=tail(=カウンタ)=0 である。図 15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 15(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

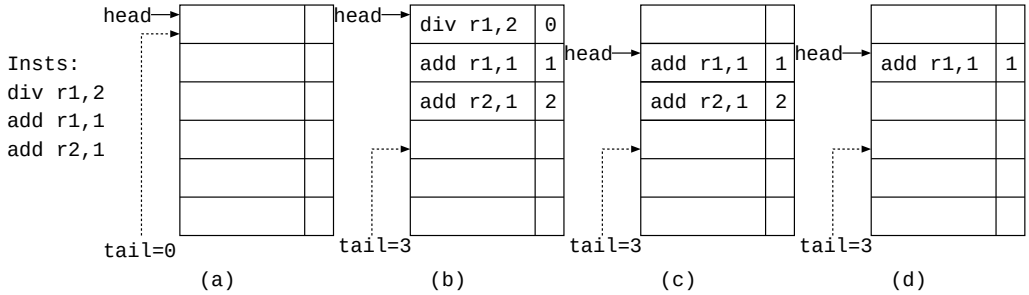


図 15: 命令キューの動作例

15(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 15(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。

pace-keeping を行う場合は、各キューの head が共通のものとなる。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 16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 16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。

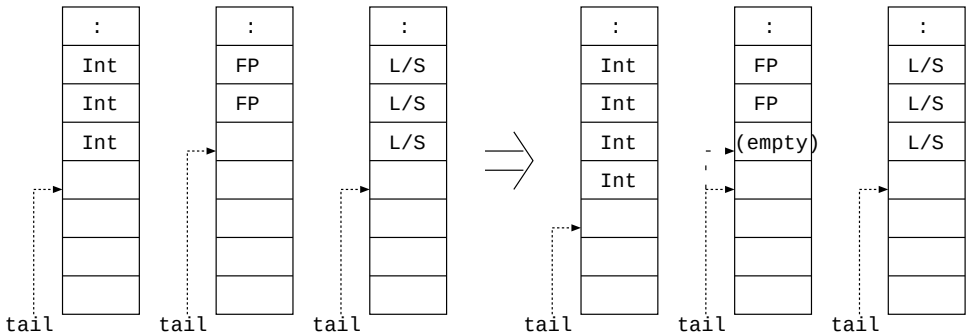


図 16: pace-keeping の様子

表 1: プログラム

program	option	instructions
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

L-1 行列の狭幅化に対しては、以下のようにした。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。

5.1.2 IPC の評価

SimpleScalar ツールセット [21] (ver.2.0) に対して、4.2 節で述べた依存行列 の狭幅化を前節のとおり実装し、表 1 に示す SPEC CINT95 の 8 つのプログラムを用いて、L-1 依存行列 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[3] のマシン構成を用いた。R10000 は、LS、整数演算、FP 演算のそれぞれに命令キューを持ち、(IWq, IW, WSq, TW) = (2,4,16,6) である。ただし、IWq、WSq は、各キューの発行幅とサイズ (2.2.3 節参照)、TW はタグのビット幅である。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。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、キャッシュー メモリ以外のすべての資源を 2 倍、すなわち、(IWq, IW, WSq, TW) =

表 2: リセットと pace-keeping による IPC の低下率 (%)

		min	max	av
R10Kx1	reset	0.01(124)	0.75(132)	0.25
	pace-keeping	0.04(124)	1.23(147)	0.47
	both	0.10(124)	1.75(147)	0.74
R10Kx2	reset	0.01(134)	0.77(132)	0.18
	pace-keeping	0.02(124)	1.01(132)	0.28
	both	0.05(124)	1.76(132)	0.48

(4,8,32,7) としたものもあわせて測定した。これを構成 R10K×2 と呼ぶ。

以下まず、3.2.3 で述べた依存行列 エントリのリセットと、4.2.2 項で述べた pace-keeping の影響を調べ、その後に依存行列 の狭幅化について述べる。

リセットと pace-keeping の影響

表 2 に、構成 R10K×1 の従来方式に対して、リセット、pace-keeping、および、その両方を行った場合の IPC の低下率を示す。なお、括弧内はベンチマークのプログラム番号を示す。これらによる IPC の低下率は 1%程度以下である。

L-1 依存行列の狭幅化の影響

次に、L-1 依存行列の狭幅化を行った場合の、従来方式に対する IPC の変化を図 17 に示す。図中、上のグラフが構成 R10K×1 の、中のグラフが構成 R10K×2 のものである。グラフの横軸は依存行列 の幅、縦軸は従来方式に対する IPC の百分率である。1 本の曲線は、1 つのプログラムに対応する。なお、w = WSq、すなわち、L-1 依存行列の狭幅化を行わない場合には、pace-keeping を行う必要はないが、同グラフではあえて行った場合の値を示している。w = WSq の場合の値が 100%より小さいのは、リセットと pace-keeping の影響による。

同グラフからは、L-1 依存行列の幅が、WSq の 1/4 以上の場合、ほとんどが pace-keeping の影響であり、IPC の低下は 1%程度に抑えられることが分かる。

ちなみに、w = 0 のときの値は、ちょうど、2.2.3 節で述べた、wakeup と select にあわせて 2 サイクルをかけた場合に相当する。その場合の IPC は低下は、10%程度にもなる。

L-1 依存行列の幅の上限　w の上限を求めるため、すべての資源を無限とし、キャッシュ、分岐予測を完全とした構成、R10K×∞についても同様に測定を行っ

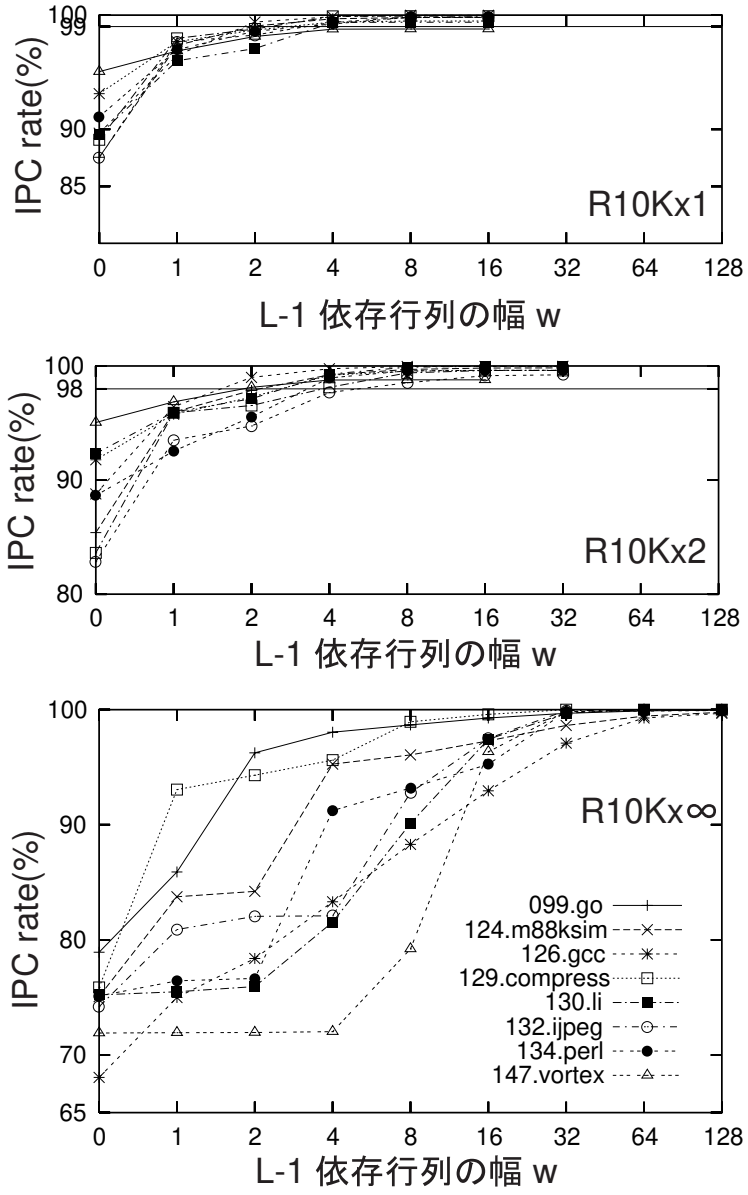


図 17: L-1 依存行列 の幅に対する IPC

た。図 17の下のグラフに示されているその結果からは、w は 16 あれば IPC の大きな落ち込みはなく、また 64 あれば十分であることが分かる。

5.2 回路の評価

富士通株式会社から提供された CS80A プロセスのデザイン・ルールに基づいて、各方式の wakeup ロジックのレイアウト設計を行った。得られたレイアウトから回路面積を求め、また抽出した RC データから、Hspice シミュレーションによって遅延を求めた。以下、5.2.1項で評価方法について説明し、5.2.2項では回路の説明を行う。

5.2.1 評価方法

実際の評価は、レイアウト作成、ネットリスト作成、シミュレーションの順に行った。以下、これらについて説明する。

レイアウト作成

RAM、CAM、行列それぞれセル 1 つ分のレイアウトを行う。使用したツールは cadence 社のレイアウトエディタである。使用したプロセス CS80A は、ゲート長 0.18μm のバルク CMOS プロセスで、ゲート、層間絶縁膜は SiO_2である。配線は、6 層アルミであるが、評価では下 3 層のみを用いている。トランジスタは、通常、高速の 2 種類が用意されており、ダイナミックロジックの N-tree には通常のトランジスタ、それ以外の部分には高速なトランジスタを使用した。

ネットリスト作成

各セルのレイアウトから、トランジスタと容量を抽出し、各セルのネットリストを作成する。そのネットリストを、・パスとなる部分、すなわち最も長いパスに沿って並べて接続した。その際、セル間にまたがる信号線については間に配線抵抗を挿入し、T 型モデルになるようにした。入力部や出力部などに使用されるドライバや、OR 等に関しては、実際のレイアウトは行わず、トランジスタ 1 つを抽出したネットリストを組み合わせて記述した。

シミュレーション

シミュレーションには、avant! 社の Star-Hspice 98.2 を用いた。条件として、温度 85 度、電源電圧 1.8V を与えた。遅延は、入力、出力電圧が (vdd+vss)/2 を通過したタイミングをもとに計測した。入力信号の立ち上がり、立ち下がり時間は 200ps とした。

5.2.2 回路の説明

ベース・モデルとしては MIPS R10000[3] を用い、その 1/2 倍、1 倍、2 倍の計算資源を持つモデルを評価した。それぞれのパラメータは、$IW' = 1, 2, 4$ に対して、$WS' = 8 \cdot IW'$、$NR' = 4 \cdot WS'$、$IW = 3 \cdot IW'$、$WS = 3 \cdot WS'$ 、$NR = 2 \cdot NR'$ である。

以下、各セルの回路について説明する。

RAM セル

連想方式における RAM セルは、IW'- readIW'- write、TW b/× $/WSq$ word である。ただし、TWは、タグのビット幅である。RAM を設計するに当たって、特に読み出しの遅延に関連して、セル・サイズを小さくする方針と、大きくする方針が考えられる。前者は、せる・サイズを小さくし、その結果ゆっくりと変化するビット線の電位をセンス・アンプを用いて読み出す方針である。後者は、セル・サイズは犠牲にして、比較的大きなビット線ドライバをセル内に配置する方法である。今回の評価では、前者を採用した。RAM セルは 4T セル、ライトポート、リードポートから構成される。4T セルは 2 個のインバータをループさせた構造になっており、ここに 1 ビットのデータを記憶する。ライトポートは 4T セルの内容を書き換えるポート、リードポートは記憶している内容を読み出すポートであり、R10K×1 の場合、ライトポート、リードポートは 2 つずつである。R10K×2 の場合はそれぞれ 2 倍、R10K×1/2 の場合は半分である。

CAM セル

CAM セルは、セルの内容と比較線を比較し、不一致の場合のみ match-line を Low にする回路である。

CAM セルは 4T セル、ライトポート、比較器から構成される。これらのうち、4T セルとライトポートは RAM の回路と同じである。比較器にはさまざまなバリエーションがある。

- 比較を複合ゲートで行うか、パスゲート (トランスミッションゲート) で行うか
- 比較部はダイナミックかスタティックか
- ダイナミックなら、比較部が match-line のプルダウンを直接行うか、プルダウン用の回路を別に用意するか

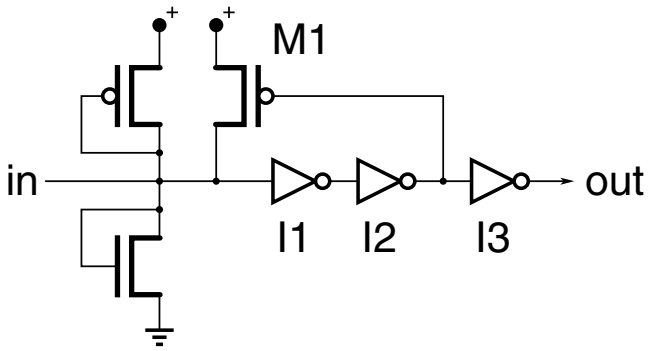


図 18: シングルエンド・センスアンプのロジック

これらのうち、最も高速であったパスゲート– ダイナミック– 直接の組み合わせを用いた。

依存行列のセル
依存行列用の RAM セルとほぼ同じ回路であるが、以下の点で RAM セルと異なる。

- リードポートの数は IW にかかわらず 1 つであり、片側のみ存在する。
- 直接方式用のセルの場合、リセットポートが 1 つある。

また、各セルの読み出し線の末端にはシングルエンドのセンスアンプが設けてある。

図 18 に、各セルで使用されているシングルエンドのセンスアンプの回路図を示す [22]。pMOS M1 は、ビットライン in を充電するが、その電位が $V_{DD}/2$ を超えたあたりで、負のフィードバック・ループによって OFF になる。メモリ・セルによって in がわずかにプルダウンされると、I1 ~ 3 は、高利得領域にあるため、急激にスイッチングする。左側のクランプ・トランジスタ・ペアは、in の電圧振幅を制限する。

シングルエンドのセンスアンプを用いることによって、より早く変化を次ブロックに伝達することが可能となっている。

5.2.3 回路面積の評価

間接、直接方式共に、書き込むべき物理レジスタを指示する tagD を格納するため、連想方式の RAM 部と同じものを別途必要とする。したがって、連想方式の CAM 部と、間接、直接方式の行列の面積を比較する。

tab:area にセルの面積を、図 19 に総面積を示す。

表 3: セル面積 (面積 (F^2) (縦 (F) × 横 (F)))

	×1/2	×1	×2
連想方式	960(24x20)	1360(34x40)	4320(54x80)
間接方式	360(20x18)	400(20x20)	572(26x22)
直接方式	400(20x20)	400(20x20)	572(26x22)

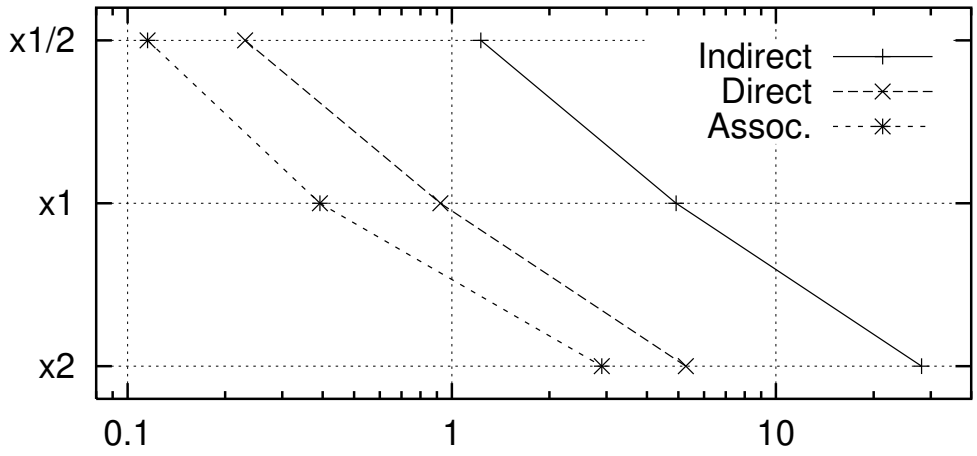


図 19: 回路面積 $(\times 10^6 \mathrm{F}^2)$

セル面積

間接方式の IR と RI、および、直接方式では、ほぼ同一のセルを用いている。直接方式では、間接方式のセルに加えてリセット用のポートを必要とするが、セル面積にはほとんど影響を与えなかった。

多ポート・メモリのセルの面積は、基本的には、ポート数の 2 乗に比例する。しかしその影響の大きさは、ポート数の絶対的な大きさに依存する。

間接、直接方式のセルでは、書き込みポート数が $IW' = 1, 2, 4$ と増加するが、4T セルや読み出しポートなどの定数成分が支配的であるため、$\times 2$ でも 4 割程度の増加に留まっている。

一方、連想方式の CAM 部の比較入力ポート数は、INT と LS は INT と LS から、FP は FP と LS からの
TAGD を受け付けるため、$2 \cdot IW' = 2, 4, 8$ と増加し、$O(IW'^2)$ が支配的になっている。

総面積

セルの総数は、連想方式は $WS \log NR'$、間接方式は $2 \cdot WS \cdot NR = 16/3 \cdot WS^2$、直接方式は WS^2 である。

間接方式と直接方式のセルの面積はほぼ同一であるから、間接方式の総面積は直接方式の総面積のほぼ 16/3 倍となっている。

連想方式の総面積は、$\log NR' = 5, 6, 7$ 程度であるので、
$O(IW'^2) \times O(WS \log NR') \simeq O(IW'^3)$ とできる。

一方、間接、直接方式の総面積は、セル面積の変化が 4 割程度であるので、$O(WS^2) = O(IW'^2)$ となり、オーダは連想方式より小さい。どのモデルにおいても直接方式の面積は連想方式の面積の 2 倍前後となっているが、直接方式の曲線の傾きは連想方式より若干緩やかである。

5.2.4 回路遅延の評価

図 20 に、各方式の回路遅延を示す。間接、直接方式は、L-1 行列の遅延である。直接方式の w は、狭幅化後の L-1 の幅である。Select ロジックの遅延は、固定優先順位の prefix-sum 方式のものである [9]。縦軸は、F.O.4 インバータの遅延 (60.0ps) で正規化してある。各バーの下から n 番目の部分は、図 1、図 5、図 8 における丸数字の n 番から $n+1$ 番までにあたる。白色の部分は主にゲート遅延からなり、灰色の部分は配線遅延の影響を受ける。

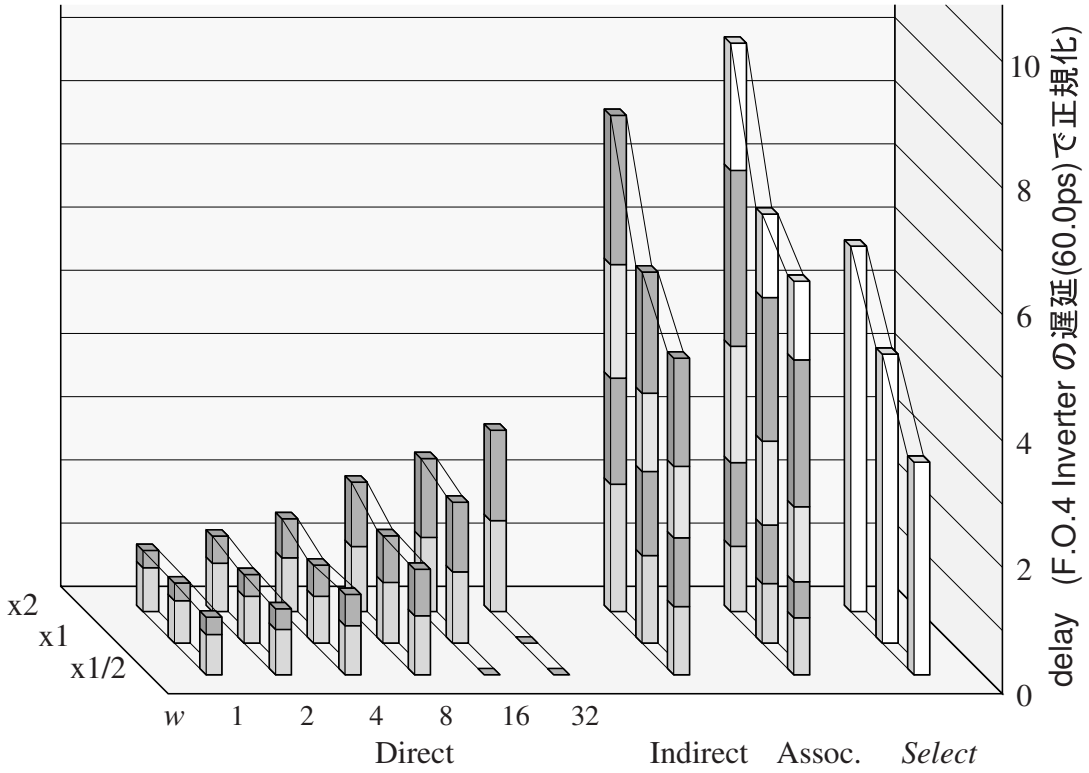


図 20: 各方式の回路遅延

回路の単純化によって、間接方式の RI の遅延は連想方式の CAM 部に比べて大幅に短縮されている一方で、IR では大幅に増加している。これは、ビット幅が $\log_2 NR'$b から NR'b へと 1 桁前後増大しているためである。結局、連想方式に対する wakeup 全体の遅延の改善はごくわずかである。

一方直接方式の遅延は、両者に対して大幅に短縮されている。L-1 の狭幅化を行わなくても、その遅延は両方式の $1/2$ 以下である。$w = WS'/4 = 8$ まで狭幅化した場合、前章で述べたように IPC は約 1%程度低下するが、遅延は更に半減される。

第 6 章　おわりに

本稿では、命令間の依存関係を直接的に表す行列依存行列を用いた Out-of-Order 命令スケジューリング 方式について述べ、また DEC alpha 21264 で採用されている間接方式について、および従来より使用されてきた、RAM と CAM を用いた連想方式との差異についても述べた。従来方式では RAM と CAM の逐次アクセス、間接方式では 2 つの RAM の逐次アクセスであるのに対し、直接方式では、小容量の RAM を 1 回読み出すのみで wakeup を実現することができる。

そして、依存行列をレイテンシにあわせて L-1 依存行列と L-2 依存行列とに分散化し、さらにこの L-1 を狭幅化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 R10000 プロセッサをモデルとして Simplescalar 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、L-1 を $WS'/4$ にまで狭幅化しても、IPCの悪化は 1%程度であることが分かった。また、$R10k \times \infty$の構成の結果より、今後ウィンドウ・サイズがいかに拡大しようとも、L-1 の幅は 16 もあれば IPC の大きな落ち込みはなく、64 もあれば十分であることが分かった。すなわち Out-of-Order 命令スケジューリング・ロジックの遅延は、ウィンドウ・サイズからはほぼ完全に独立になったと結論付けることができる。

また、富士通株式会社から提供された $.18\mu$m CMOS プロセスのデザイン・ルールに基づいて各方式のレイアウト設計、ネットリスト作成を行い、Hspice を用いて遅延を計測した。その結果、間接方式は、2 つの行列を逐次的に用いて wakeup を実現するため、従来の連想記憶を用いた方式と比較すると、回路は単純化されているものの、回路規模が大幅に増大するため、遅延の改善はほとんど見られなかった。直接方式では、MIPS R10000 と同様の構成において、間接方式と比較して面積については 3/16、遅延については半分以下となる。よって、今後、この部分がスーパースケーラ・プロセッサにおいてクリティカルとなる可能性は極めて低いと言える。

以上から、スーパースケーラ・プロセッサ において、この直接依存行列の方

式を用いれば、従来方式に比べ、1%程度のIPCのペナルティを犠牲に、wakeupの遅延を半分以下とすることができる。そして、Out-of-Order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。

謝辞

富士通株式会社には、LSI の設計情報をご提供いただきましたたので、ここに深甚なる謝意を表します。

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る謝意を表します。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、五島正裕助手に深く感謝致します。さらに、共同研究者である小西正人氏、芦川司氏、小出義和氏をはじめとして、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] Keller, J.: The 21264: A Superscalar Alpha Processor with Out-of-Order Execution, *9th Annual Microprocessor Forum* (1996).

[2] Palacharla, S. et al.: Complexity-Effective Superscalar Processors, *ISCA24* (1997).

[3] Yeager, K.: The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, No. 4 (1996).

[4] Palacharla, S. et al.: Quantifying the Complexity of Superscalar Processors, Technical report, Univ. of Wisconsin-Madison (1996).

[5] Morancho, E. et al.: Recovery Mechanism for Latency Misprediction, *PACT* (2001).

[6] Brown, M. et al.: Select-Free Instruction Scheduling Logic, *MICRO-34* (2001).

[7] Thornton, J. E.: Parallel operation in the Control Data 6600, *Proc. AFIPS Fall Joint Computer Conf. Part II*, Vol. 26, pp. 33–40 (1964).

[8] Tomasulo, R. M.: An effective algorithm for exploiting multiple arithmetic units, *IBM J.*, Vol. 11 (1967).

[9] Henry, D. et al.: Circuits for Wide-Window Superscalar Processors, *ISCA27* (2000).

[10] Farkas, K. I., Chow, P., Jouppi, N. P. and Vranesic, Z.: The Multicluster architecture: reducing cycle time through partitioning, *MICRO-30* (1997).

[11] Canal, R. et al.: Dynamic cluster assignment mechanisms, *HPCA6* (2000).

[12] Canal, R. et al.: A low-complexity issue logic, *ICS'00* (2000).

[13] 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, *JSPP2000*, pp. 197–204 (2000).

[14] 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, No. 4, pp. 652–662 (2001).

[15] Sato, T. and Arita, I.: Simplifying Wakeup Logic in Superscalar Processors, *Joint Symp. on Parallel Processing 2001*, pp. 23–30 (2001). (in Japanese).

[16] Goshima, M., Nguyen, H., Agata, A., Nakashima, Y., Mori, S., Kitamura,

T. and Tomita, S.: Instruction issue logic of the Dualflow architecture, *J. IPS Japan*, Vol. 42, No. 4, pp. 652–662 (2001). (in Japanese).

[17] Farrell, J. et al.: Issue logic for a 600-MHz out-of-order execution microprocessor, *IEEE J. of Solid-State Circuits*, Vol. 33, No. 5 (1998).

[18] SPARC International Inc.: *The SPARC Architecture Manual Version 9* (1994).

[19] 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 (SWoPP '98), pp. 115–120 (1998).

[20] *SimpleScalar LLC: http://www.simplescalar.com/*.

[21] Burger, D., Austin, T. M. and Bennett, S.: Evaluating Future Microprocessors: The SimpleScalar ToolSet, Technical Report CS-TR-1308, Univ. of Wisconsin-Madison (1996).

[22] Bakogulu, H.: *Circuits, Interconnections, and Packaging for VLSI*, Addison-Wesley (1990).