行列に基づく Out-of-Order スケジューリング方式の評価

五島正裕[†] 西野賢悟[†] 小西将人[†]
中島康彦[†] 森眞一郎[†]
北村俊明[†] 富田眞治[†]

Out-of-order スーパースケーラ・プロセッサは，命令スケジューリングのため，オペランドの有効性を追跡する *wakeup* と呼ぶロジックを持つ．我々は，命令間の依存関係を表す行列を読み出すことで *wakeup* を実現する方式と，その行列を狭幅化することにより遅延を IPC に対するペナルティに転化する手法を提案した．しかし，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．本稿では，これらの違いを明らかにする．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方式のロジックを設計し，Hspice によって遅延を測定した．その結果，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．

Evaluation of Matrix-based Out-of-Order Scheduling Schemes

Masahiro Goshima,[†] Kengo Nishino,[†] Masahito Konishi,[†]
Yasuhiko Nakashima,[†] Shin-ichiro Mori,[†] Toshiaki Kitamura[†]
and Shinji Tomita[†]

An out-of-order superscalar processor has a logic called *wakeup*, which manages availability of the data for instruction scheduling. We have proposed a new scheduling scheme which substitutes association of the tags by reading a matrix which represents dependences between instructions, and a method to change the delay of the matrix into IPC penalties. A similar scheme based on matrices, however, has already adopted in DEC Alpha 21264. This paper clarifies the difference between them. We designed the logic circuits guided by a design rule of a 0.18μm CMOS process provided by Fujitsu Limited, and calculated the delays by Hspice. The evaluation result shows that the circuit delay of our scheme is less than half of that of the scheme adopted in 21264.

1. はじめに

現在の out-of-order スーパースケーラ・プロセッサでは，命令スケジューリングを行うウィンドウ・ロジックの遅延がプロセッサの動作周波数を制限する主要因の 1 つとなりつつあり，その遅延を短縮する研究が盛んに行われるようになってきている[1)~10)]．

ウィンドウ・ロジックのうちでは，***wakeup*** と呼ばれる部分が特にクリティカルになると予測されている[1)~4)]．命令の発行に必要なデータが揃っていない命令はウィンドウ中で『眠る』ことになる．*Wakeup* は，この『眠っている』命令を『起こす (wake up)』ロジックである．従来から採用されてきた方式は，各命令が待ってい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．この方式を**連想方式**と呼ぶことにする．連想方式では，およそ 0.5 サイクルの間に RAM と CAM を逐次的にアクセスする必要があり，このことが *wakeup* の遅延をクリティカルにしていた[1),2)]．

そこで我々は，この連想処理を廃し，行列を用いて *wakeup* を行う方法を提案した[1),2)]．一方，DEC Alpha 21264 では，行列を用いた類似の方法が既に採用されている[5)~7)]．これらの方式は，命令間の依存を表現する行列を用いて *wakeup* を実現するという点で一致しているが，依存の表現の方法が異なる．前者では生産者から消費者への依存を表す単一の行列を用いる．一方後者では，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それぞれ表す 2 種類の行列を用いる．そこで本稿では，前者を**直接方式**，後者を**間接方式**と呼ぶことにする．

本稿の目的は，これらの 2 方式の違いを明らかにすることにある．以下まず 2 章では，その準備として out-of-order スケジューリングの原理と従来の連想方式についてまとめる．そして 3 章と 4 章で間接方式と直接方式について詳しく述べ，その定性的な違いを明らかにする．定量的な比較は 5 章で行う．

† 京都大学

2. 連 想 方 式

前述の通り間接方式では，2 種類の行列を用いて *wakeup* を行う．しかし，6), 7) などでは，prrdy から命令への依存を表す行列のみが記述されているため，直接方式との違いが不明瞭になっている．これは，*wakeup* という言葉が曖昧に用いられているためである．本章では，従来の連想方式の説明を行うと共に，*wakeup* のより正確な定義を与える．

2.1 Out-of-Order 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 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 (*wakeup*)』，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 スケジューリングの処理は，以下の 5 つのフェーズに従って進む．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☆．$I_l/I_r/I_c$ は，それぞれ，命令ウィンドウの $l/r/c$ 番エントリ（以下，**iwent[*l*/*r*/*c*]**）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（**preg[*L*/*R*]**）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，空いている preg の 1 つが割り当てられる．この preg 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．

(2) *Dispatch*　その後命令は，命令ウィンドウに格納される．*Rename* の結果に従って，tagL[c] = tagD[l] = L，tagR[c] = tagD[r] = R となる．

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．

☆ 簡単のため各命令は左/右 2 つのソースを持つとしたが，本稿の議論は 3 つ以上のソースを持つ場合にも容易に拡張できる．

2.3 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．

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 preg 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，preg の書き込みアドレスとして使用される．

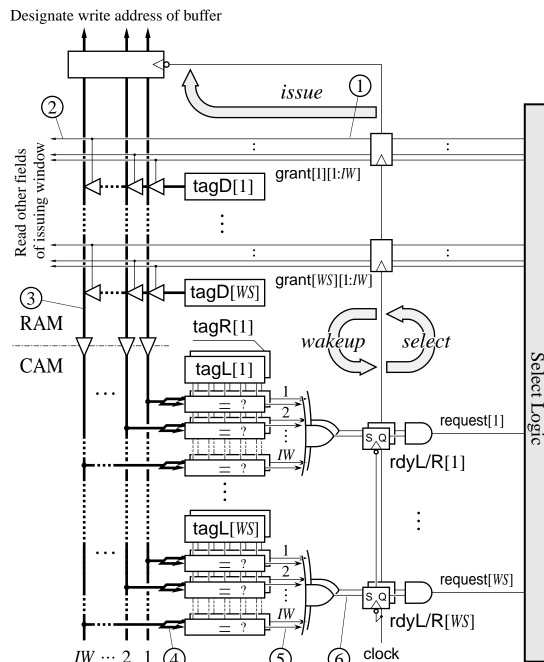

図 1　連想方式の *wakeup* ロジック

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，*IW* 組の grant を *wakeup* ロジックに対してアサートする．

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 [11]．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．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・プロセッサでは，*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．

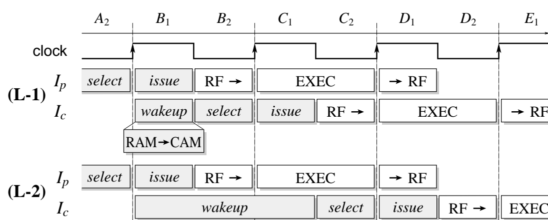

図 2　命令パイプラインにおける *wakeup*，*select*，*issue*

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．それによる IPC の悪化は最大 15%程度にもなり [1),2)]，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．

2.4 *Wakeup* の定義

次章からは，行列を用いた *wakeup* の 2 つの方式，間接方式と直接方式について述べる．前述の通り，各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 [3),4),6),7)] が，本稿では，本章で述べてきた通り，RAM 部と CAM 部，すなわち，grant から request までを *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．

3. 間接方式 [5)~7)]

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 preg が利用可能かどうかを表す 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 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 preg との関係を表す．したがって，それぞれ，*WS* 行 ×*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 3 に，間接方式の概念図を示す．同図は，2.2 節と同様に，iwent[*l*/*r*] に格納された命令 I_l/I_r が生産する結果を，preg[*L*/*R*] 番を介して，iwent[*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

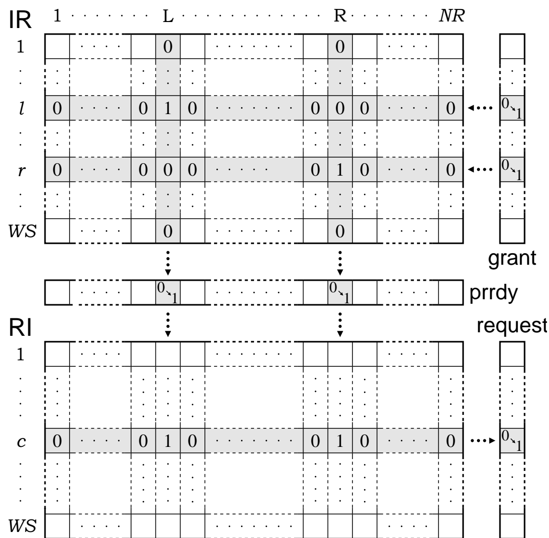

図 3　間接方式の概念図

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 3 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．

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，preg[L/R] 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．

3.2　間接方式のロジック

図 4 に，間接方式のロジックを示す．ロジックは，図 3 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×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 4 の場合，c 行 X 列のセルの

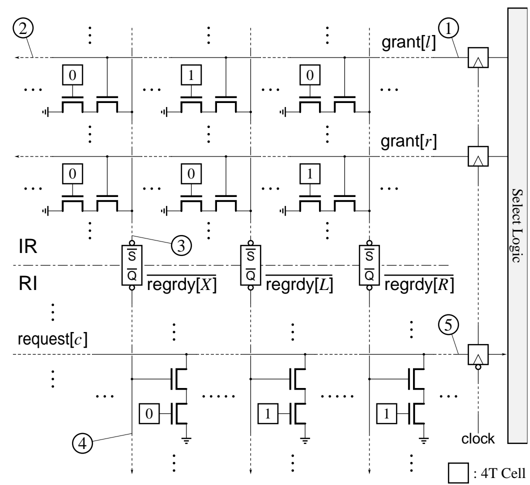

図 4　間接方式のロジック

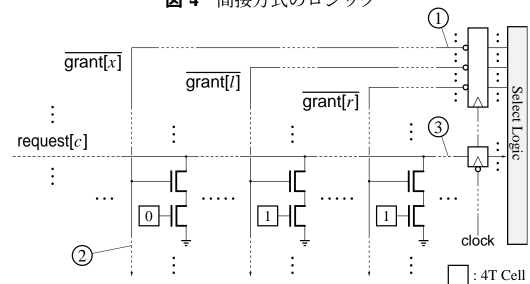

図 5　直接方式のロジック

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 4 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．

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．

3.3　連想方式との関係

間接方式は，前章で述べた連想方式との関係からは，以下のように説明できる．IR/RI は，RAM 部/CAM 部と 1 対 1 に対応する．IR/RI の各行は，基本的には，preg 番号，すなわち，タグをデコードしたものである．

連想方式では IW 個のタグを IW 個の部分回路で処理していた．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できるようになっている．そのため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．

4. 直接方式 [1),2)]

間接方式が IR，RI の 2 種類のを逐次的にアクセスするのに対して，文献 1)，2) で提案した直接方式では直接 grant から request を求める．また，1)，2) では，行列の実効サイズを縮小しアクセスを高速化する手法も併せて提案している．

文献 1)，2) で提案した方式では，ソース・オペランドの数だけの行列を必要としていたが，本稿ではそれらを単一の行列に単一化する方法を提案する．以下まず 4.1 節で 1)，2) に基づいて直接方式の原理について説明した後，4.2 節と 4.3 節で単一化について述べ，4.4 節では単一化した行列を高速化するという視点から述べる．

4.1 直接方式の原理

図 6 に，直接方式の依存行列の概念図を示す．行列は，rdyL/R 用に各 1 つづつ用意された $WS \times WS$ の行列である．ただし，対角要素は使用しない．iwent[p] の命令 I_p の実行結果を iwent[c] の命令 I_c が消費する場合，c 行 p 列の要素は 1，そうでなければ 0 とする．

Wakeup においては，発行される命令に対応する列の行ごとの OR を求めれば，セットすべき rdyL/R を表す列ベクトルを求めることができる．

この処理は，間接方式における IR と転置の関係にあり，IR と同様のロジックによって実現される．

4.2 ウィンドウ・エントリと物理レジスタの寿命

行列の単一化について理解するためには，iwent と preg の寿命の違いについて理解しておく必要がある．そこで本節でまず，そのことについてまとめた後，次節で行列の単一化について述べる．

ある命令に対して，それが格納される iwent 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 preg は，異なる寿命を持つ．iwent と preg 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 [6)]，基本的には，それぞれ以下のタイミングで開放される：

iwent　命令の実行が終了すれば開放してよい．

preg　命令の実行が終了した時点ではなく，以下のように，当該 preg 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**　lreg を上書きする次の命令が完了する時まで [11)]

この寿命の違いは，以下のように，iwent と preg は個別に管理されるべきであることを意味する．

iwent は，サイクル・タイムとのトレードオフと観点から言って，preg に比べ高価な資源である．iwent 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，iwent と preg 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，iwent にも preg の寿命が強制されることになる．すなわち，対応する preg の寿命が尽きるまで，preg より高価な iwent まで再利用できなくなってしまう．

以上の理由から，iwent と preg 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，iwent と preg を個別に管理している．preg の数 $\boldsymbol{NR}$ は，$NR = 2 \cdot WS$ とすることが多い [5),11)]．

4.3 行列の単一化

1)，2) で示した直接方式では，ソースごとに 1 つの行列を必要とした．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．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 6 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．

単一化された行列のロジックは，**図 5** のようになる．図 4 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．

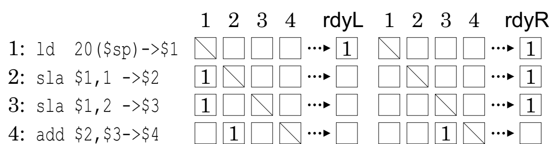

図 6　直接方式の依存行列

対 処 法 1

間接方式の prrdy と同様，行列の前にレジスタ **insnrdy** を置くことにしよう．

その役割を果たすためには，insnrdy 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 insnrdy は，前節の説明に従えば，iwent より長い，preg と同様の寿命を持つとすることができる☆．すなわち，格納された命令 I_p の実行が終了した後，iwent[p] は原理的には解放してよいが，insnrdy[p] は解放できない．

そのため iwent[p] も，insnrdy[p] が解放されるまで再利用することはできない．結局，iwent[p] も，insnrdy[p] と同じ preg 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 [6]．

4.4 行列アクセスの高速化

本節では，1), 2) で提案した行列アクセスの高速化手法について述べる．そのうちの一部は間接方式にも適用可能であり，その方法については 4.5 節で述べる．

4.4.1 行列の分散化

実際のプロセッサでは，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．実際，最近のレジスタ・リネーミング方式のプロセッサの多くは，整数 (**INT**)，ロード/ストア (**LS**)，浮動小数点 (**FP**) 命令の系統ごとにウィンドウを分散化している．例えば MIPS R10000 は，INT 用，LS 用，FP 用のサブウィンドウを持つ [5),11)]．このような分散化は，わずかな IPC のペナルティを犠牲に，ロジックを大幅に縮小できるため，非常に重要である．

☆ 正確には，リオーダ・バッファ方式の preg と同じ寿命を持つ．

R10000 と同様の分散化を施した場合の行列の分散化の様子を**図 7 (b)** に示す☆☆．R10000 では，INT，LS，FP の各サブウィンドウの発行幅とサイズはそれぞれ，$\boldsymbol{IW'} = IW/3 = 2$，$\boldsymbol{WS'} = WS/3 = 16$ である．

分散化の行列に対する第一の効果は，書き込みポートの削減である．INT 命令を *dispatch* するときに書き込まれる行は，対応する WS' 行に制限される．この部分のセルの書き込みポート数は，IW から IW' へと 1/3 に削減される．LS，FP についても同様である．

また，R10000 のように INT のみのサブウィンドウを持つ場合，次項で述べる多階層化が可能になる．

4.4.2 行列の多階層化

R10000 の構成では，レイテンシが 1 であるパスは，INT から INT，INT から LS の 2 つである．図 7 (b) では，影を付けた部分がこれに相当する．この部分を取りだし，これを **L-1**，残りを **L-2 行列**と呼ぶ．

L-1 アクセスは *select* と合わせて 1 サイクル以内に行う必要があるが，L-2 はパスのレイテンシに合わせて適当にパイプライン化してよい．図 2 (L-2) は，レイテンシが 2 の場合にあたる．同図では，L-2 には 1.5 サイクル，すなわち，L-1 の 3 倍の時間をかけており，L-2 がクリティカルになる可能性は極めて低い．

一方 L-1 は，元の 行列から比べると格段に少量化され，その分だけ遅延も短縮される．更に，L-1 に対

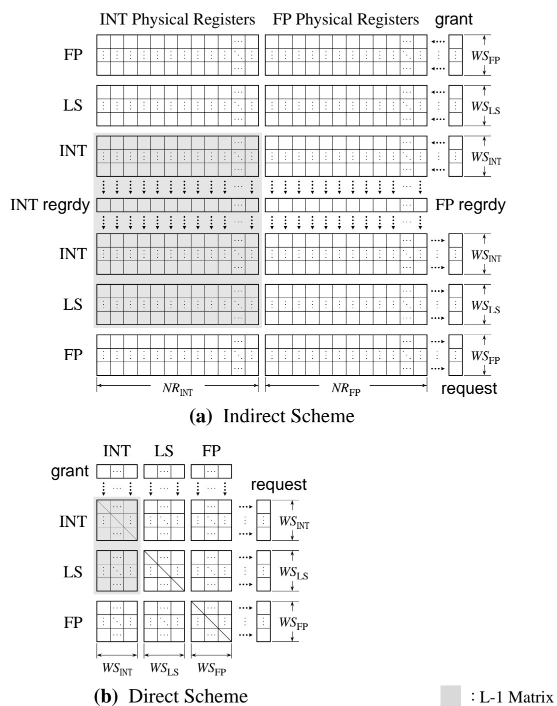

図 7 行列の分散化

☆☆ SPARC など，INT–FP レジスタ間の転送命令が無い ISA では，同図中破線で記した部分行列を省略できる．

しては次節で述べる狭幅化を適用することができる．

4.4.3 L-1 行列 の狭幅化

依存する命令間の距離は短い場合が多い．この性質を利用して，L-1 を更に縮小することができる．各行において，先行する w 命令に対するビットだけを残し，それ以外を L-2 に移すのである．*Wakeup* は，命令間の距離が w 以下の場合には L-1 によって，そうでない場合には L-2 によって行われる．後者の場合には，1 サイクルのペナルティが生じる．

図 8 に，L-1 の縮小の様子を示す．左は元々の，右が縮小後の L-1 である．L-2 に移されるセルは，左図中で薄く示してある．メモリのラインのうちでは，書き込みポートのビットラインが $2 \cdot IW'$ 本と最も本数が多い．したがって図 8 では，それらが直線になるように，L-1 に残されるセルを矩形領域に集めている．w は，L-1 を構成するアレイの幅にあたる．そのため，この技術を **L-1 行列の狭幅化**と呼ぶ．

図 8 右から明らかなように，狭幅化された L-1 のワード，ビットラインは，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 *wakeup* の遅延は，*WS* とは独立に，w によって決まる．

$w \geq \min(WS'/4, 64)$ であれば，IPC の低下は 1〜2%に抑えられることが分かっている [1),2)]．

4.5 間接方式に対する高速化手法の適用

間接方式の IR，RI に対しても，直接方式と同様の分散化と多階層化を適用ことができる．図 7 (a) に，その様子を示す．同図 (a) は，同図 (b) と同様，R10000 の構成に対するものである．

preg は，INT，FP のそれぞれに分けられ，それぞれの個数は，$NR' = 4 \cdot WS'$ とすることが多い [5),11)]．

しかし，L-1 の狭幅化は，間接方式には適用することができない．直接方式と同様に，割り当てられる iwent と preg との関係を制限して行列を縮小することは可能だが，直接方式における命令間の近さのような有効なヒューリスティクスが，iwent と preg の間には存在しないからである．

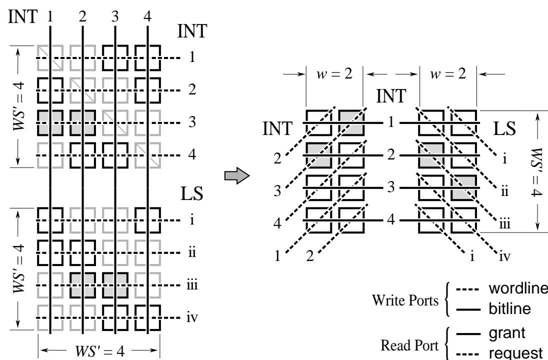

図 8 L-1 行列の狭幅化

5. 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，Hspice によって遅延を求めた．

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．

ベース・モデルとしては MIPS R10000[11)] 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，$WS' = 8 \cdot IW'$，$NR' = 4 \cdot WS'$，$IW = 3 \cdot IW'$，$WS = 3 \cdot WS'$ である．

5.1 回 路 面 積

間接，直接方式共に，書き込むべき preg 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．

表 1 にセルの面積を，**図 9** に総面積を示す．

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，×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

表 1 セル面積 (面積 (F^2) (縦 (F) × 横 (F)))

	x1/2	x1	x2
連想方式	960 (24× 40)	1360 (34× 40)	4320 (54× 80)
間接方式	360 (20× 18)	400 (20× 20)	572 (26× 22)
直接方式	400 (20× 20)	400 (20× 20)	572 (26× 22)

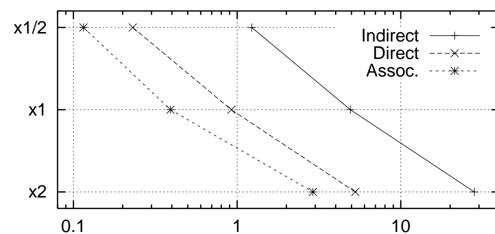

図 9 回路面積 (×10^6F^2)

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　回 路 遅 延

図 10 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 [1),2),8)]．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 1，4，5 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．

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，IPC は約 2%程度低下するが [1),2)]，遅延は更に半減される．

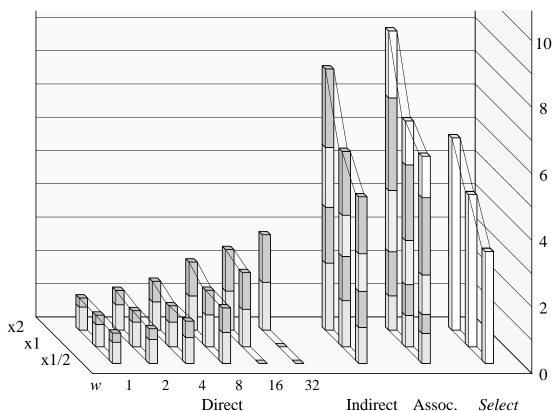

図 10　各方式の回路遅延

6. おわりに

本稿では，out-of-order スーパースケーラ・プロセッサの行列を用いたスケジューリング方式として，DEC Alpha 21264 で採用されている間接方式と，我々が提案した直接方式について述べた．富士通株式会社から提供された 0.18μm CMOS プロセスを用いて，それらの回路の面積と遅延を比較した．

間接方式は，2 つの行列を逐次的に用いて *wakeup* を実現する．間接方式では，回路は単純化されるものの，回路規模が大幅に増加するため，従来方式に対してほとんど遅延の改善が見られなかった．

一方直接方式は，間接方式の 2 つの行列を 1 つにまとめた方式と考えることができる．そのため，間接方式に対して，面積は 3/16，遅延は 1/2 以下になることが分かった．

謝辞

富士通株式会社には，LSI の設計情報をご提供頂いたので，ここに深甚なる謝意を表したい．

本研究の一部は文部省科学研究費補助金，基盤研究 (B)(2) #13480083 による．

参　考　文　献

1) 五島正裕ほか: スーパースケーラのための高速な動的命令スケジューリング方式, 情報処理学会 HPS 論文誌, Vol. 42, No. SIG 9(HPS 3) (2001).

2) Goshima, M. et al.: A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors, *MICRO-34* (2001).

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

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

5) 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).

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

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

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

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

10) 佐藤寿倫ほか: 連想検索を取り除いたスーパースカラプロセッサ向け命令発火機構, *JSPP* (2001).

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