計算機アーキテクチャ 139−18
（2000．8．4）

Dualflowアーキテクチャの命令発行機構

グェン ハイハー† 五島正裕† 縣 亮慶†
中島康彦†† 森眞一郎† 富田眞治†

Superscalarの動的命令スケジューリングのロジックは，CAMで構成され，配線遅延に支配されるため，LSIの微細化に伴ってクリティカルになる．Dualflowは，制御駆動とデータ駆動の性質をあわせ持つアーキテクチャであり，このCAMをRAMに置き換えることができる．本稿では，双方のロジックのトランジスタ・レベルの回路図を示し，その遅延の差を定性的に評価する．

Instruction issue logic of the Dualflow architecture

NGUYEN HAI HA,† MASAHIRO GOSHIMA,† AKIYOSHI AGATA,†
YASUHIKO NAKASHIMA,†† SHIN-ICHIRO MORI† and SHINJI TOMITA†

The dynamic instruction scheduling logic of a superscalar is composed of a CAM, and becomes more critical with smaller feature sizes. Dualflow, a hybrid processor architecture between control- and data-driven, can replace the CAM by a RAM. In this paper, we show transistor-level schematic of both logics, and give qualitative evaluation to them.

1. はじめに

Superscalar（以下，SS）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SS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*を制限する主な要因となりつつある．*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．

SS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[1]．*wakeup*は，動的命令スケジューリングを行うロジックの一部で，命令の発行に必要なソース・オペランドの有効性を追跡する部分である．

*wakeup*ロジックが将来クリティカルになると予測される理由は2つある．まず第1に，このロジックは，他の多くの構成要素とは異なり，複数のパイプライン・ステージを割り当てることができない．第2に，このロジックは**CAM**で構成され，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．以上の理由により*wakeup*ロジックは，命令パイプラインの深化，LSIの微細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，**Dualflow**と呼ぶ命令セット・アーキテクチャを提案した[4),5)]．Dualflowは，制御駆動とデータ駆動の両方の性質をあわせ持つアーキテクチャである．Dualflowは，制御駆動型計算機と同様の制御の流れを持つが，レジスタを定義しない．データの宛先となる命令は，命令間の変位として，生産側の命令コード中に即値的に埋め込まれる．

この，生産側の命令が消費側の命令を直接的に指定するという性質によって，SSではCAMで構成される*wakeup*ロジックをRAMに置き換えることができ，SSと同様のout-of-order実行を行いながら，その複雑さを大幅に軽減することができる．

本稿では，Dualflowの*wakeup*ロジックについて述べる．以下，まず2章では，SSにおける*wakeup*ロジックの構成法を紹介し，その遅延の内訳を明らかにする．次いで3章でDualflowアーキテクチャについてまとめ，そして4章で，Dualflowの*wakeup*ロジックについて詳しく述べ，SSの*wakeup*ロジックとの比較を行う．

† 京都大学情報学研究科
Gradudate School of Informatics, Kyoto University
†† 京都大学経済学研究科
Gradudate School of Economics, Kyoto University

2. Superscalar の *wakeup* ロジック

本章では，SS の *wakeup* ロジックについて述べる．まず，2.1 節において，SS の動的命令スケジューリングの原理についてまとめた後，2.2 節でスケジューリングの処理とパイプラインの関係について説明する．そして 2.3 節で，*wakeup* ロジックについて詳述する．

2.1 Superscalar の動的命令スケジューリング

Out-of-order SS は，マシン状態を表すレジスタとは別に，各命令の実行結果を一時的に保存するバッファを必要とする．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．動的スケジューリングは，このバッファを I-structure のように用いることによって，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．

先行する命令 I_d が定義する結果を後続の命令 I_u が使用する場合を考えよう．I_d から I_u にオペランドが渡されることに注目すると，動的スケジューリングの処理の進行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** へのリネーミングが行われる．
I_d には，空いているバッファが割り当てられ，バッファはオペランドが『ない』状態に初期化される．このバッファの ID をタグという．
I_u は，論理レジスタ番号の依存関係から，対応する I_d に割り当てられたバッファの ID—タグを得る．I_u は，タグで示されるバッファにデータが書き込まれるのを待つ．

(2) ***wakeup*** I_d の実行にともなって，I_u が実行可能になることを検出する．
I_d が実行されると，その結果がタグで示されるバッファに書き込まれ，バッファはオペランドが『ある』状態に遷移する．
I_u は，タグで示されるバッファにオペランドが『ある』ことを見て，発行可能な状態になる．

(3) ***select*** 必要なオペランドが揃った命令から，実際に発行するものが選択され，発行される．

SS 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と述べたが，本稿の議論では両者の違いは重要ではない．重要なのは，原理的には，**オペランドを受け渡すバッファを，I_d，I_u の双方がタグを用いて特定する** ということである．

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* を行う各ロジック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 SS は，この遅延のため，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，それだけ分岐予測ミス・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．例えば，*wakeup* と *select* のそれぞれに 1 サイクルかけた場合，先行する命令が生産するデータを消費する命令は，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．このことはオペランド・バイパスを一切行わないことと等価であり，それによる IPC の悪化はクロック速度の向上に見合わない可能性が高い．以上の理由により，*wakeup* と *select* は，合わせて 1 サイクルで実行しなければならない．

2.3 Superscalar の *wakeup* ロジック

前述したように，SS の *wakeup* ロジックは，CAM を用いて実装される．**図 1** に，ウィンドウ・ロジックの実装例を示す[1]．

図 1 中，tagD はデスティネーションの，tagL/R は左右のソースのタグを格納する．tagD と tagL/R の一致比較器を行い，待っているバッファにオペランドが『ある』か『ない』かを表すフラグ rdyL/R を求める．

図 1 右は，左の網掛け部分に相当する．この回路の上半分は，tagD を記憶する $TAGW$ b×WS word の RAM である．ただし $TAGW$ は，タグのビット幅（5〜7 程度）である．この RAM には，IW 本の書き込みポートと，IW 本の読み出しポートが必要である．

回路の下半分は，$TAGW$ b×2·WS word の CAM である．CAM の上半分は tagL/R を記憶する RAM セルとそれへの IW 本の書き込みポートで，下半分は IW 個の 1b 比較器である．

SS の *wakeup* ロジックの遅延は，以下の 4 つに分解できる：

Tag Read *select* ロジックによって選択された（最大）IW 個の命令の tagD を読み出す．

Tag Drive 読み出された IW 個の tagD を，縦に引かれたタグ・ラインによって，2·WS 個の CAM セルに放送する．

Tag Match 入力されたタグと，待っているオペランドのタグを比較する．1 個の比較器は，タグの全てのビットに渡って引かれたマッチ・ラインに対する wired-AND として実現されている．

Match OR IW 個の一致比較器の出力を OR して，rdyL/R フラグをセットする．

図から分かるように，ロジックの遅延は配線遅延に支配されている．これらの配線遅延のため，LSI の微細化にともなって *wakeup* ロジックがクリティカルになっていくと予測される．

wakeup の遅延

Palacharla らは，各ロジックの詳細なモデリングを行った上で，Spice を用いてそれらの遅延時間を見積もっている[1]．その結果を**図 2** に示す．

グラフ中，横棒は各々の遅延時間を表す．3 つのグル

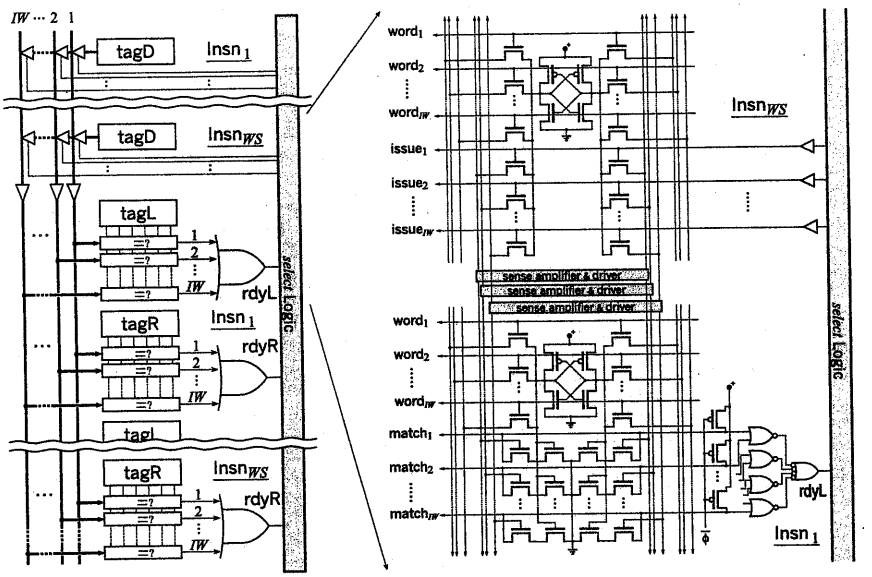

図1 SS のウィンドウ・ロジック

Fig. 1 Window logic of superscalar

ープは，feature sizeがそれぞれ .8μm，.35μm，.18μm の場合である．各グループ内の2つは，*IW, WS* がそれぞれ 4，32；8，64 の場合を表す．

グラフからは，LSI が微細化されていくにしたがって，Match OR と *select* の遅延は大幅に短縮されているが，Tag Drive と Tag Match の遅延はそれほどではないことが分かる．それは，Match OR と *select* ではゲート遅延が支配的であるのに対して，Tag Drive と Tag Match では配線遅延が支配的であるためである．グラフ中の折れ線は，Tag Drive と Tag Match の遅延の和の全体に占める割合を表している．LSI の微細化にともなって，Tag Drive と Tag Match の占める割合が急激に増加していることが分かる．

また，*IW, WS* が 4，32 の場合と 8，64 の場合で，遅延に大きな差がないことも分かる．この程度の領域では，定数成分が無視できないためである．演算器やレジスタ・ファイルに対するのと同様に，ウィンドウ・ロジックをクラスタリングすることによって実効的な *IW, WS* を削減することが考えられている[2]が，オペランド・バイパスの場合とは異なり，*IW, WS* を減らしたところで遅延が劇的に短縮される訳ではない．

なお，彼らは Tag Read に関する考察を行っていないが，その遅延は，他の部分に関して無視できるほど小さいという訳ではない．図1から分かるように，Tag Read も配線遅延．TagD を格納する RAM の，ワード線は match と，ビット線は tagD ラインと，それぞれ同程度の長さである．Tag Read 全体の遅延は Tag Drive+Tag Match と同程度であると予想される．

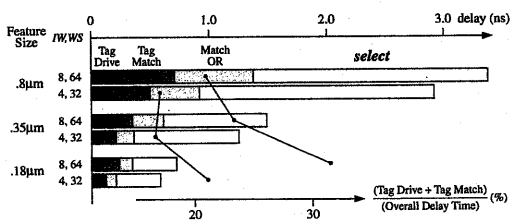

図2 *wakeup* と *select* の遅延

Fig. 2 Propagation delay time of *wakeup* and *select*

3. Dualflow アーキテクチャの概要

SS の *wakeup* ロジックは，端的に言えば，データを受け渡すためのバッファにオペランドが『ある』か『ない』かを表すフラグの配列である．

ここでまず，この配列を RAM を用いて実装する方法について考えよう．RAM を用いる方式は直感的である．タグをアドレスとする 1b×*WS* word の RAM を用いてテーブルを構成すればよい．SS では，I_d，I_u の双方がタグによってバッファを特定する，すなわち，I_d が実行されてフラグをセットする時，および，待っている I_u がフラグを読み出す時に，双方ともタグをアドレスとしてこの RAM にアクセスすることになる．

I_d は1サイクルにたかだか *IW* 個しか存在しないので，この RAM の書き込みポートは *IW* (4〜8) 本あればよい．ところが読み出しポートは，待っている命令が必要とするオペランドの数，すなわち，2·*WS* (32〜128) 本も必要となる．したがって，RAM を用いる方式は，SS の *wakeup* ロジックとしては，現実

的ではない.

読み出しポートを省略するため，フラグは，バッファに対してではなく，I_u に対して割り当てる必要がある．すると必然的に，SS の *wakeup* ロジックは，CAM とならざるを得ないのである．

さてSSで，I_d に加えて I_u がデータを受け渡すバッファをタグを用いて特定するのは，端的に言えば，その制御駆動モデルによるものである．

本稿で提案する **Dualflow** は，ISA のレベルからデータ駆動的性質を導入することで，この CAM を RAM に置き換えることができる．

以下本章では，データ駆動的性質がどのようにして CAM を RAM に置き換えるのかを明らかにしよう．まず 3.1 節と 3.2 節で，Dualflow の実行モデルと実装について述べたる．問題の *wakeup* ロジックについては，続く 4 章で詳細に述べる．

3.1 Dualflow の実行モデル

Dualflow は，以下のように，制御駆動とデータ駆動の両方の性質を合わせ持つ：

制御駆動 プログラム・カウンタがあり，分岐命令によって制御の流れを移譲する．

データ駆動 アーキテクチャはレジスタを定義せず，命令間で直接的にデータの授受を行う．

命令フォーマット (1)

まず，命令フォーマットについて簡単に触れておく．命令中には，通常の RISC のような，ソースやデスティネーションとなるレジスタを示すフィールドはない．代わりに，命令の実行結果の宛先命令を示すフィールドがある．宛先フィールドについては後述する．

各命令は，(レジスタ・ファイルなどから) 能動的にデータを取り出すのではなく，先行する命令が送りつけたデータを受動的に使用し，実行結果を宛先フィールドが示す後続の命令に送りつける．そのため，命令中にはソースを指定するフィールドはない．

実行モデル

Dualflow の実行は，**プレース** と呼ぶ論理的なデータ構造の上で行われる．各プレースは，3 つの **スロット** からなる．スロットの 1 つは命令を格納する **命令スロット** であり，残りの 2 つは命令の左右のソース・オペランドを格納する **データ・スロット** である．

各プレースには，命令とデータがばらばらに届く．命令と必要なデータが揃ったプレースが，その順序とは無関係に，すなわち，out-of-order に実行される．

以下のように，命令は制御駆動的にデータはデータ駆動的にプレースに届く：

命令 命令は通常の制御駆動と同様にプログラム・カウンタにしたがってメモリからフェッチされ，フェッチされた順序でプレース列の命令スロットに格納されていく．

データ 各プレースが実行されるとその結果は，データ駆動と同様に，命令中に示される宛先に送られる．ただし，宛先の指定の方法はデータ駆動とは異なる．データ駆動では，実行結果の宛先は命令であり，各命令は宛先の命令のアドレスを指示する．一方 Dualflow では，宛先は命令ではなく，後続のプレースのデータ・スロットである．

すなわち，データ駆動では命令のあるところでデータとデータが待ち合わせるが，Dualflow ではプレースで命令とデータが待ち合わせる．

命令フォーマット (2)

ここで再び命令フォーマットに話を戻そう．前述した宛先フィールドは，正確には，宛先のデータ・スロットを示す．宛先フィールドは，自命令と宛先データ・スロット間の変位と，宛先データ・スロットの左右を表すサブフィールドからなる．

現在では，ハードウェア量とのトレードオフから，disp サブフィールドは 5b，宛先の数は 2 を想定している．その場合，実行結果を送ることができるのは，距離が 31 以内にある最大 2 つの命令に制限される．

制御駆動では命令が，データ駆動ではデータが，それぞれ計算の主体であると言われる．そのような観点から言えば，Dualflow では，命令とデータのどちらかが主でどちらかが従であるということはない．

本節で示したように，Dualflow は，データ駆動的性質を持つ実行モデルを採用しているが，それは，動的命令スケジューリングのハードウェアを簡単化するためである．次節では，実際にこのことがどのようにハードウェアを簡単化するのかを説明する．

3.2 Dualflow の実装

本節では，Dualflow の基本的な実装方法について説明し，前節で述べた実行モデルが実装におよぼす効果を明らかにする．本節では主に，動的スケジューリングを行うロジックについて述べる．それ以外の部分は SS と同じと考えてよい．

3.2.1 プレース・ウィンドウ

まず，プレース列上での実行をハードウェア上に実現するために，プレース列に対する有限のウィンドウを考える．実行を完了したプレースがウィンドウ上から順次削除されることによって，ウィンドウのエントリはサイクリックに再利用される．

プレース・ウィンドウのサイズ *WS* は実装依存であるが，その最小値は命令フォーマット中の offset フィールドの幅から定まる．例えば offset が 5b であるとすると，*WS* は 32 以上となる．

Dualflow の実装は，原理的には，このプレース・ウィンドウをそのままハードウェア化すればよい．ただしもちろん，単に 1 個のメモリを用いたのでは，ポートの数が多くなりすぎて現実的ではない．次項では，現実的な実装方法については述べる．

3.2.2 プレース・ウィンドウの実装

さて，out-of-order SS は，リオーダ・バッファを用いる方式と，物理レジスタ・ファイルを用いる方式に

大別されると述べた．プレース・ウィンドウは，実際には後者に似た実装される．すなわちプレース・ウィンドウは，**命令キューと（物理）レジスタ・ファイル**によって実現される．ウィンドウの，命令スロットは命令キューに，データ・スロットはレジスタ・ファイルに，それぞれ格納される．

図3 に，プレース・ウィンドウのブロック図を示す．

命令キューとレジスタ・ファイルは，SS と同様に，適当に複製し，チップ中の然るべき場所に分散配置することが望ましい．命令キューは実行ユニットごと，レジスタ・ファイルは整数系と浮動小数点数系ごとに複製することが順当であろう．

複製された各命令キュー，および，各レジスタ・ファイルでは，同一の ID をもつエントリは 1 つのプレースによって占有されるという点に注意する必要がある．例えば，整数レジスタ・ファイルのあるエントリが使用された場合には，浮動小数点レジスタ・ファイルの同一の ID を持つエントリは（他のプレースによって）使用されることはない．また，SS では，例えば整数命令キューには整数系の命令だけが詰めて置かれるが，Dualflow では，他の命令キューに置かれる命令に対応するスロットを空けておく必要がある．

3.2.3 動的命令スケジューリングの処理

前述した動的命令スケジューリングの 3 つの処理は，Dualflow では以下のようになる．やはり I_d から I_u にデータが受け渡されるものとする：

(1) ***rename*** 命令がフェッチされると，タグへの変換が行われる．

I_d に対して，タグを求める．Dualflow では，宛先データ・スロットの ID がタグに相当する．宛先は，自命令と宛先スロット間の変位として示される．したがってタグは，単に自命令が格納されるエントリの ID に変位を加算することによって求めることができる．加算器の幅は $\log_2 WS$，すなわち 5〜6b 程度である．

I_u に対しては，タグを求める必要はない．Dualflow では，バッファは I_d ではなく I_u に割り当てられる．すなわち，ウィンドウのエントリにはバッファとして用いるデータ・スロットも含まれるため，命令がフェッチされるとバッファも自動的に割り当てられる．したがって，SS のような特別な割り当て処理は必要ない．

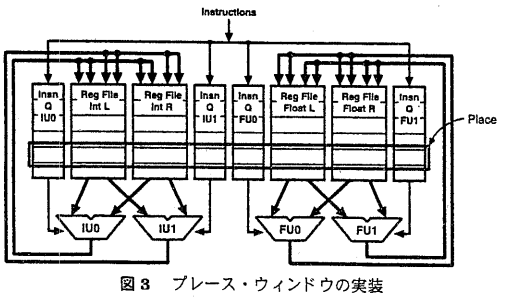

図3 プレース・ウィンドウの実装

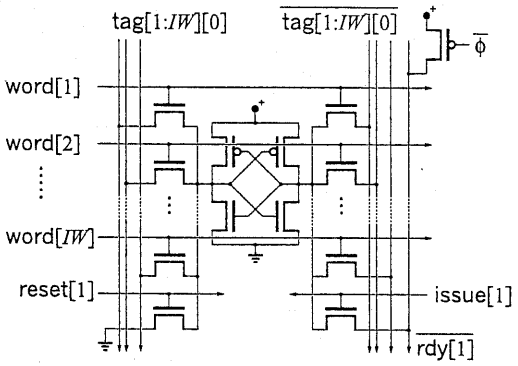

図4 Dualflow の *wakeup* ロジックのセル

(2) ***wakeup*** I_d の実行にともなって，I_u が実行可能になることを検出する．

I_d が実行されると，SS と同様に，その結果がタグで示されるデータ・スロットに書き込まれ，データ・スロットはオペランドが『ある』状態となる．

I_u は，オペランドが『ある』ことを見て，発行可能な状態になる．SS ではタグで示されるバッファを見るが，Dualflow では，タグで示されるバッファではなく，自身のデータ・スロットを見る．

(3) ***select*** SS と全く同様に，実行可能になった命令から実際に発行するものを選択する．

前述したように，*wakeup* において I_u が，タグで示されるのではなく，自身のデータ・スロットを見るという性質によって，Dualflow の *wakeup* ロジックは大幅に簡略化できる．次章では，*wakeup* ロジックについて詳しく述べる．

4. Dualflow の *wakeup* ロジック

本章では，本稿の主眼である，Dualflow の *wakeup* ロジックについて詳述する．

4.1 Dualflow *wakeup* ロジック

Dualflow の *wakeup* ロジックは，基本的には，命令ウィンドウの各エントリごとに，当該命令の宛先情報，すなわち，タグを格納したテーブルである．

3.2.3 項では，タグは，*rename* ステージにおいて，自命令が格納されるエントリの ID に変位を加算することによって求めると述べた．しかし実際には，*wakeup* の高速化のため，$\log_2 WS$ ビットの加算結果を，更に WS b のビット・ベクトルにデコードしておく．1 つのビット・ベクトルは 1 つの命令に対応し，ワード中の第 i ビットは当該命令の結果がウィンドウの第 i 番エントリに送られることを示す．1 つの命令の宛先は最大 2 つであるから，1 つのビット・ベクトルでは，最大 2b が “1” であり，残りは “0” である．ビット・ベクトルは，命令がウィンドウに格納されると同時に，*wakeup* ロジックに書き込まれる．

表1 Tag Read と Dualflow *wakeup* の比較
Table 1 Comparison between Tag Read and Dualflow*wakeup*

	Tag Read	Dualflow *wakeup*
ビット幅	*TAGW*	*WS*
ライト・ポート数	*IW*	*IW*
リード・ポート数	*IW*	1
リセット・ポート数	0	1
差動出力	可	不可

wakeup ロジックは，基本的には，*WS* b×*WS* word の RAM となる．**図4** に，*wakeup* ロジックのセルのを示す．セルは，ビット・ベクトルを書き込むための *IW* 本の書き込みポートと，1本のリセット用のポート，そして，1本の読み出しポートを持つ．

wakeup の処理は，単にこの RAM を読み出すことによって行うことができる．読み出しポートのビット・ライン $\overline{\text{rdy}}$ は，high にプリチャージされ，issue がアサートされたときに，当該セルがセットされていれば，プルダウンされる．

wakeup 全体では，最大 *IW* 本の issue ラインが同時にアサートされ，各 $\overline{\text{rdy}}$ ラインには，issue に対応する *IW* 個のセルが接続される．issue がアサートされたワードに対して bit-wise の OR をとることになる．ただし，正しいプログラムでは同一の宛先に複数の実行結果が送られることはないから，各 $\overline{\text{rdy}}$ ラインに対してたかだか1つのセルの出力が low となり，そのセルによってプルダウンが行われる．

もちろん，同時に複数のセルによってプルダウンされたとしても，物理的には問題ない．プログラムの間違いの検出は，物理レジスタ・ファイルへの実行結果の書き込み時などに行うことができる．

$\overline{\text{rdy}}$ ラインの電位を，センス・アンプによって増幅し，最終的な出力を得る．

4.2 Suprescalar の *wakeup* ロジックとの比較

前述のように，SS の *wakeup* ロジックの遅延は，Tag Read，Tag Drive，Tag Match，Match OR に分解できる．Dualflow の *wakeup* ロジックは，ちょうど Tag Read と比較することができる．

Tag Read と Dualflow の *wakeup* ロジックの遅延は両方とも，基本的には *WS* word の RAM の読み出しの遅延である．ただし，それぞれの RAM は**表1**に示す点が異なる．

Dualflow *wakeup* では，通常の RAM とは異なり，差動出力が得られない☆．そのため，ノイズ・マージンの点で不利だが，この程度の大きさの RAM ではその影響は大きくないと思われる．

Dualflow の *wakeup* は，Tag Read に比べて，ビット幅が広い分だけ，ビット線の遅延は長くなる．ただしビット線の遅延は，ビット線を分割することなどによって短縮することが可能である．

ビット幅よりは，ポート数の影響の方がより大きいと思われる．セルのサイズには，ポート数の2乗に比例する成分があるため，ポート数が多くなると急激に増大し，配線遅延を増大させる[3]．Tag Read では，Dualflow *wakeup* より，$IW-2$ 本のポートが余計に必要である．

また，図1から分かるように，Tag Read では，CAM 部分との整合性のため，レイアウト上の制約が強い．

以上から，Dualflow の *wakeup* の遅延は SS の Tag Read の遅延と同程度であり，Tag Drive，Tag Match，Match OR を考え合わせると，全体としては SS の *wakeup* の遅延より大幅に短縮できると予想される．

5. おわりに

Dualflow は，ISA のレベルからデータ駆動的性質を導入することによって動的命令スケジューリングのロジックを大幅に簡単化できるため，クロック速度の点で SS より有利である．

本稿では，双方の *wakeup* ロジックのトランジスタ・レベルの回路図を示し，その遅延の差を，定性的に評価した．

今後は，Hspice によるシミュレーションなどを行って，SS と Dualflow のクロック速度の差を定量的に評価を行っていく予定である．

謝辞

本研究の一部は文部省科学研究費補助金，基盤研究(B)(2) #12480072，同#12558027 による．

参考文献

1) Palacharla, S., Jouppi, N. P. and Smith, J. E.: Quantifying the Complexity of Superscalar Processors, Technical report, Univ. of Wisconsin-Madison (1996).
2) Palacharla, S., Jouppi, N. P. and Smith, J. E.: Complexity-Effective Superscalar Processors, *ISCA24* (1997).
3) Yamada, K., Lee, H., Murakami, T. and Mattausch, H. J.: An Area-Efficient Circuit Concept for Dynamical Conflict Management of N-Port Memories with Multi-GBit/s Access Bandwidth, *24th European Solid-State Circuit Conf.*, pp. 140–143 (1998).
4) 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).
5) 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).

☆ もちろん，ダミーのビット線をひくことはできる．