Dualflow アーキテクチャの命令発行機構

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[†]

Superscalar 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ソース・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．*wakeup* は，RAM を読み出した結果で CAM にアクセスするという構造を持ち，配線遅延に支配されるため，LSI の微細化にともなってクリティカルになると予測されている．Dualflow は，制御駆動とデータ駆動の性質をあわせ持つ命令セット・アーキテクチャであり，Superscalar と同様の out-of-order 実行を行いながら，*wakeup* を大幅に簡単化することができる．本稿では，Dualflow の *wakeup* を単一の RAM によって実現する方法を示す．双方のロジックのトランジスタ・レベルの回路図からその遅延の差を定性的に評価した結果，Dualflow の *wakeup* は，Superscalar の *wakeup* から CAM を省略するのと同程度の高速化を達成できることが分かった．

Instruction Issue Logic of the Dualflow Architecture

Masahiro Goshima,[†] Nguyen Hai Ha,[†] Akiyoshi Agata,[†]
Yasuhiko Nakashima,[††] Shin-ichiro Mori,[†]
Toshiaki Kitamura[†††] and Shinji Tomita[†]

A superscalar has logic called *wakeup*. The logic manages availability of the source operands of instructions in the instruction window for dynamic instruction scheduling. The delay time of *wakeup* consists of read delay time of a RAM and match access delay time of a CAM. Since the delay of the logic is dominated by the wire delay, it will be more critical with smaller feature sizes. Dualflow, a hybrid instruction set architecture between control- and data-driven, can realize out-of-order execution similar to superscalars with greatly simpler *wakeup* logic than superscalars. In this paper, we give single-RAM implementation of the Dualflow *wakeup*. The qualitative evaluation with transistor-level schematics of the logic shows that the delay time of the implementation is shorter than that of superscalars approximately by the delay time of the CAM.

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．

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ために，命令ウィンドウ中で待っている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ロジックである．

wakeup は，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．これを，RAM→CAM と表すことにする．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・ステージを割り当てることが難しい．以上の理由により *wakeup* 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．

† 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
††† 京都大学総合情報メディアセンター
Center for Information & Multimedia Studies, Kyoto University

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[2]．命令間のデータの受け渡しは， CtD のようにレジスタを介して間接的に指定されるのではなく，データ駆動のように生産側の命令が消費側の命令を直接的に指定することで行われる．

Dualflow では，この，命令間のデータ依存関係を直接的に指定するという性質によって，SS と同様の out-of-order 実行を行いながら，*wakeup* の複雑さを大幅に軽減することができる．前述のように SS の *wakeup* は，RAM→CAM という構造を持つが，Dualflow では，この CAM を RAM に置き換え，RAM→RAM とすることができる[3]．

本稿では実装上の工夫により，さらに *wakeup* を高速化する方法について述べる．具体的には，SS では RAM→CAM として実装される *wakeup* 全体を，単一の RAM で実装する方法を示す．

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る．そして 4 章で，Dualflow の *wakeup* について詳しく述べ，SS の *wakeup* との比較を行う．

2. Superscalar の *wakeup*

SS 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すべての遅延は，*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．

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 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[4]．

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に働かず，その中でも特に *wakeup* と呼ばれるロジックが LSI の微細化にともなってクリティカルになると予測される．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，SS の動的命令スケジューリングの原理についてまとめ，同時に *wakeup* の役割を明確にする．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．そして 2.3 節で，*wakeup* 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，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．

SS における動的命令スケジューリングは，このバッファを用いて，局所的にデータ駆動型の計算を行うことと見なすことができる．先行する命令 $\boldsymbol{I_d}$ が定義する結果を後続の命令 $\boldsymbol{I_u}$ が使用する場合を考えよう．I_d から I_u 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**への変換が行われる．

$\boldsymbol{I_d}$ には，その実行を結果を保存するため，バッファの 1 エントリが割り当てられる．このエントリの ID——タグを tagD ということにする．エントリはデータが『ない』状態に初期化される．

$\boldsymbol{I_u}$ は，左/右のソース・オペランドの論理レジスタ番号から，左/右それぞれに対し，依存する I_d 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_u 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_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．

$\boldsymbol{I_d}$ 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．

$\boldsymbol{I_u}$ は，tagL/R で示されるエントリにデータが『あ

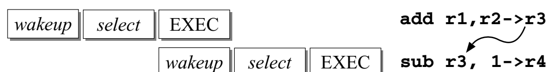

図 1 *wakeup* と *select* のパイプライン化
Fig. 1 Pipelining *wakeup* and *select*.

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 SS 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 の結果を使用する 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 である演算器——すなわち，通常は ALU——からのオペランド・バイパスをいっさい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．このような観点から，*wakeup* と *select* は，実際上，合わせて 1 サイクルで実行しなければならないとできる．

2.3 Superscalar の *wakeup*

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が，それらのうちでは *wakeup* がよりクリティカルになると予測されている．そのため，本節以降では *wakeup* について詳しく述べる．*wakeup* がクリティカルになる理由については，2.4 節で詳しく述べる．

まず，I_u の実行に必要な左/右のソース・オペランドに対応するバッファに，データが『ある』ことを表すフラグ rdyL/R のテーブルを考える．*wakeup* の役割は，I_d の実行にともなってこのテーブルを更新することである．

SS では，このテーブル rdyL/R は CAM を用いて実装される．**図 2** 左に，SS *wakeup* のブロック図を示す[1]．回路の上半分は tagD を記憶する RAM で，下半分が問題の CAM である．

SS *wakeup* は，発行された I_d の tagD を RAM から読み出し，それをキーとして rdyL/R を記憶する CAM にアクセスするという構成を持つ．したがってこれを RAM→CAM 方式と呼ぶことにする．

CAM の入力側は tagD/L/R をキーとする *IW* 本の比較入力ポートであり，入力された tagD と一致する tagL/R に対応する rdyL/R をセットする．出力側は，ポートを持たず，rdyL/R を記憶する $2 \cdot WS$ 個のセルからの出力線が，マルチプレクスされることなく *select* に接続されている．

2.4 Superscalar の *wakeup* の遅延

図 2 右の回路図は，同図左のブロック図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 RAM のセルである．この RAM の語構成は，tagD/L/R のビット幅（5〜7 程度）を ***TW*** として，*TW*b × *WS*word となる．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．各読み出しポートは，その下に描かれている CAM の比較入力ポートに直接接続されている．

回路の下半分は，tagD/L/R をキーとする *TW*b × $2 \cdot WS$ word の CAM のセルである．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 tagD との比較器（1b 分）である．

この回路の遅延は，以下の 4 つに分解できる：

1. **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．
2. **Tag Drive** *IW* 個の tagD を，縦に引かれたビット線によって $2 \cdot WS$ 個の CAM セルに放送する．
3. **Tag Match** 放送された tagD と，tagL/R を比較する．*TW*b 比較器は，*TW* 個のセルにわたって引かれたマッチ線に対する wired-AND として実現されている．
4. **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 2 から，Tag Read，Tag Drive，Tag Match のロジックは，主に，ゲートではなく，配線によって構成されていることが分かる．

Palacharla らは，SS の各ロジックの詳細なモデリングを行ったうえで，Spice を用いてそれらの遅延時間を見積もっている[1]．そのうち，*wakeup* と *select* に関する結果を**図 3** に示す．

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

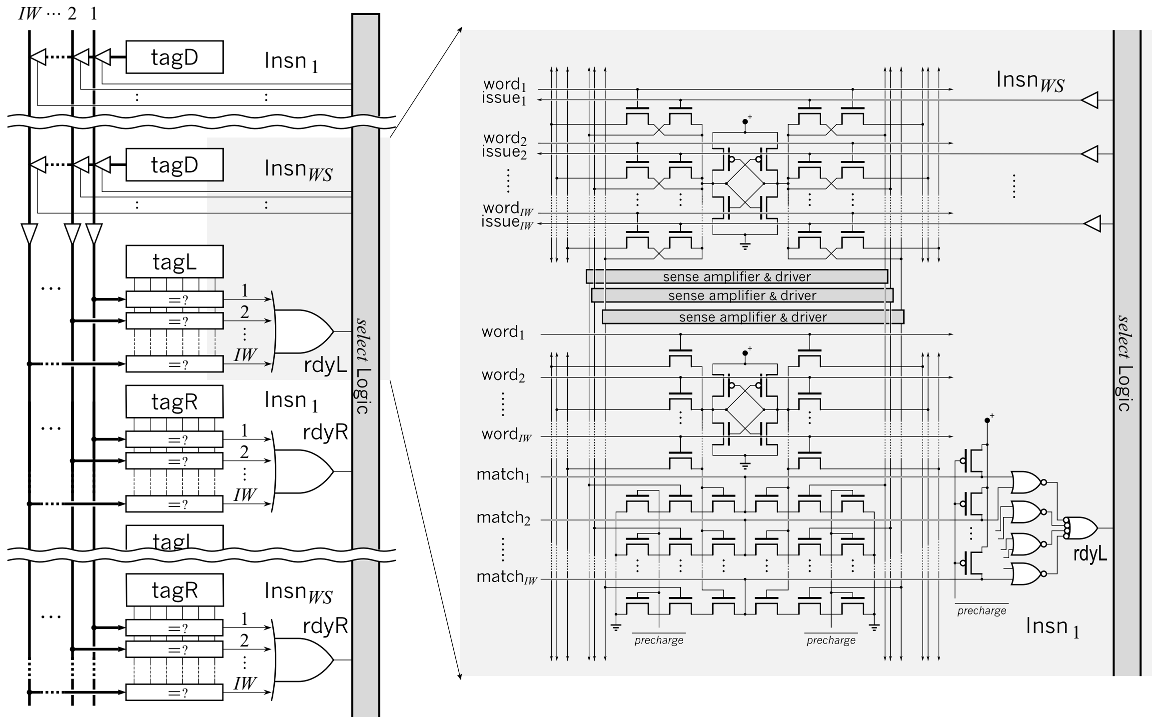

図 2　Superscalar のウィンドウ・ロジック
Fig. 2　Window logic of a superscalar.

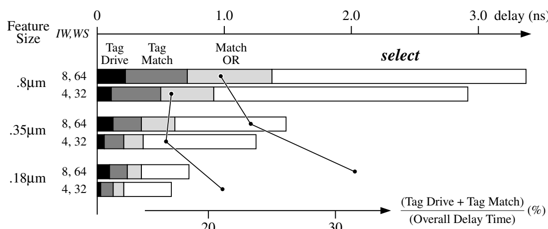

図 3　*wakeup* と *select* の遅延
Fig. 3　Propagation delay time of *wakeup* and *select*.

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．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．

また，同グラフからは，*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．それは，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[1]．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[5] 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大幅に短縮されるわけではない．

なお，Palacharla らは Tag Read に関する考察を行っていないが，その遅延も無視することはできない．Tag Read は TagD を格納する RAM の読み出しの遅延であり，Tag Drive，Tag Match と同様，配線遅延に支配されている．図 2 に示されるワード線長，ビット線長から，配線遅延が支配的になった領域では，Tag Read 全体の遅延は Tag Drive+Tag Match と比較できる程度になると予想される．

3. Dualflow アーキテクチャの概要

本章では，Dualflow アーキテクチャについて概説する．まず 3.1 節で実行モデルについて述べ，3.2 節で主に命令ウィンドウの実装方法についてまとめる．問題の *wakeup* ロジックについては，次章で詳細に述べる．

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あわせ持つ：

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．

以下では，まずモデルの全体像について述べた後，

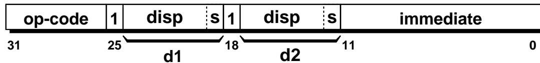

図 4 命令フォーマットの例
Fig. 4 Example of instruction format.

実行例を用いて説明を行う．

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．**図 4** に命令フォーマットの例を示す．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 d_n フィールドがある．d_n フィールドについては後で詳しく述べる．

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 d_n フィールドが示す後続の命令に送りつける．そのため，命令中にはデータの送り元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・カウンタに従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．図 4 に示した命令フォーマットにおいて，前述した d_n フィールドは，正確には，宛先の命令ではなく，宛先のデータ・スロットを示す．そのため d_n フィールドは，自命令と宛先データ・スロット間の変位を表す disp サブフィールドのほか，宛先データ・スロットの左右を表す s サブフィールドからなる．

```
line label  instruction
 1          imm   a    2L
 2          imm   b    1R
 3          sub        1L, 2L
 4          bneg NEG
 5          mov        2L
 6          b END
 7    NEG:  subr 0     1L
 8    END:  mov        X
```

図 5 |a − b| を計算するプログラム
Fig. 5 Program to calculate |a − b|.

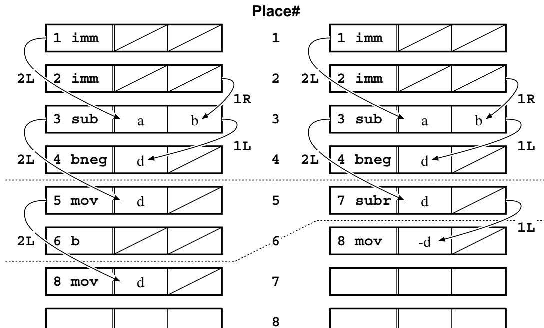

図 6 図 5 のプログラムを実行後のプレース
Fig. 6 Places after execution of the program in Fig. 5.

なお現在では，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．

実　行　例

では，**図 5** に示す |a − b| を計算するプログラムを例に，Dualflow の動作を具体的に説明しよう．このプログラムは，まず，3 行の `sub` 命令で d = a − b を求める．そして d が負である場合には 4 行の `bneg` 命令から `NEG` に分岐し，さらに 0 − d を計算して最終的な結果とする．

図 6 に，実行後のプレース列の様子を示す．プログラムは以下のように実行される：

(1) 最初プログラム・カウンタは 1 行を指している．この時点で 4 行の条件分岐命令 `bneg` までの制御の流れは確定しているので，1〜4 行の各命令を，プレース 1〜4 の命令スロットにそれぞれ格納することができる．

(2) 1/2 行の `imm` 命令は即値を生成する命令で，データを必要としない．したがってフェッチ後ただち

に実行されて，値 a/b を 2L/1R で示されるプレース 3 の左/右データ・スロットに送る．
(3) プレース 3 は，3 行の sub 命令と，1/2 行の imm 命令からのデータの到着によって実行可能となり，実行結果 d は 1L/2L で示されるプレース 4/5，それぞれの左データ・スロットに送られる．
2L で示されるプレース 5 に入る命令は，条件分岐命令 bneg の結果に依存するので，この時点ではフェッチできないことに注意されたい．したがってこの sub 命令は，**そこにどのような命令が来るかにかかわらず，プレース 5 に実行結果を送りつけることになる**．
その一方でプレース 5 は，命令より先にデータを受け取ることになる．これは，データ駆動ではありえないことであるが，SS ではごく普通のことである．
(4) プレース 5 の命令スロットには，bneg が not taken であれば 5 行の mov が，taken であれば 7 行の subr が格納される．
以降は taken であった場合（図 6 右）について説明する．この時点で以降の制御の流れは確定する．
(5) subr は，sub とは逆に，右オペランドから左オペランドを減ずる命令である．この場合は即値 0 を持っているので，0 − d を計算する．subr がプレース 5 の命令スロットに格納される時点で，データ d はすでに到着しているので，このプレースはフェッチ後ただちに実行される．
(6) 結果 −d はプレース 7 の mov 命令によって X に送られる．4 行の条件分岐 bneg によって実行命令数が異なり，この mov 命令が格納されるプレースも異なる．

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いわれる．そのような観点からいえば，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．

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する．本節では主に，動的スケジューリングに関連する命令ウィンドウの周辺について述べる．それ以外の部分は通常の SS と同じと考えてよい．

3.2.1 プレース・ウィンドウ

まず，図 6 に示した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．たとえば offset が 5b であるとすると，*WS* は 32 以上となる．

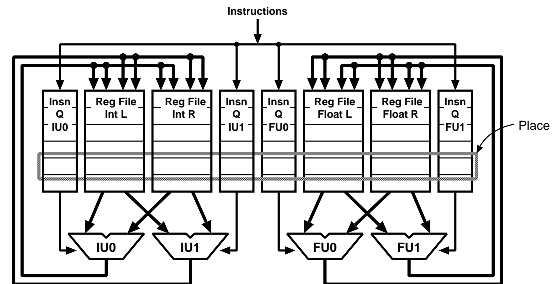

図 7 プレース・ウィンドウの実装
Fig. 7 Implementation of place window.

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 個のメモリを用いて実現したのでは，ポートの数が多くなりすぎて現実的ではない．現実的な実装方法について次項で述べる．

3.2.2 プレース・ウィンドウの実装

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・ウィンドウに対しても，この 2 つの方式に対応する実装方法が考えられる．

以下では後者に似た実装について述べる．すなわちプレース・ウィンドウを，**命令キュー**と**（物理）レジスタ・ファイル**によって実現する．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 7 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，チップ中のしかるべき場所に分散配置することが望ましい．たとえば，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．

ただし，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 を持つエントリは 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．たとえば，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，たと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．

そのため，レジスタ・ファイルの利用効率は悪化す

るが，容量の増加によってそれを補うことができる．レジスタ・ファイルのハードウェア量自体が問題視される可能性は低い．また，レジスタ・ファイル・アクセスをパイプライン化しても 2%程度の性能低下しかみられないという報告もあり[6]，容量の増加が性能に与える影響は大きくないと考えられる．

4. Dualflow の *wakeup*

本章では，本稿の主眼である，Dualflow の *wakeup* について詳述する．2.3 節では，SS の *wakeup* は RAM→CAM 方式によって実装されると述べた．Dualflow では，この CAM を RAM に置き換え，RAM→RAM として実装することができる[7]．本稿では，RAM→RAM をさらに単一の RAM に置き換える方法を示す．

まず 4.1 節で Dualflow の動的命令スケジューリングについてまとめた後，4.2 節で RAM→RAM によって構成できることを示す．そして，4.3 節で単一の RAM によって実現する方法について説明する．

4.1 Dualflow における動的命令スケジューリング

2.1 節で述べた SS の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．

$\boldsymbol{I_d}$ に対して tagD 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．自命令と宛先スロット間の変位は disp サブフィールドで示されている．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 disp の値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 $\log_2 WS$ (5〜7) b である．

$\boldsymbol{I_u}$ に対してはタグを求める必要はない．Dualflow では，バッファのエントリは I_d ではなく I_u に割り当てられる．命令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，I_u 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．SS のような特別な処理は必要ない．

(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．

$\boldsymbol{I_d}$ が実行されると，SS と同様に，その結果が tagD 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．

$\boldsymbol{I_u}$ 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 tagL/R で示されるバッファを見るが，Dualflow では，I_u 自身に割り当てられたデータ・スロットを見る．

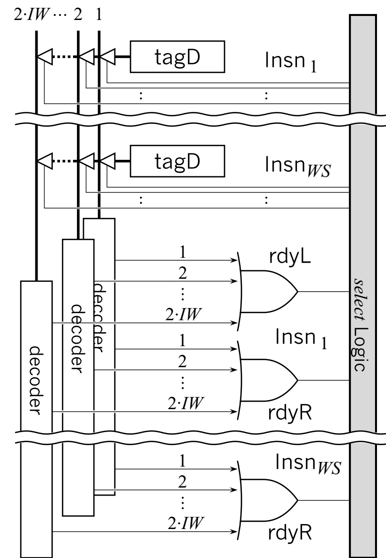

図 8 Dualflow *wakeup*（RAM→RAM）
Fig. 8 Dualflow *wakeup* (RAM→RAM).

(3) *select* SS とまったく同様に，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．

このように Dualflow では，*select* は SS と同じだが，*rename* と *wakeup* が簡単化される．*rename* は，分岐予測ミス・ペナルティに影響する．*rename* が簡単化された分だけ，デコード・ステージが短縮され，その分だけ分岐予測ミス・ペナルティが削減される．一方 *wakeup* の簡単化は，クロック速度の向上につながる．次章節以降では，*wakeup* について詳しく述べる．

4.2 Dualflow *wakeup*

Dualflow においても，SS と同様，I_u の実行に必要な左/右のソース・オペランドに対応するバッファにデータが『ある』ことを表すフラグ rdyL/R のテーブルが *wakeup* の本体となる．

2.3 節で述べたように，SS ではこのテーブルを CAM を用いて実現する．一方 Dualflow では，このテーブルを RAM を用いて実現することができる．Dualflow における tagD は，I_u に割り当てられたバッファのエントリを直接指定するものである．したがってテーブル rdyL/R を RAM として実現し，その更新は tagD をアドレスとして書き込みを行えばよい．すなわち Dualflow の *wakeup* では，SS では CAM である部分を変則的な RAM に置き換え，RAM→RAM として実現することができる．

図 8 に，Dualflow *wakeup* のブロック図を示す．同図中では下左にあるデコーダは，rdyL/R を格納する RAM の行デコーダにあたる．

表 1　ビット・ベクトルの例
Table 1　Examples of bit-vectors.

宛 先	ビット・ベクトル
1L	L:0 · · · 001，R:0 · · · 000
1L，2R	L:0 · · · 001，R:0 · · · 010
1L，2L	L:0 · · · 011，R:0 · · · 000

このロジックの処理は，以下の 3 つに分解できる：

1. **Tag Read**　SS の Tag Read と同様，1 つ目の RAM から tagD を読み出す．
2. **Tag Decode**　入力された $2 \cdot IW$ 個の tagD をデコードする．デコーダの出力が SS *wakeup* における match 線に対応する．
3. **OR**　1 つの rdyL/R に対応する $2 \cdot IW$ 本のデコーダの出力線を OR する．

4.3　Dualflow *wakeup* の改良

前節では Dualflow *wakeup* は RAM→RAM として実現できることを示した．本節では，さらにこれを単一の RAM として実現する方法を示す．

前述のロジックでは，Tag Decode において tagD は 2 進数から**ビット・ベクトル**にデコードされている．しかし，このデコードは必ずしも *wakeup* において行う必要はない．基本的な着想は，このデコードを命令パイプラインのフロント・エンド部で済ませておくということである．あらかじめデコードしておけば，実行される命令のビット・ベクトルを OR するだけで rdyL/R を求めることができる．

このビット・ベクトルを OR するロジックは，ビット・ベクトルを格納する変則的な RAM として実現できる．以下ではこのことについて詳しく説明する．

ビット・ベクトル

まず，命令の実行結果の宛先を表す**ビット・ベクトル**をより厳密に定義しよう．1 つのビット・ベクトルは，命令の $\mathrm{d}_{1:2}$.disp フィールドをデコードし，それぞれ宛先の左/右ごとに論理和をとったものである．$\mathrm{d}_{1:2}$.disp は各 5b であり，1 つのビット・ベクトルは 31b である．**表 1** に，ビット・ベクトルの例を示す．なお，前述のロジックでは，自命令が格納されるウィンドウのエントリ ID に disp を加算して tagD を求めたが，ビット・ベクトルを求める場合には加算は必要ない．

ロジックの構成

図 9 に，Dualflow *wakeup* の構成を示す．

同図は rdyL を求める部分であり，rdyR のために図とまったく同じロジックがもう 1 つ必要である．rdyL と rdyR はそれぞれ並列に求められるため，遅延を考える上ではどちらか一方に着目すればよい．なお，*select* は，同図の rdyL 用のロジックの右側に配置し，rdyR

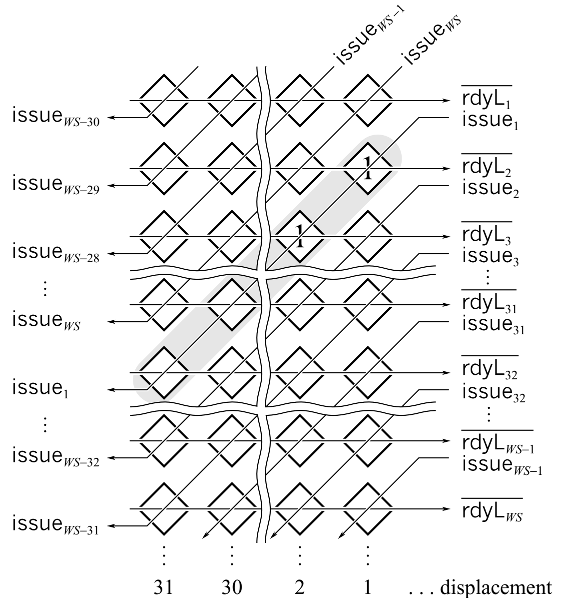

図 9　Dualflow の *wakeup*（左オペランド部）
Fig. 9　Dualflow *wakeup* (left operand part).

用は *select* を挟んで rdyL 用と鏡面対称となるように配置するとよいだろう．

このロジックは，端的にいえば，スキューイングした RAM である．同図中，◇ は 1b の RAM セルを表している．網掛けで示した部分は，エントリ ID=1 の命令に対するビット・ベクトルを格納する 31b 分の RAM セルであり，宛先 1L，2L に対応するビット・ベクトル，L:0 · · · 011 が格納されているところを表している．同図のロジック全体を 1 つの RAM と見た場合，斜めに走る issue がワード線に，横に走る $\overline{\mathrm{rdyL}}$ がビット線に対応する．先に，ビット・ベクトルを求める際には加算が必要ないと述べたが，それはこのスキューイングがその役割を果たすからである．

このロジックでは，RAM セルは 31b × *WS* word 分あるが，issue，$\overline{\mathrm{rdyL}}$ ともに，*WS*b ではなく，31b 分にしか接続されていない点に注意されたい．

なお，同図には読み出しポート部分のみを示してあり，これとは別にビット・ベクトルを書き込むための *IW* 本の書き込みポートが必要である．書き込みポートでは，ワード線は issue と平行に，ビット線は縦方向に引かれる．読み出しポートとは異なり，書き込みポートのビット線は，通常どおり *WS*b 分の RAM セルに接続される．

ロジックのセル

図 10 に，RAM セルの詳細な回路図を示す．このセルは，図 9 中の右上端（第 *WS* 番命令の第 1b）の

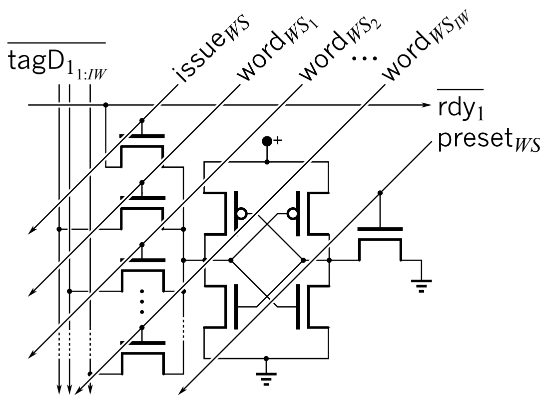

図 10　Dualflow の *wakeup* のセル
Fig. 10　Dualflow *wakeup* Cell.

ものである．1 つのセルは，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 本のプ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．

Dualflow *wakeup* では，通常の RAM とは異なり，差動出力が得にくい．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考えられる．ノイズが問題となる場合には，ビット線と平行に，つねに high にチャージされたダミーのビット線をひくことで対処することもできるであろう．

ロジックの動作

各ビット・ベクトルは，$\text{d}_{1:2}$.disp からデコードされたものである．したがって，このロジックにおけるエントリの解放と再利用の方法は，図 8 に示したロジックにおけるエントリ（tagD）のそれとまったく同じでよい．

ビット・ベクトルは，命令パイプラインのデコード・ステージにおいて $\text{d}_{1:2}$.disp からデコードされ，命令が命令ウィンドウに格納されると同時にこのロジックに書き込まれる．

wakeup の処理は，単にこのロジックを RAM として読み出すことによって実現される．ビット線 $\overline{\text{rdyL}}$ は，あらかじめ high にプリチャージされ，ワード線 issue がアサートされたときに，交差点上のセルがセットされていればプルダウンされる．$\overline{\text{rdyL}}$ 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．

たとえば，図 9 に示した状態でエントリ ID=1 にある命令が発行された場合を考えよう．この命令の宛先は 1L，2L，すなわち，ID=2，3 のエントリである．issue_1 がアサートされると，網掛けで示した RAM セルからビット・ベクトル L:0···011 が読み出され，その結果 $\overline{\text{rdyL}_2}$と $\overline{\text{rdyL}_3}$がアサートされる．

ロジック全体では，毎サイクル（最大）*IW* 本の issue がアサートされ，各 $\overline{\text{rdyL}}$ 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る．$\overline{\text{rdyL}}$ の値は，それらのセルの出力を横方向に OR をとったものとなる．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，OR をとるといっても各 $\overline{\text{rdyL}}$ に対して出力が low となるセルはたかだか 1 つである．プログラムのエラーによって複数のセルの出力が low となることがありうるが，物理的には問題ない．なお，エラーの検出は，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．

表 2　Superscalar Tag Read と Dualflow *wakeup*
Table 2　Supersclar Tag Read vs. Dualflow *wakeup*.

	SS Tag Read	Dualflow *wakeup*
ビット数	*TW*	31†
ワード数	*WS*	31
ライト・ポート数	*IW*	*IW*
リード・ポート数	*IW*	1
差動出力	可	不可

†: ただし，ワード線は斜め．

4.4　Suprescalar の *wakeup* との比較

2.3 節で述べたように，SS の *wakeup* の遅延は，Tag Read，Tag Drive，Tag Match，Match OR に分解できる．Dualflow の *wakeup* は，これらのうち，Tag Read と比較することができる．SS Tag Read と Dualflow *wakeup* の遅延は，双方とも，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 2** に示す点が異なる．図 2 右上の SS Tag Read の RAM セルと図 10 に示した Dualflow *wakeup* の RAM セルを比較されたい．SS Tag Read に対して，Dualflow *wakeup* の有利な点，不利な点は以下のようにまとめられる：

有利な点

レイアウトに対する自由度　図 2 からも分かるように，SS Tag Read では，CAM 部分との整合性のため，必ずしもその遅延が最小となるようにレイアウトできるわけではない．一方 Dualflow *wakeup* には，そのような制限はない．

ポート数　総ポート数は $IW - 2$ 本少ない．セル・サイズは，ポート数の 2 乗に比例する成分を持つため，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といわれている[8]．

ビット線長 ビット線 $\overline{\text{rdyL}}$ に接続される RAM セルは 31b 分であり，$WS \geq 32$ の領域では SS Tag Read のより少なく，しかも定数である．

また，上記のレイアウトとポート数の問題から，実際のビット線長の差はセル数の差以上に大きなものとなる．

不利な点

ワード線長 SS Tag Read に対しビット数が $31/TW$（4〜5 程度）倍ある．また，斜めであるため，ワード線 issue がかなり長い．

SS Tag Read に対する Dualflow *wakeup* の不利な点はワード線長のみであるが，ワード線長の全体の遅延に与える影響は大きくないため，Dualflow *wakeup* の遅延は SS Tag Read のそれと同程度以下であると予想される．

5. おわりに

Dualflow は，データ駆動的性質を導入することによって out-of-order 実行機構を大幅に簡略化するが，その代償として，コードはデータの授受に関する制約を受ける．それは，宛先フィールドを静的に計算するという制約である．制約は，以下の 3 つに分類できる：

数と距離 2 個を超えるスロット，あるいは，32 以上離れたスロットにはデータを送れない．

条件分岐 データの授受が条件分岐を越える場合にも，分岐の結果によって宛先を変えることはできない．

基本ブロック データの授受が 1 つ以上の基本ブロックを越える場合には，命令間の距離，すなわち，宛先フィールドの値を静的に求めるとができない．関数呼び出し，if-then-else 構造，ループなどを越えたデータの授受が，これにあたる．

それぞれの場合に対して，データを中継するための `mov` 命令を挿入して制約を満たす必要がある．

簡単なコンパイラを作成し，SPEC ベンチマークを用いて実行命令数を計測したところ，必要な最適化がほとんど実装されていないためもあり，有効な命令の 30〜150%にもあたる無用な `mov` 命令が実行されるという結果を得ている[7]．

また，

- プレース・ウィンドウを複数の命令キュー，物理レジスタ・ファイルで構成した場合に，同一 ID を持つエントリは同一の命令によって占有される，
- 命令の発行ではなく，完了までエントリを解放できない，

など，プレース・ウィンドウのエントリの利用効率が悪く，その点でも IPC が悪化する．

本稿では，*wakeup* を単一の RAM によって実現する手法を示した．SS の *wakeup* は RAM→CAM として実装されるが，Dualflow の *wakeup* の遅延は，この 1 つ目の RAM と同程度以下であると推測される．その場合，*wakeup* 全体では，RAM→CAM から CAM を省略した程度の高速化を達成することができる．*wakeup* の遅延は，おそらく半分以下になるであろう．

それでも現状では，クロック速度の向上に IPC の悪化が見合わない可能性が高い．今後は，Dualflow 特有の最適化などを行い，SS との IPC の差を縮めていく必要がある．

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) 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，情報処理学会研究報告，98-ARC-130, pp.115–120 (1998).
3) 五島正裕，グェンハイハー，縣 亮慶，森眞一郎，富田眞治：Dualflow アーキテクチャとそのコード生成手法，情報処理学会研究報告，99-ARC-134, pp.163–168 (1999).
4) Keller, J.: The 21264: A Superscalar Alpha Processor with Out-of-Order Execution, *9th Annual Microprocessor Forum* (1996).
5) Palacharla, S., Jouppi, N.P. and Smith, J.E.: Complexity-Effective Superscalar Processors, *ISCA24* (1997).
6) Tullsen, D.M., et al.: Exploiting Choice: Instruction Fetch and Issue on an Implementatable Simultaneous Multithreading Processor, *23rd. Annual Int'l Symp. on Computer Architecture*, pp.191–202 (1996).
7) 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).
8) 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).

(平成 12 年 8 月 31 日受付)
(平成 13 年 2 月 1 日採録)

五島　正裕（正会員）
1968年生．1992年京都大学工学部情報工学科卒業．1994年同大学大学院工学研究科情報工学専攻修士課程修了．同年より日本学術振興会特別研究員．1996年京都大学大学院工学研究科情報工学専攻博士後期課程退学，同年より同大学工学部助手．1998年同大学大学院情報学研究科助手．高性能計算機システムの研究に従事．

グェン ハイハー（学生会員）
1974年生．1993年ハノイ大学工学部退学．1999年京都大学工学部情報工学科卒業．2001年同大学大学院情報学研究科修士課程修了．プロセッサアーキテクチャの研究に従事．

縣　　亮慶（学生会員）
1977年生．1997年国立舞鶴工業高等専門学校卒業．1999年京都大学工学部情報学科卒業．2001年同大学院情報学研究科修士課程修了．プロセッサ・アーキテクチャの研究に従事．

中島　康彦（正会員）
1963年生．1986年京都大学工学部情報工学科卒業．1988年同大学大学院修士課程修了．同年富士通（株）入社．スーパコンピュータVPPシリーズのVLIW型CPU，Mアーキテクチャ・命令エミュレーション，高速CMOS回路等に関する研究開発に従事．博士（工学）．1999年京都大学総合情報メディアセンター助手．同年同大学大学院経済学研究科助教授，現在に至る．計算機アーキテクチャに興味を持つ．IEEECS，ACM各会員．

森　眞一郎（正会員）
1963年生．1987年熊本大学工学部電子工学科卒業．1989年九州大学大学院総合理工学研究科情報システム学専攻修士課程修了．1992年九州大学大学院総合理工学研究科情報システム学専攻博士課程単位取得退学．同年京都大学工学部助手．1995年同助教授．1998年同大学大学院情報学研究科助教授．工学博士．並列/分散処理，計算機アーキテクチャの研究に従事．IEEE，ACM各会員．

北村　俊明（正会員）
1955年生．1978年京都大学工学部情報工学科卒業．1983年同大学大学院博士課程研究指導認定退学．同年富士通（株）入社．汎用コンピュータ，スーパーコンピュータVPPシリーズのVLIW型CPU，Mアーキテクチャ・命令エミュレーション，米国HAL社においてSPARCプロセッサ等の研究開発に従事．博士（工学）．2000年京都大学総合情報メディアセンター助教授，現在に至る．計算機アーキテクチャに興味を持つ．電子情報通信学会，IEEE，ACM各会員．

富田　眞治（正会員）
1945年生．1973年京都大学大学院博士課程修了，工学博士．同年京都大学工学部情報工学教室助手．1978年同助教授．1986年九州大学大学院総合理工学研究科教授．1981年京都大学工学部情報工学科教授．1998年同大学大学院情報学研究科教授．計算機アーキテクチャ，並列計算機システムに興味を持つ．著書「並列計算機構成論」，「並列処理マシン」，「コンピュータアーキテクチャI」等．電子情報通信学会，IEEE，ACM各会員．