計算機アーキテクチャ 139−19
（2000．8．4）

Superscalarにおける低遅延な命令スケジューリング方式

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

従来のSS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．我々は，命令に宛先を埋め込むことでこのCAMをRAMに置き換えることができるアーキテクチャDualflowを提案してきた．本研究では，SSの命令をデコード時に宛先に変換することで，SSの命令互換性を保ったままDualflow並みの高速な動的命令スケジューリングを行う回路を提案する．本方法ではデコードステージが1サイクル伸びるが，それによる分岐ミスペナルティの増加は少ない．

A low-latency instruction scheduling scheme for superscalars

Masahiro Goshima,† Nguyen Hai Ha,† Akiyoshi Agata,†
Shin-ichiro Mori,† Yasuhiko Nakasima†† 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. We propose new architecture. By translating instructions on decode time, it has low-latency dynamic instruction scheduling. It require one more cycle on decode, but branch miss penalty hardly increase.

1. はじめに

Superscalar（以下，SSと略）のIPCを向上させる最も直接的な方法は，命令の発行幅（***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を向上させることができた．

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，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[3)~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，通常の制御駆動型計算機と同様，PCと分岐命令による制御の流れを持つが，制御駆動型のようなレジスタを定義しない．命令間のデータの受け渡しは，制御駆動のようにレジスタを介して行うのではなく，データ駆動のように命令間で直接的に行われる．命令が生産するデータの宛先となる命令は，両者の間の変位として，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[2)]．

しかし，消費側の命令を即値的に，すなわち，静的に指定することはコード生成上の制約となる．そのつじつまを合わせるため，本来無用な命令を挿入せざる

† 京都大学情報学研究科 Grad. School of Informatics, Kyoto U.
†† 京都大学経済学研究科 Grad. School of Economics, Kyoto U.

を得ない．最適化の程度によるものの，その命令数の増加は無視することができなかった[4]．

そこで本稿では，命令セット・アーキテクチャとしては通常の RISC SS と同様のものを用い，命令パイプラインのフロントエンドにおいて動的に Dualflow と同様の直接指定形式に変換することを考える．バックエンドで Dualflow と同様の実行を行うことができれば，命令数を増加させることなく，クロック速度を向上させることができるであろう．

しかしその一方で，フロントエンドでの変換の処理を行う必要があるので，そのための分岐予測ミス・ペナルティの増加が懸念される．

以下，2 章ではこの変換を行う回路の概要を説明し，3 章ではその動作を説明し，4 章ではゲートレベルの回路を示し，5 章でまとめと今後の課題を述べる．

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．

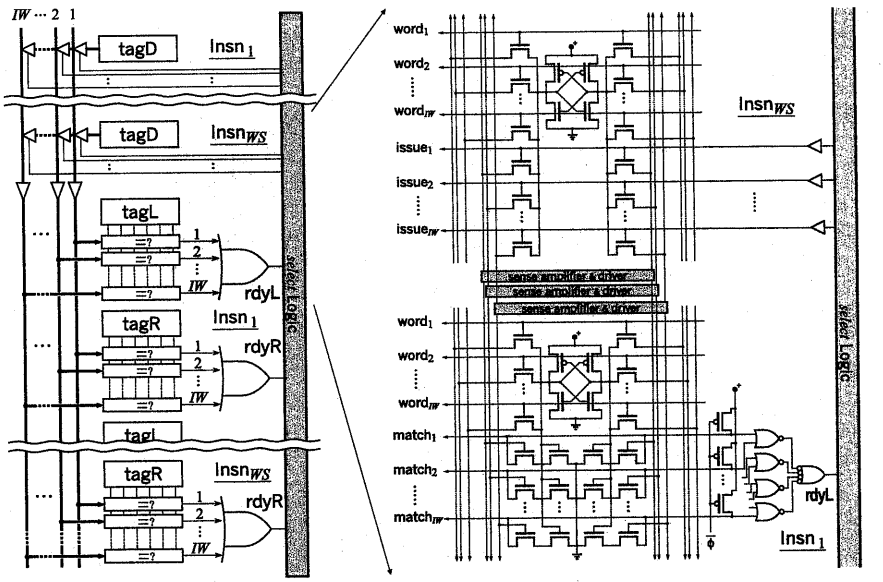

図 1 SS のウィンドウ・ロジック
Fig. 1 Window logic of superscalar

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．

2.3.0.1 *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．

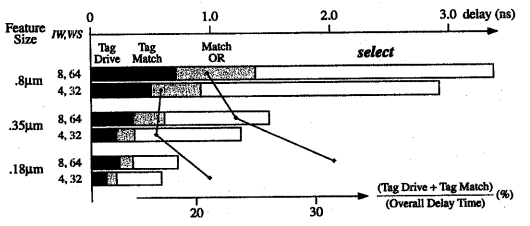

図 2 *wakeup* と *select* の遅延
Fig. 2 Propagation delay time of *wakeup* and *select*

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[7]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．

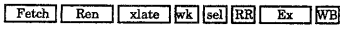

図3　パイプライン構成

3. 本 方 式

3.1 概　　要

3.1.1 アイデア

wakeup ロジックは、命令が実行されることで、それに依存する命令を検出するロジックであることは既に述べた。従来の SS は、発行した命令に依存する命令を *wakeup* の中で求めていたが、デコード時に命令ウィンドウ内の全命令間の依存関係を調べ、テーブルに入れれば、発行した命令に依存する命令を調べるには、テーブルを引くだけで済む。このテーブルは RAM に似た回路で実現でき、SS の CAM より高速である。

しかしそのためには、新しく命令が 命令ウィンドウに入る度に、依存関係を調べ、テーブルの内容を更新する必要がある。そこでこの処理を行うステージを設け、***xlate*** (translate) と名付ける。*xlate* では、新しく命令ウィンドウに入った命令と命令ウィンドウ内の全命令との依存関係を調べ、テーブルを更新する。

本方式では、このようにデコード時に依存関係を求めることで、*wakeup* を高速に行える。以下、本方式について説明する。

3.1.2 パイプラインステージ

この処理を行う場合、デコードに *xlate* ステージが増え、図 3のようなパイプライン構成になる。

以下、*xlate*, *wakeup* ステージについて述べ、最後にハザード回避のタイミングについて述べる。

3.2 *xlate* ステージ

xlate は、新しく命令ウィンドウに入る命令を元に、フラグを更新するステージである。

3.2.1 フラグの説明

フラグは、命令ウィンドウ内の命令間の依存関係を表す 2 次元のテーブルである。フラグの内容は、その行の命令の tagD と その列の命令の tagL が同じときだけ 1 である。つまり、n 行目 m 列目が 1 というのは、エントリ m のオペランドがエントリ n の命令の結果に依存しているということである。

この対角上にはフラグが存在しない．なぜなら，そのようなフラグは自分自身に依存していることを表すものであり，それは有り得ないからである．

3.2.2 フラグの更新内容

まず、$IW = 1$ の単純なモデルで、どのフラグをどう更新するかを説明する。

命令がエントリに入ると、フラグの行と列の両方を更新する必要があるように見えるが、実際は列だけで良い。なぜなら、新しい命令の tagD に依存する命令はまだ命令ウィンドウ内には無いからである。だから、命令に対応する列のフラグは、クリアすれば良いのだ

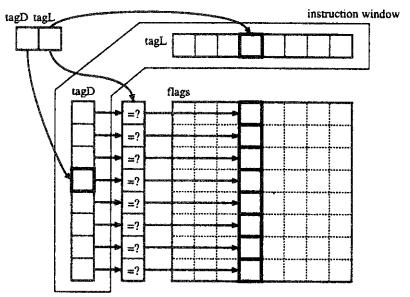

図4　フラグの更新処理

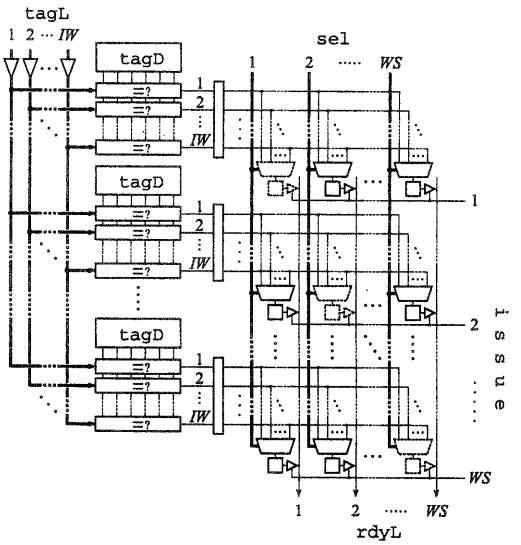

図5　提案方式のウィンドウ・ロジック

が、行はすでにエントリから命令が削除されるときにクリアされているので、そのままで良い。従って、更新すべきフラグは、新しい命令が入ったエントリの列だけである。

図 4に、更新処理の大まかな図を示す。左上にあるのは *rename* でリネーミングされた新しい命令で、そのステージ内で命令ウィンドウに追加される。

xlate ステージでは、この tagL と命令ウィンドウ内の全 tagD を比較し、その結果を新しい命令が入るエントリと同じ列のフラグに書き込む。

3.2.3 実際のモデル

図 5が実際のブロック図である。図右半分の、$WS \times WS$個の小さな正方形が、フラグである。フラグの先のゲートや、フラグ間の縦横の配線は次の *wakeup* で説明するので、今は無視していただきたい。基本は前節と同じだが、一度に複数の列を更新するため、比較器や配線が IW 個あり、フラグへの入力はセレクタ (台形の記号) を通るようになっている。

また、1 サイクルを前半後半 2 つのステージに分る

ため、左右をパイプライン・ラッチで分離している。パイプライン・ラッチの左が前半で、新しい命令の tagL と命令ウィンドウ内の全 tagD を tagD 下の比較器で比較し、パイプライン・ラッチに結果を書き込んでいる。

右側が後半で、パイプライン・ラッチに書いた比較結果はセレクタを通過してフラグに書き込まれる。このセレクタを制御するのが sel 信号であり、これにより、パイプライン・ラッチのどの比較結果をどの列に書くかを制御できる。

sel 信号について説明する。sel 信号は、セレクタを制御し、比較元命令の入ったエントリに対応した列に比較結果を入れることを目的とする。そのために、その tagL が入った列のセレクタの選択信号には，命令の *fetch* 時の位置をデコードしたものが入力される．

例えば、*fetch* 時に n 列目であった命令がエントリの m 番目に入ったとすると、*fetch* 時に n 列目であった命令の tagL との比較結果はフラグの m 列目に入る必要がある。このような場合、m 列目の sel 信号に、n をデコードした値を入れる。

ここで、n ではなく n をデコードしたものを用いるのは、セレクタ回路を単純化するためである。

3.2.4 ま と め

xlate は、命令間の依存関係を表すフラグを更新する。*xlate* は、新しい命令と命令ウィンドウ内の命令の依存関係を調べ、フラグに反映させる。更新される部分は、新しい命令に対応する列だけである。*rename* で命令ウィンドウに命令が入ってから *xlate* でフラグが更新されるまで、1 サイクルを要する。

3.3 *wakeup* ロジック

3.3.1 概 要

wakeup ロジックは、issue された命令の tagD と同じ tagL を持つ命令を検出し、rdyL 信号を出す。本方式の *wakeup* ロジックは、issue を word 線とする RAM に似た回路で構成する。図 5では、フラグとその先のゲート、縦横の issue,rdyL 線がその役割を負っている。

3.3.2 構 造

wakeup ロジックは RAM に似た構造で、フラグ、ゲート、issue 線、rdyL 線から成る。図の issue 線 が RAM の word 線に対応し、rdyL 線が bit 線に対応する。

右の issue 信号は，命令ウィンドウ内の命令が発行され，間もなく結果が得られることを表す．つまり，その命令のデスティネーションが間もなく ready になることを表す．issue 信号は，各フラグ行の出力ゲートを開き，ゲートを出た値は，縦方向に OR され下から出力されるようになっている．

3.4 タイミング

3.4.1 ready-bit

ここまで説明しなかったが、生産側の命令が既に終了し、命令ウィンドウ内に無いことも考えられる。このような場合、今まで述べてきた方法では消費側の命令を発行できない。

そこで、SS と同じように、オペランドに書き込みがあったら、オペランドがそれを記憶するようにする。具体的には、各物理レジスタにそれぞれ **ready-bit** を設け、そこに記憶するようにする。

ready-bit は，レジスタにオペランドが「ある」かどうかを表すフラグである．*rename* でレジスタが割り当てられるとクリアされ、オペランドが無いことを表す．その後，レジスタに書き込みがあると ready になり，オペランドがあることを表す．

消費側の命令が命令ウィンドウに入るときは，まずこの ready-bit を参照してオペランドがあるかどうかを判断する．オペランドがあれば，生産側の命令を待つ必要は無いので，直ちに rdyL(オペランドの ready フラグ) を ready にする．ready でない場合のみ，生産側の命令を待つ．このように，rdyL を ready にする道は ready-bit によるものと生産側を待つものの 2 通りがある．

3.4.2 パイプラインの動作

パイプラインの動作について説明する．先ほど説明したように，オペランドの rdyL が ready にされる道は 2 通りある．

(1) 命令ウィンドウに追加する時，既にレジスタの ready-bit が ready である場合 (図 6(a))

(2) 生産側の命令が発行され，それによりオペランドが ready になることが判った場合 (図 6(b))

まず，1 の場合のパイプライン動作について説明する．*rename* で命令を命令ウィンドウに追加する時，既に ready-bit が ready ならオペランドの rdyL がセットされる．この場合は *rename* 時に rdyL をセットするので，次のサイクルには既に *select* で選択可能である．従ってパイプラインは，図 6(a) のようになる．

ここで ready-bit の読み書きをするタイミングについて説明する．ready-bit を読み出すのは，*rename* の後半である．ready-bit に ready を書き込むのは，レジスタに値を書き込む時でなく *select* の直後である．これより遅くすると，それに依存する後続命令の開始が遅れてしまい，望ましくない．また，ready-bit への書き込みは *select* の結果に依存するので，これより早くはできない．

この 1 の方法をとるには，先行命令が ready-bit に書く以降のサイクルで ready-bit を読み出す必要がある．したがって，先行命令が ready-bit に書く以降のサイクルで *rename* をする必要がある (図 6(c,d))．これより早く *rename* をした命令は，次に説明する 2 の方法で rdyL を ready にする必要がある．

2 の場合について説明する．これは *xlate* の後で 先行命令の issue が来て，*wakeup*，*select* と進む場合である．この場合のパイプラインは，図 6(b) のようになる．*xlate* の直後に先行命令の issue 信号が来ない

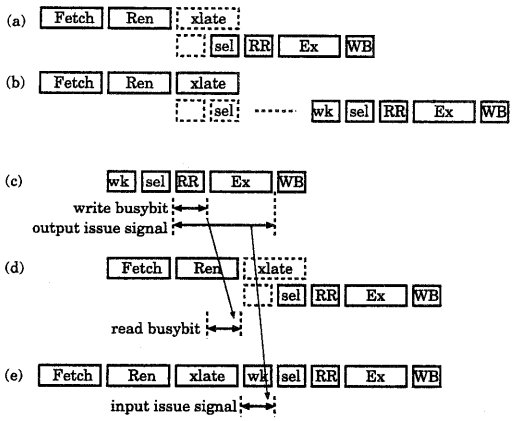

図 6 パイプライン構成

と，*wakeup·select* を行えず，バブルが生じる．

1 の方法がうまく行かない場合は，全てこの方法で行う必要がある．1 の方法というのは，生成側命令が ready-bit を ready にしたサイクル以降で消費側命令が *rename* を行う場合に有効であった (図 6(c,d))．従って，その 1 サイクル前に *rename* を行う場合 (図 6(c,e))，生成側命令は ready-bit を ready にしてからさらに 1 サイクル後まで issue を出して *wakeup* させる必要がある．

このように、ready-bit を読み出すのは、*rename* の後半で、書き込むのは *select* の直後である。issue 出力のタイミングは、*select* から、ready-bit に書き込んでから更に 1 サイクル後までである。

3.4.3 命令ウィンドウのエントリ解放のタイミング

普通の回路では，発行した命令は直ちに命令ウィンドウから削除されるのだが，本方式では異なる．命令を発行して issue 信号を出す時，そのエントリのフラグの行が書き換わっていると，*wakeup* が誤動作をしてしまう．この状況は，発行した命令のエントリに新しい命令が入り，その後続の命令が *xlate* を行うと生じる．

従って命令発行後も，最後の issue 信号を出す 1 サイクル前までは，そのエントリに新しい命令を入れられない (2 サイクル前に新しい命令が入ると，1 サイクル前に *xlate* が行われ，issue を出したときにはフラグが書き換わっているため)．

また，*select* の次の半サイクルで，選択された命令を命令ウィンドウから取り出してレジスタにアクセスするので，それが終わるまではエントリを書き換えられない．

3.4.4 分岐ミス時のペナルティ

分岐ミス時，分岐以前の命令の *wakeup·select* が終わってから分岐後の命令の rename が終わるまで少なくとも 3 サイクル (分岐命令の実行，分岐先の *fetch·rename*) はかかる．

従って，分岐以前の命令のデスティネーションの ready-bit は大抵 ready になっていると考えられる．この場合，分岐ミス後の命令は，*xlate* なしに *wakeup·select* が可能である．従って，分岐ミス時のペナルティーは *xlate* の無い普通の SS とほぼ同等であると考えられる．

3.4.5 ま と め

命令ウィンドウのエントリを解放するタイミングは、従来の方法と比べて幾分遅れることが分った。また、分岐ミス時のペナルティは従来の方法と比べてほとんど増加しないことが分った。

4. おわりに

動的命令スケジューリングの回路は将来，あるいは現在の SS プロセッサのクリティカルパスになる部分であり，サイクルタイムを短くする上で重要である．本論文では，実行時に SS の命令から動的に宛先を求めて *wakeup* の速さは Dualflow のそれと同程度である。

また，パイプライン構成についても考察した．その結果，分岐ミス時のペナルティーは従来の SS と比べてほとんど増加しないことがわかった．ただ，命令ウィンドウの開放は従来の回路に比べて遅れる事がわかった．

これらの得失から，本方式は従来の回路と比べ，より高速になると思われる．今後は，ゲートレベルの設計，レイアウト，Spice によるシミュレーションを行い，従来の回路と比較を行う予定である．

謝辞

なお，本研究の一部は文部省科学研究費補助金 (基盤研究 (B)(2) 課題番号 12480072 ならびに 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) グェンハイハー, 縣亮慶, 五島正裕, 中島康彦, 森眞一郎, 富田眞治: *(SWoPP2000)* (2000). (発表予定).
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, 情処研報 99–ARC–134, pp. 163–168 (1999).
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).