Dualflow アーキテクチャの提案

五島正裕[†]　グェンハイハー[†]　縣　亮慶[†]
森眞一郎[†]　富田眞治[†]

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．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．ただし実行命令数は増加する．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．今回のコンパイラには必要な最適化がほとんど実装されていないため，実行命令数の 50%も無用な命令が占めるという結果を得た．

Proposal of the Dualflow Architecture

Masahiro Goshima,[†] Nguyen Hai Ha,[†] Akiyoshi Agata,[†]
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. But the architecture requires extra instructions. We developed a compiler and evaluated it with the SPEC benchmark. Since the compiler implements almost no optimization techniques, about 50% of executed instructions are useless.

1. はじめに

Superscalar（以下，SS と略）の IPC を向上させる最も手っ取り早い方法は，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 を向上させるとこができた．

しかし現在では，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ってきている．*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 の基本構造の中では，動的命令スケジューリングを行うためにオペランドの有効性を追跡するロジックが特にクリティカルである．しかしその複雑さは SS にとって本質的なもので，根本的な対処法は存在しない．

このような背景から我々は，**Dualflow** と呼ぶ新しいアーキテクチャを提案する．Dualflow は，ISA のレベルからデータ駆動的な性質を導入することにより，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．

以下，2 章で SS の複雑さについて述べた後，3 章で Dualflow のアーキテクチャについて説明する．4 章でコード生成と最適化手法，そして，性能評価とその結果について述べる．

2. Superscalar の複雑さ

本章では，LSI の微細化により配線遅延の影響が増大することも考慮した上で，将来，SS のどこがクリティカル・パスとなるかを考える．

2.1 *IW, WS* と遅延

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，直接システムのクリティカル・パスの延長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，実効的な遅延—1 サイクルに終えなければならない処理の遅延を，劇的に短縮することができるからである．

例えば，命令フェッチやレジスタ・リネーミングな

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

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 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，処理の遅延を，一部の命令のレイテンシや，何らかのペナルティに転化するものである．これらの技術が有効であるのは，クロック速度の向上に対して，IPC の悪化の度合いが小さいからである．

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ない．以下では，その理由について詳しく述べる．

2.2 Superscalar の動的命令スケジューリング

まず 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，その遅延と，クリティカル・パスとの関係について考察する．

2.2.1 動的命令スケジューリングの原理

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．

さて，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*** 各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る．このバッファの ID をタグという．この時，バッファはオペランドが『ない』状態に初期化される．
   - I_u は，論理レジスタ番号から，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．

Out-of-order 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．またこのことは，in-order SS にもほぼそのまま当てはまる．In-order SS では，*rename* を行わず，論理レジスタ番号をそのままタグとして用いると考えればよい．

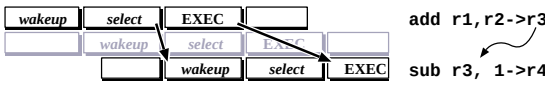

図 1 *wakeup* と *select* のパイプライン化

この性質のために，SS のウィンドウ・ロジックは，タグをキーとする CAM を用いて実装する必要が生じる．その理由などについては次章以降で詳しく述べることとし，次項ではまず，上述の 3 つの処理の遅延時間と，クリティカル・パスとの関係について考える．

2.2.2 遅延とクリティカル・パス

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*** は合わせて 1 サイクルで実行する必要があり，パイプライン化することができない．図 1 に，*wakeup* と *select* にそれぞれ 1 サイクルかけた場合のパイプラインの様子を示す．図の場合，`add` の結果を使用する `sub` は，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．

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[2)]．その結果を図 2 に示す．*wakeup* ロジックは普通，タグをキーとする **CAM** で構成される．3.3 節で詳しく述べるが，CAM の遅延は，Tag Drive，Tag Match，Match OR に分解することがで

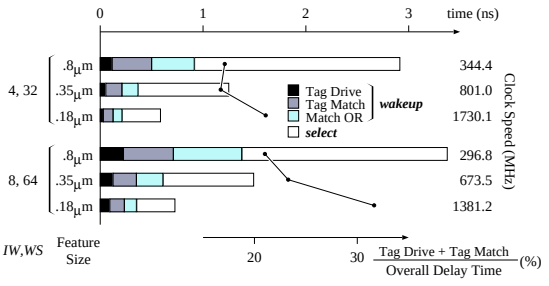

図 2 *wakeup* と *select* の遅延

きる．グラフ中，横棒は各々の遅延時間を表す．右端の数値は，*wakeup*+*select* の遅延時間がクリティカル・パスとなった場合の動作周波数である．上下のグループは，*IW, WS* が，それぞれ，4，32；8，64 の場合を表す．各グループ内の 3 つは，feature size が .8μm，.35μm，.18μm の場合である．

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．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．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る[3]．しかし，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．

2.3 本章のまとめ

本章では，*IW, WS* が増加し，微細なテクノロジを用いて作られる将来の SS において，そのクロック速度を規定する原因となる基本構造について考察した．その結果は以下のようにまとめられる：

- 多くの基本構造の遅延はパイプライン化などによってクリティカル・パスから外すことができるが，*wakeup* と *select* は 1 サイクルで実行する必要がある．
- 特に，*wakeup* ロジックは CAM であり，その連想処理部分が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っていくと予想される．

図 2 に示した遅延時間の絶対値は，実際よりかなり小さめに見積もられているようである．例えば R10000 の整数命令キューは，$IW = 2$，$WS = 16$，.35μm で，動作周波数は 200MHz である．この値は見積よりずっと小さいが，文献[4]では，既にこの部分がクリティカル・パスになっていると述べられている．

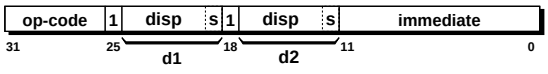

図 3　命令フォーマットの例

3. Dualflow

詳しい理由は 3.3 節で述べるが，SS の *wakeup* ロジックが CAM を用いて実装されるのは，端的に言えば，その制御駆動モデルによるものである．すなわち，データ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ため，タグの一致比較 ——— 連想処理が必要となるのである．

本稿で提案する **Dualflow** は，データ駆動的性質を導入することでこの連想処理を不要とするアーキテクチャである．

本章では，以下，3.1 節で実行モデルについて述べた後，3.2 節以降で実装について説明する．

3.1 Dualflow の実行モデル

以下では，まずモデルの全体像について述べた後，実行例を用いて説明を行う．

命令フォーマット（1）

まず，Dualflow の命令フォーマットについて簡単に触れておく．図 3 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を示す $\mathtt{d}_n$ フィールドがある．$\mathtt{d}_n$ フィールドについての詳細は後述する．

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 $\mathtt{d}_n$ フィールドが示す後続の命令に送りつける．そのため，命令フォーマット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．

```
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 4　|a − b| を計算するプログラム

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 3 に示した命令フォーマットにおいて，d_n フィールドは，正確には，宛先のデータ・スロットを示す．フィールド中，disp サブフィールドは自命令と宛先データ・スロット間の変位を，s サブフィールドは宛先データ・スロットの左右を，それぞれ表す．

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．その制限の妥当性については，4 章で述べる．

実　行　例

では，図 4 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，更に 0 − d を計算して最終的な結果とする．

図 5 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直ちに実行されて，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関わらず，プレース 5 に実行結果を送りつけることになる．一方プレース 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得ないことであるが，SS ではごく普通のことである．

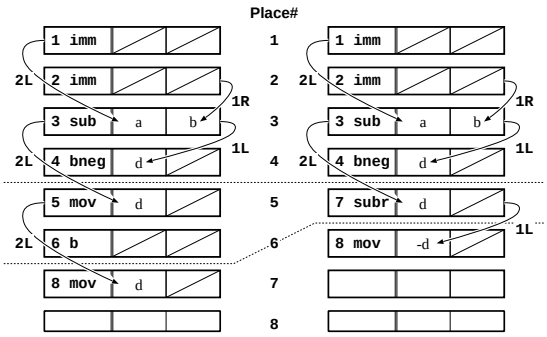

図 5　図 4 のプログラムを実行後のプレースの状態：左は条件分岐が Not Taken の，右は Taken の場合

(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 5 右）について説明する．この時点でプレース 5 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 は既に到着しているので，このプレースはフェッチ後 直ちに実行される．

(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，図 5 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 6 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．

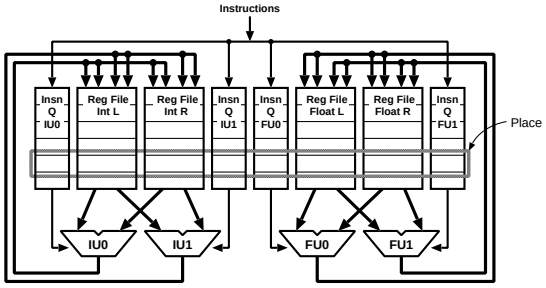

図 6　プレース・ウィンドウの実装

各レジスタ・ファイルは，Dualflow では更に，左右のオペランドごとに複製することが自然にできる．SS では，実行結果が左右のどちらのソースとして使用されるか分からないのに対して，Dualflow では，宛先にオペランドの左右が明示されるためである．

Dualflow では，1 つの命令が演算結果を最大 2 箇所のデータ・スロットに送るため，レジスタ・ファイルの書き込みポート数は $2 \cdot IW$ 必要になる．左/右オペランド用にそれぞれ別個のレジスタ・ファイルを用意すると，各メモリの書き込みポート数が $2 \cdot IW$ であることに変わりはないが，読み出しポート数はそれぞれ IW となる．SS では，書き込み IW ポート，読み出し $2 \cdot IW$ ポートであるから，ポートの総数は $3 \cdot IW$ で同じになる．

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．
   - Dualflow では，宛先データ・スロットの ID が SS のタグに相当する．宛先は，自命令と宛先スロット間の変位として disp フィールドで示される．したがってタグは，単に自命令が格納されたエントリの ID に disp を加算することによって求めることができる．加算の幅は $\log_2 WS$ b である．
   - Dualflow では，バッファは，I_d ではなく I_u に割り当てられる．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．I_u は，自命令のデータ・スロット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．
   - SS と同様に，I_d が実行されると，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．
   - SS では，I_u はタグで示されるバッファを見る．Dualflow では，SS とは異なり，I_u は自命令のデータ・スロットにオペランドが『ある』ことを見て，発行可能な状態になる．
3. ***select*** 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．

次節では，*wakeup* ロジックについて詳しく述べる．

3.3 *wakeup* ロジック

wakeup ロジックは，端的に言えば，各オペランドが利用可能になったかどうかを表すフラグの配列である．前述したように，SS ではこのテーブルを普通 CAM を用いて実装するが，説明の都合上まず RAM を用いる方式について考えよう．

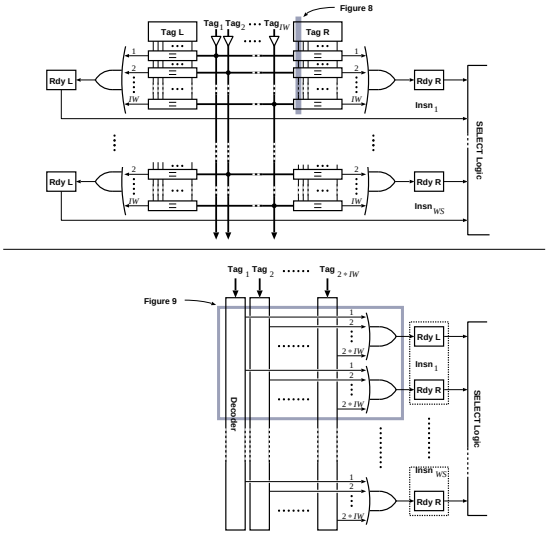

図 7　SS（上）と Dualflow（下）の *wakeup* ロジック

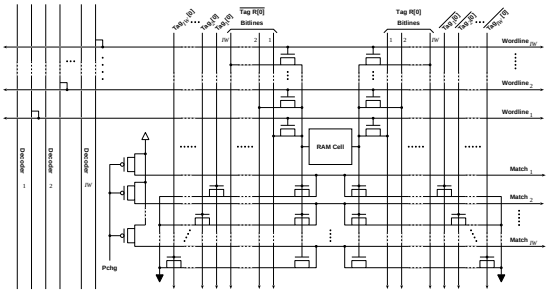

図 8　CAM セル

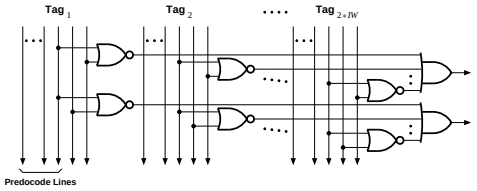

図 9　デコーダ

3.3.1　SS の *wakeup* ロジック

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し，I_d はタグを与えてフラグをセットすればよい．

SS では，I_d に加えて I_u もタグによってバッファを特定するので，I_u もタグをアドレスとして RAM にアクセスすることになる．そのための読み出しポートは，待っているオペランドの数 ——— $2 \cdot WS$ 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な解ではない．

そこで実際には，SS の *wakeup* ロジックは，タグをキーとする CAM として実装される．図 7 上に，一般的な *wakeup* ロジックのブロック図を示す．ただしこの図は，クリティカル・パスに関係のある部分だけを抜き出したものである．図中，各行は命令ウィンドウ中で待っている命令に対応し，待っているオペランドが利用可能かどうかを表すフラグ（RdyL/R）の他，そのオペランドのタグ（TagL/R）を格納するフィールドからなる．

2.2 節で述べたように，SS の *wakeup* ロジックの遅延は，以下の 3 つに分解できる：

Tag Drive　入力された IW 個のタグを，それぞれ $2 \cdot WS$ 個の一致比較器に対して放送する．

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．

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

図 8 に，CAM セルの構成例を示す．この図は，図 7 上の網掛けの部分に相当する．この回路の上半分は，Tag L/R を書き込むための IW 本の書き込みポートを，下半分は IW 個の 1b 比較器を構成する．

IW 個のタグは，縦に引かれたタグ・ラインによって，$2 \cdot WS$ 個の CAM セルに放送される．

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 として実装されている．マッチ・ラインは，high にプリチャージされ，一致しないビットに対応するゲートによって pull-down される．

タグ・ラインとマッチ・ラインの配線遅延が，それぞれ Tag Drive と Tag Match の遅延を支配している．これらの配線遅延のため，2.2 節で述べたように，LSI の微細化にともなって Tag Drive と Tag Match がクリティカルになっていくと予測される．

3.3.2　Dualflow の *wakeup* ロジック

Dualflow の *wakeup* ロジックも，SS と同様，1b×$2 \cdot WS$ word のメモリによって構成される．

Dualflow では，I_d は，SS と同様，タグによってバッファを特定する．一方 I_u は，SS とは異なり，バッファと 1 対 1 の関係にあるため，タグを必要としない．Dualflow では，RAM を用いて *wakeup* ロジックを構成しても，読み出しポートは必要とせず，RAM セルから直接配線を引き出すことができる．

図 7 下に，Dualflow の *wakeup* ロジックのブロック図を示す．図中上部から $2 \cdot IW$ 個のタグが入力されると，それぞれデコードされ，その出力が OR されて，フラグがセットされる．

ポート数と OR のファンインが，IW ではなく $2 \cdot IW$ となるのは，各命令が最大 2 個のデータ・スロットに結果を送るためである．

ロジックの OR から右の部分は，OR のファン・イン以外は，SS のそれとほぼ同じである．ただし，ファン・インが 2 倍になるので，この部分の遅延は SS に比べてわずかだが悪化する．

OR の左側は，デコーダの並びである．図 9 に，こ

の部分の詳細を示す．この図は，図 7 下の枠内に相当する．この部分は，$2 \cdot IW$ ポート，$2 \cdot WS$ word の RAM の行デコーダと機能的には等価であるが，以下の点でそれよりコンパクトである：

- 各ワードラインのファン・アウトは 1 であり，ドライバは必要ない．
- 実際に RAM セルがある訳ではないので，各デコーダのワードラインのピッチを狭くし，その分アドレス・ラインを短縮することができる．

宛先の数が 2 個という制約は，このように，*wakeup* ロジックの遅延と直接的なトレードオフの関係にある．IPC に大きな影響がない範囲でできる限り小さくすることが望ましい．

詳しくは定量的評価の結果を待つ必要があるが，SS と Dualflow の *wakeup* ロジックの遅延の差は，直感的には，CAM と RAM の遅延の差であるとできる．

3.4 本章のまとめ

前節までの議論から，Dualflow の SS に対する優位点をまとめると，以下のようになる：

1. **分岐予測ミス・ペナルティ** *rename* ロジックの遅延はたかだか $\log_2 WS$ b の加算器の遅延として与えられる．したがって，デコード・ステージを SS より短くできる可能性が高く，その分だけ分岐予測ミス・ペナルティを削減できる．
2. **クロック速度** SS では CAM で構成されるため，クロック速度を制限する可能性が高い *wakeup* ロジックを，RAM で構成することができる．

Dualflow は，制御駆動とデータ駆動の性質を合わせ持ち，SS と同様の動的命令スケジューリングを行うアーキテクチャである．本章の最後に，データ駆動型計算機と SS との違いについてまとめておく．

データ駆動型計算機との違い

前述したように，SS の動的命令スケジューリングは，局所的にデータ駆動型の計算を行うことであるとみなせる．データ駆動型計算機との違いは，データ駆動型の計算が行われる範囲が，プログラム・カウンタによって示される実行のスレッドの先端のごく狭い領域に限定されていることである．

Dualflow も，この点に関しては SS と同様の制御駆動的な性質を利用している．

この局所性によって SS と Dualflow は，データ駆動型計算機に比べ，小容量で高速な待ち合わせ機構を用いることができる．

Suprescalar との違い

Dualflow では，データ駆動的性質のため，連想処理を省略する代わりに，1 命令の結果を直接的に使用できる命令の数が制限される．これは，1 個の命令が定義する結果を複数の命令が使用するという，計算の自然な性質に反するものでもある．

一方 SS は，制御駆動的性質のため，1 命令の結果を使用できる命令の数には制限がない．そして，我々が嫌った連想処理によって，同時に $WS - 1$ 個もの命令にデータを受け渡すことができる．

しかし，実際のプログラムの多くでは，1 つの命令が定義する結果を使用する命令は，1 か 2 である場合が 90%程度以上を占めている[5]．また，3 個以上の命令が同時に発行可能になったとしても，演算器やレジスタなどの資源制約から，必ずしもその全てを引き続くサイクルに発行できるわけではない．

したがって我々は，この制約自体は決してアーキテクチャの優劣に直結するものではないと考えている．

4. 性能評価

Dualflow は，動的命令スケジューリングのハードウェアを簡略化する代償として，コード生成上の制約を受ける．コンパイラを作成し，性能評価を行った．本章ではその結果を示す．

4.1 コンパイラ

Dualflow の C コンパイラを作成するにあたっては，GCC をベースにした．GCC を通常のプロセッサにポーティングするには，プロセッサの構成に合わせて 2 つのファイルを記述するだけですむ．Dualflow に対しては，専用のパスを追加する必要があったが，上流のパスはそのまま流用することができた．

Dualflow では，宛先フィールドの値，すなわち，命令間の距離を静的に決定しなければならないという制約がある．制約は，以下の 3 つに分類できる：

数と距離 2 個を越えるスロット，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 データの授受が基本ブロックを越える場合には，命令間の距離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．

それぞれについて，以下のようにして制約を満たす：

数と距離 `mov` 命令を挿入し，データを中継する．各 `mov` 命令の 2 つの宛先によって，2 分木を組む．

条件分岐 taken/not taken 側それぞれの基本ブロック内部で適当に `mov` を挿入することによって受け取るデータ・スロットの配置を一致させる．

基本ブロック 関数呼び出しに対しては，距離を静的に決めることができないので，メモリを介してデータの授受を行う．SS で，すべてのレジスタを caller-save とした場合と同じと考えてよい．その他の場合には，間にある各基本ブロックに中継のための `mov` 命令を挿入する．図 4 で示したプログラムでは，5 行の `mov` がこれにあたる．

なおいずれの場合でも，距離が長くなりすぎるようであれば，メモリを介してデータを受け渡す．

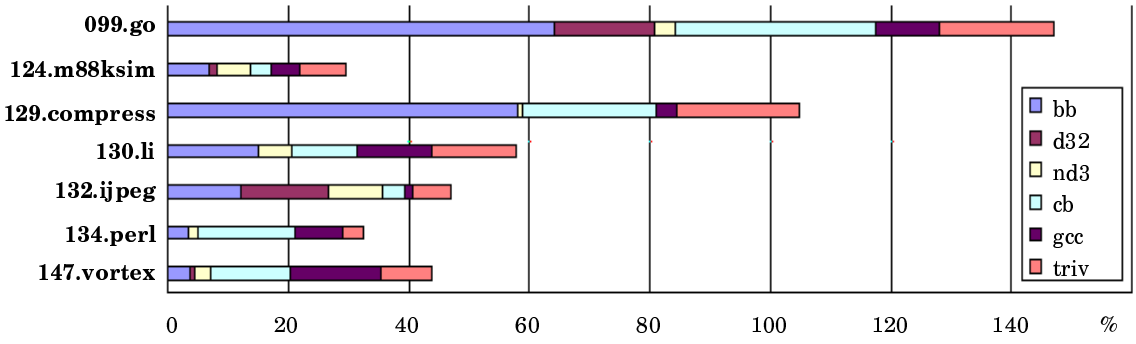

図 10　実行コードの増加率

4.2 測定結果

上述のコンパイラでSPECint95の126.gcc以外☆のプログラムをコンパイルして，インタープリタを用いて実行し，最初の1G命令に対して，有効な命令に対するmovとnopによる実行コードの増加率を計測した．

図10に結果を示す．図中の内訳は以下のとおり：

bb　基本ブロックの制約のために挿入されるmov．
d32　距離が32以上ある場合に挿入されるmov．
nd3　宛先が2個を越える場合に挿入されるmov．
gcc　GCCが勝手に挿入してしまうmov．
cb　条件分岐の制約のために挿入されるmovとnop．
triv　コンパイラの単純な不備によって挿入されているmov．今後簡単に削除することができる．

最適化がほとんど行われていないため，最大で150%ものmov，nopが実行されている．性能について論じられる段階にはないが，d32とnd3は，他に比べて最適化との関係が薄いため，一定の議論が可能である．d32とnd3は，*wakeup*ロジックの遅延と直接的なトレードオフの関係にあるため，重要である．

d32は，5つのプログラムでは非常に少ないが，090.goと132.ijpegでは多い．090.goでは，他の要因による無用な命令—特にbbが大量に挿入された結果として，命令間の距離が実際以上に伸びてしまっていることが原因と推定される．また現在のコンパイラは，一時変数の数が32に近づいた時に，安易にmovを挿入して距離を長くするという悪循環に陥っている．132.ijpegでd32が多いのは，主にこの理由による．bbの削減と命令の並び替えによる距離の最小化によって，d32は大幅に削減できる可能性がある．

nd3は，全てのプログラムで5〜10%程度である．nd3は，コンパイラの最適化によっては除去することができない．しかし，nd3によってクリティカル・パスが伸びてしまうのか，このデータからは判断できない．クリティカル・パス上にはなく命令フェッチのバンド幅が浪費されるだけであるのなら，*IW*が大きい場合には，この程度の増加は許容できる可能性が高い．

☆ 126.gccが使用する*alloca()*関数が実装できていないため．

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，分岐予測ミス・ペナルティとクロック速度の点でSSより有利である．しかし一方で，データ駆動的性質はコード生成上の制約になるので，無用な命令の増加による実効IPCの低下が懸念された．

そこで実際にCコンパイラを作成して，SPECベンチマークを用いて性能評価を行った．今回は最適化をほとんど行うことができなかったため，有効な命令よりも無用な命令の数が多いという結果となった．

今後は，まず第一に，コンパイラの最適化を充実させる必要がある．また，実際にトランジスタ・レベルの設計を行うなどして，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．

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., Jouppi, N. P. and Smith, J. E.: Quantifying the Complexity of Superscalar Processors, Technical report, Univ. of Wisconsin-Madison (1996).
3) Palacharla, S., Jouppi, N. P. and Smith, J. E.: Complexity-Effective Superscalar Processors, *ISCA24* (1997).
4) Yeager, K. C.: The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, No. 4, pp. 28–40 (1996).
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).