ALU Cascadingのための動的命令スケジューラ

尾形 幸亮[†] 姚 駿[††] 嶋田 創[††]
三輪 忍[†††] 富田 眞治[††]

低負荷時に低クロック周波数で動作するプロセッサにおいて，効率的な実行による消費電力の削減を狙い，ALU の出力を別の ALU の入力につなぎ，1 クロック・サイクル中にデータ依存関係にある命令の組を同時に実行する ALU Cascading を適用するアイデアがある．この手法をスーパスカラ・プロセッサに適用する場合，ALU Cascading 可能な組を同時に wakeup 可能な命令スケジューラが必要となる．本論文では，この ALU Cascading を行える命令スケジューラについて提案を行う．提案する命令スケジューラを用いた ALU Cascading を SPECint2000 を用いて評価した結果，ALU Cascading を行うと，IPC が平均で 3.8%向上するという結果になった．また，ALU Cascading 用の追加ハードウェアの評価を行い，追加ハードウェアによる実行ステージのクリティカル・パスの遅延時間の増加は 2.1%に過ぎず，また，消費電力の増加はプロセッサ全体の 1%未満であることを示した．

A Dynamic Instruction Scheduler for ALU Cascading

Kosuke Ogata,[†] Jun Yao,[††] Hajime Shimada,[††] Shinobu Miwa[†††] and Shinji Tomita[††]

To reduce power consumption via efficient execution under low workload and low clock frequency execution, there's an idea to adopt ALU cascading which executes several instructions under data dependency relationship in one clock cycle. Such execution is achieved by concatenating the output of the ALU into the input of the other ALU. To implement this technique to current superscalar processor, we have to prepare instruction scheduler which can wakeup pair of cascaded instructions simultaneously. In this paper, we propose the instruction scheduler which enables ALU cascading. The evaluation result shows that the IPC of SPECint2000 improves by 3.8% in average with ALU cascading under the proposed instruction scheduler. Additionally, we evaluated an additional hardware for ALU cascading. The result shows that the additional hardware for ALU cascading only increases 2.1% of the delay in the execution stage, and the power consumption increase is less that 1% in the whole processor power consumption.

1. はじめに

近年，製造プロセスの微細化にも関わらずプロセッサの消費電力が増大する傾向にあり，アーキテクチャ側からの消費電力を削減するための技術が重要性を増してきている．この中で，消費電力はクロック周波数に比例することに着目し，プロセッサの負荷に応じてクロック周波数を低下させる技術は，数多く研究開発されている．

通常のプロセッサは，**図 1**(a) のように，クロック・サイクル時間のほとんどを使って組み合わせ論理を動作させるように設計する．そのため，通常のプロセッサにおいてクロック周波数を低下させた場合，図 1(b) のように，クロック・サイクル時間の後半に組み合わせ論理の動作しない時間ができる．近年のプロセッサで多用されている Dynamic Voltage Frequency Scaling(DVFS) と呼ばれる技術では，電源電圧を低下させて組み合わせ論理の動作時間を延ばすためにこの時間を利用している (図 1(c))．消費電力は電源電圧の 2 乗に比例するため，DVFS ではクロック周波数の低下のみならず，電源電圧の低下によっても消費電力を削減できる．

上記のクロック周波数低下時のクロック・サイクル

† 三菱電機株式会社情報技術総合研究所
Information Technology R&D Center, Mitsubishi Electric Corporation
†† 京都大学大学院情報学研究科
Graduate school of Informatics, Kyoto University
††† 東京農工大学工学府
School of Engineering, Tokyo University of Agriculture and Technology

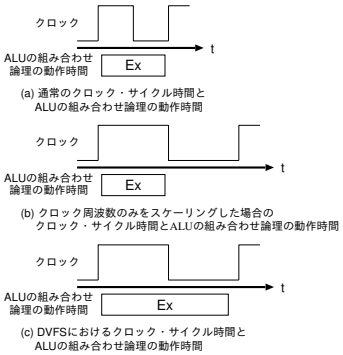

図 1 DVFS などの適用で延長されるクロック・サイクル時間

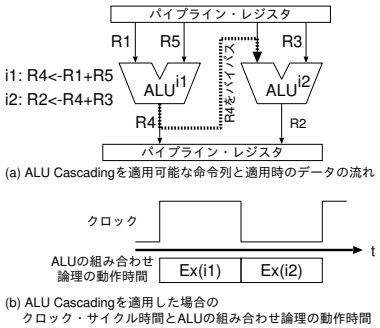

図 2 ALU Cascading

時間後半の空き時間を利用する別のアイデアとして，実行ステージに対して ALU Cascading と呼ばれる手法を適用する提案がある[1)~4)]．ALU Cascading は，ある ALU の出力を別の ALU の入力につなぎ，データ依存関係にある ALU 演算命令を 1 クロック・サイクルで実行するものである．以下，例を挙げて説明する．まず，前提として，プロセッサに対する負荷が低いためにプロセッサのクロック周波数が最大値の半分まで落とされており，クロック・サイクル時間が 2 倍になっていると仮定する．この時，**図 2**(a) の命令 i1,i2 のようなデータ依存関係にある 2 つの ALU 演算があり，それらを ALU Cascading によって実行するとする．この場合，まず，両命令を同時に別々の ALU に発行する．データ依存関係において先行命令である命令 i1 の演算は，左の ALU でクロック・サイクルの前半に終了する．ここで，命令 i1 の結果は，パイプライン・レジスタへ送られると共に，ALU 間に設けられたバイパス回路を通って，命令 i2 が発行された右 ALU の入力へ送る．そして，右 ALU では後続命令である命令 i2 の演算がクロック・サイクルの後半を使って実行される．このようにして，図 2(b) のように，実行ステージにおけるクロック・サイクル時間の後半を演算に利用することができる．この結果，ALU Cascading 適用中の IPC が向上する．必要とされるプロセッサ性能が ALU Cascading 適用前と同じ場合，この IPC 向上を利用して，適用前よりクロック周波数を更に下げることができるため，ALU Cascading を利用した消費電力削減が可能となる．

この ALU Cascading の適用例として，過去にはベクトル・プロセッサ[5)] やメディア・プロセッサ[6)] への適用例があったが，一般的なスーパスカラ・プロセッサへの適用例は少ない．これは，スーパスカラ・プロセッサの命令スケジューラはデータ依存関係にある命令を連続したサイクルに送り出すことに特化して設計されているため，コストの増加なしに，ALU Cascading に必要な，データ依存関係にある命令を同一サイクルに送り出すという動作を追加するのが難しいためと考えられる．過去の研究では，限定された命令の組に対して ALU Cascading を行う命令スケジューラは提案されているが[4)]，任意の命令の組に対して ALU Cascading を行う命令スケジューラは提案されていない．もちろん，一般的な CAM を用いる命令スケジューラにおいても，1 クロック・サイクル中に wakeup-select を複数回行ったり，3.3 節で説明するように本提案を CAM を用いて実現したりすることで任意の命令の組に対する ALU Cascading を行うことができるが，消費電力や回路面積が大幅に増大してしまう．これは，文献 1)，2) のように消費電力削減の一貫として ALU Cascading を適用する場合，同時に適用されている他の消費電力削減手法の効果を減じてしまう点で好ましくない．

本論文では，DMT(Dependency Matrix Table)[7)] 方式の命令スケジューラをベースとし，回路面積や消費電力の大幅な増大なしに，任意の命令の組に対する ALU Cascading を行う命令スケジューラを提案する．本スケジューラの実現のため，ベースとなる DMT 方式の依存行列表，レジスタ・マップ表，およびデータ・パスに拡張を加えるが，いずれも，回路面積や消費電力の増大を抑える形で拡張を行っている．そのため，本論文では ALU Cascading による性能向上と同時に，ALU Cascading のために追加された回路が消費する電力を調査し，追加回路の消費電力の増加が，ALU Cascading で削減できる消費電力を上回らないことも示す．

本論文の構成は以下の通りである．まず 2 節では，ベースとなる研究として DMT 方式の構成及び動作について述べる．次に 3 節で，DMT 方式を拡張した ALU Cascading を行うための仕組みについて述べる．4 節ではプロセッサ・シミュレータによる IPC の評価

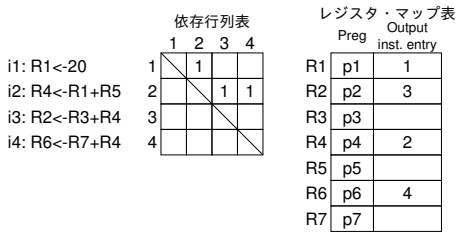

図 3 DMT 方式の依存行列表とレジスタ・マップ表

結果と，HDL を用いて合成した回路の消費電力を評価した結果を示す．そして 5 節で関連文献について述べ，最後に 6 節でまとめを述べる．

2. Dependence Matrices Table(DMT) 方式の命令スケジューラ

2.1 DMT 方式の概要

DMT 方式は，命令スケジューリング論理の構成手法の 1 つで，命令発行ステージにおいて命令の wakeup-select 動作を高速化するために考案された手法である．DMT 方式は，一般的な命令スケジューリング論理における CAM のタグ部に代わって配置される．その論理的な構造は**図 3** の依存行列表のようになっており，行列の縦横のサイズは命令ウィンドウのエントリ数となっている．行列の各要素は 1 ビットの SRAM である．行列の縦横に振られた番号は命令ウィンドウ内における命令番号に対応している．また，行列の行方向は依存元となる命令番号，列方向は依存先となる命令番号を表しており，命令 m の演算結果が命令 n のソース・オペランドとなっているとき，m 行 n 列にフラグが立つ．例えば，図 3 では，1 行 2 列目にフラグが立っているが，これは命令 i1 のディスティネーション・オペランドが命令 i2 の左ソース・オペランドとなっているためである．2 行 3 列目および 2 行 4 列目にある 2 つのフラグも同様に i2 と i3，および，i2 と i4 の依存関係を示している．上記の依存行列表の更新作業は，一般的な命令スケジューリング論理と同様に wakeup と select によって行われる．wakeup では，前サイクルで select された命令の命令番号の行のフラグを全てクリアし，それによって生成されたフラグのない列について，対応する命令の ready ビットを立てる．select は従来の命令スケジューラと同様，命令ウィンドウにおいて ready ビットが立っている命令の中から発行すべき命令を選択する．また，DMT 方式を用いた命令スケジューラでは，後続命令が先行命令の番号を検索するために，図 3 右のようにレジスタ・マップ表に Output inst. entry というフィールドが追加される．Output inst. entry は，その論理レジスタ番号に結果を出力する最新の命令の，命令ウィンドウにおける命令番号を表す．Output inst. entry は命令デコード時に書き込まれ，2.2 節で示すように依存行列表の更新に使用される．

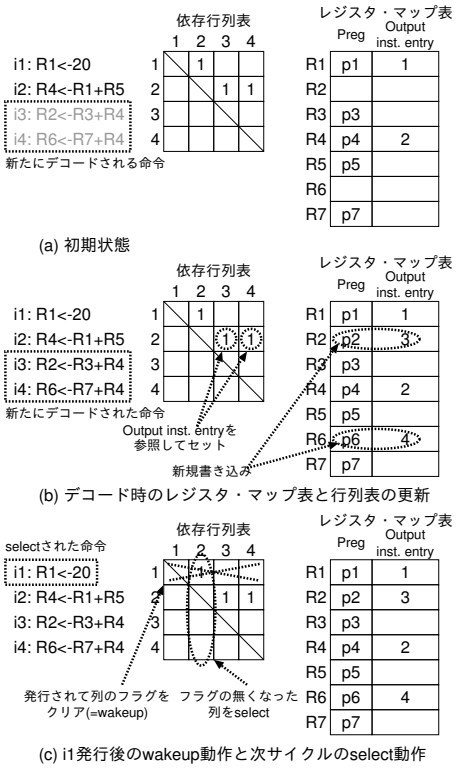

図 4 DMT 方式における依存行列表とレジスタ・マップ表の動作

2.2 DMT 方式の命令スケジューリング時の動作

図 4 に示す例を用いて DMT 方式の動作を説明する．図 2(a) の初期状態では，命令 i1,i2 がすでに命令番号 1,2 として依存行列表に登録されている．i2 は i1 に依存しているため，依存行列表の 1 行 2 列目にフラグが立っている．以下，i3,i4 を新たにデコードし，命令番号 3,4 として依存行列表に登録し，命令スケジューリングを行う様子を示す．なお，デコード幅と発行幅は 2 命令と仮定する．

(1) 命令デコードとレジスタ・マップ表読み出し

i3,i4 のソース論理レジスタ番号で図 4(a) の初期状態のレジスタ・マップ表を検索し，Preg エントリより物理レジスタ番号を得る．同時に，まだ値が生成されていない論理レジスタについては，その物理レジスタに値を出力する命令の命令番号を Output inst. entry より得る．図の例では，i3,i4 のソースの R4 のみ命令番号を得る．また，同時にデコードされた命令の間のデータ依存関係をチェックし，必要があれば，後続の

命令に先行する命令の命令番号を渡す．i3,i4 間には依存関係はないので，この部分による命令番号の追加はない．

(2) レジスタ・マップ表の更新

i3,i4 のデコードにより，i3,i4 に命令ウィンドウの命令番号 3,4，および，物理レジスタ p2,p6 が割り当てられたとする．この時の動作は，図 4(b) に示すように，レジスタ・マップ表の R2,R6 の Preg エントリに割り当てられた物理レジスタを，Output inst. entry にその物理レジスタに結果を出力する i3,i4 の命令番号を書き込むことになる．

(3) 依存行列表の更新

(1) のレジスタ・マップ表読み出しにより，命令番号 3 のエントリに書き込まれる i3 は命令番号 2 に依存し，命令番号 4 のエントリに書き込まれる i4 は命令番号 2 に依存していることが分かった．そのため，依存行列表の 2 行 3 列と 2 行 4 列にフラグを立てる．図 4(b) に更新後の依存行列表を示す．

(4) i1 の発行後の wakeup-select

i1 が発行されると，図 4(c) のように依存行列表の 1 行目のエントリの全てのフラグがクリアされる．これにより，2 列目のフラグがなくなるため，i2 が ready となる．select において，ready な命令は i2 のみのため，i2 のみが select される．なお，発行された i1 のレジスタ・マップ表の Output inst. entry は，発行と同時にクリアされる．

なお，次のサイクルの wakeup-select では，2 行目のフラグがすべてクリアされ，3 列目と 4 列目のフラグがなくなり，i3,i4 が ready となる．

3. DMT 方式をベースとした ALU Cascading のための動的命令スケジューラ

図 4(b) で例に示している命令列では，i2→i3,i2→i4 の ALU Cascading が可能である．この例の場合，i2 が ready となると同時に i3,i4 も ready になるようにフラグを立てれば，i2 と i3，i4 を同時に select し，ALU Cascading が行える．このように，2 つ前のデータ依存先の命令で，後続の命令の wakeup を行うことが，DMT 方式を用いた ALU Cascading の命令スケジュールの概要である．DMT 方式を用いて ALU Cascading を行う意義は，ALU Cascading の対象となる命令の組みの wakeup を少ない追加ハードウェアで行える点である．

3.3 節で説明するように，上記の考え方をもとに，従来の CAM を用いた命令スケジューラでも ALU Cascading のための命令スケジューリングは可能だが，消費電力や回路面積が大幅に増大してしまう．そのため，ALU Cascadig に必要なハードウェア・コストを抑え，上記の問題を克服することができる本稿で提案する方式の方が優れていると考える．以下，DMT 方式を用いた ALU Cascading を行うためのハードウェアの拡張とその動作を説明する．

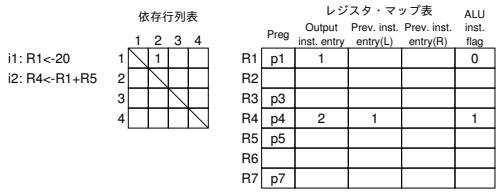

図 5 拡張されたレジスタ・マップ表

3.1 レジスタ・マップ表の拡張

ALU Cascading の実装は，2 節で述べた DMT 方式の命令スケジューリングを拡張する形で行う．まず，1 つ前の依存先の命令の命令番号を得ることができる DMT 方式のレジスタ・マップ表を，ALU Cascading のスケジューリングに必要な，2 つ前の依存先の命令の命令番号を得ることができるように拡張する．この拡張により，**図 5** のように Prev. inst. entry(L)，Prev. inst. entry(R) が追加される．Prev. inst. entry は，Output inst. entry の示す命令が持つソース・オペランドが，どの命令の出力に由来するかを記憶する．たとえば図 5 では，論理レジスタ R4 に結果を出力する命令は i2 であるため，レジスタ・マップ表において R4 に対応する Output inst. entry には命令番号 2 と記憶されている．ここで，更に i2 の左ソース・オペランドが i1 の出力に依存しているので，レジスタ・マップ表の同じ行において Prev inst. entry(L) に命令番号 1 と記憶されている．なお，Prev inst. entry は命令が持てるソース・オペランドの数だけ必要である．

図 5 ではさらに，その論理レジスタに結果を書き込む命令が ALU Cascading の対象となる命令かを判別するための，ALU inst. flag も追加されている．ALU Cascading 用のスケジューリングを行う場合，ALU inst. flag がセットされている場合は Prev inst. entry を参照して 2 つ前のデータ依存先の命令で wakeup を行い，それ以外の命令は従来と同様に Output inst. entry を参照して 1 つ前のデータ依存先の命令で wakeup を行うようにする．

3.2 DMT 方式を用いた ALU Cascading 用の命令スケジューリング

ALU Cascading を行う場合の依存行列表への登録やレジスタ・マップ表の更新を，図 5 および**図 6** の例を使って説明する．**図 2.2** 節と同様に，図 5 の初期状

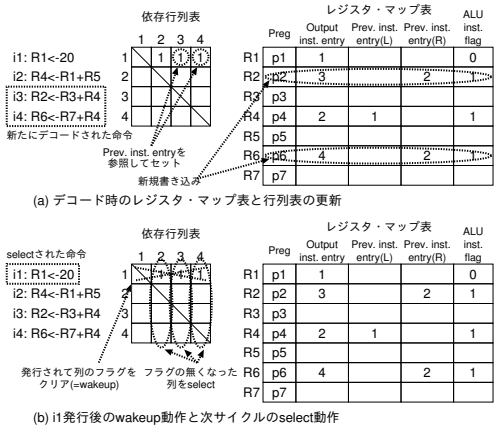

図 6 ALU Cascading 用スケジューリング時の依存行列表とレジスタ・マップ表の動作

態では命令 i1,i2 がすでに命令番号 1,2 として依存行列表に登録されており，依存行列表の 1 行 2 列目にフラグが立っている．また，レジスタ・マップ表において i2 に対応する Prev inst entry(L) には命令番号 1 が記憶されている．i1 は即値ロード命令なので ALU inst. flag はセットされておらず，i2 は ALU 演算命令なので ALU inst. flag はセットされている．

(1) 命令デコードとレジスタ・マップ表読み出し

図 6(a) のように，新たに命令 i3，i4 がデコードされたとする．まず，2.2 節 (1) と同様に，i3,i4 のソース論理レジスタ番号でレジスタ・マップ表を検索し，物理レジスタ番号を得る．まだ値が生成されていない論理レジスタについては，その物理レジスタに値を出力する命令の命令番号を得る．図 6(a) の例では，i3,i4 のソースの R4 で検索し，ヒットしたエントリの Output inst. entry から i2 の命令番号 2 を得る．さらに同じエントリの Prev inst. entry(L) から i1 の命令番号 1 を得る．さらに，ALU inst. flag より，この論理レジスタに結果を出力する命令は ALU 演算命令であることを知る．

(2) レジスタ・マップ表更新

2.2 節 (2) と同様に，i3,i4 に命令ウィンドウの命令番号 3,4，および，物理レジスタ p2,p6 が割り当てられたとする．レジスタ・マップ表の R2,R6 のエントリに割り当てられた物理レジスタと，その物理レジスタに結果を出力する i3,i4 の命令番号を書き込む．さらに，i3，i4 ともに右ソース・オペランドが i2 に依存しているため，それらのエントリの Prev inst. entry(R) に本節 (1) で得られた i2 の命令番号 2 を書き込む．

(3) 依存行列表の更新

(1) のレジスタ・マップ表読み出しで，i3，i4 が依存している i2 は，i1 に依存していることが分かった．また，ALU inst. flag より，i2 は ALU 演算命令であり，ALU cascading に組み込めることも分かった．そこで，通常の DMT 方式では Output inst. entry から得られた命令番号 2 を用いて 2 行 3 列と 2 行 4 列にフラグを立てるところを，ここでは Prev inst. entry から得られた命令番号 1 を使って 1 行 3 列と 1 行 4 列にフラグを立てる．

このような手順で，2 つ前のデータ依存先の命令に対する依存を依存行列表に記述する．なお，図の例では 2 つ前のデータ依存先の命令は 1 つしかないが，最大で 4 個の命令に依存することが考えられる．このように，複数のデータ依存先がある場合でも，依存行列表は 1 列中の複数のエントリにフラグを立てることによって複数の依存を記述できるため，問題はない．

(4)i1 の発行後の wakeup-select

i1 が発行されると，図 6(b) のように依存行列の 1 行目のエントリのフラグが全てクリアされる．2,3,4 列目のフラグは全てなくなるので，i2,i3,i4 は同時に ready となる．このように，(3) で依存行列表の更新に Prev inst. entry の情報を用いたことにより，依存関係にある演算命令を一度に ready にすることができる．なお，ALU に空きがなく，全ての演算命令を一度に発行できない場合は，命令ウィンドウ内において古い命令から順に発行できる分だけ発行される．このように，古い命令から発行するため，i3 や i4 のように後続命令だけが発行されることはない．また，図 5 に示した例では，i2→i3 と i2→i4 のように，ALU Cascading を適用できる組が 2 つあるが，提案構成では両方の組に対して同時に ALU Cascading を適用できる．以下，これを，1 対多の ALU Cascading と呼ぶ．文献 4) のグルーピング方式では，このような 1 対多の ALU Cascading はできず，提案構成がグルーピング方式よりも高い性能を得られる理由の 1 つとなる．

3.3 CAM を用いた命令ウィンドウによる ALU Cascading

2 つ前のデータ依存先の命令で wakeup を行うという考え方をもとに，以下のようにすれば，CAM を用いた命令ウィンドウでも i2 と i3，もしくは，i2 と i4 を同時に ready にすることができる．

(1) レジスタ・マップ表の Prev inst. entry の代わりに，その物理レジスタに値を書き込む命令のソース・タグを書き込んでおく．

(2) 通常，命令ウィンドウに命令を登録する時，レジスタ・マップ表から読み出されたソース物理

レジスタ番号をソース・タグとして CAM に登録するが，ALU Cascading を行う時には (1) でレジスタ・マップ表に新たに追加した，ソースを生成する命令のソース・タグを書き込む．

従来の CAM を用いた命令ウィンドウでは，1 エントリあたり CAM は 2 つ必要になるが，上記の方法で ALU Cascading を行う場合は，更に 4 つ必要となる．この構成でもクロック・サイクル時間に悪影響を与えずに ALU Cascading は行えるが，命令ウィンドウ用の CAM の追加は回路面積や消費電力の点から望ましくない．

依存行列表では，従来の命令ウィンドウの 2 つの CAM を 1 つの依存行列表にまとめられるのと同様に，上記の 4 つの CAM を 1 つの依存行列表にまとめることができるため．この問題によるコストの増加がなくなる．

3.4 命令発行後の Cascading 実行

ALU Cascading の対象となる先行／後続命令が select されると，それらの命令は別々の ALU を割り当てられ，レジスタ・フェッチが始まる．Cascading 実行のための同時に発行された命令間の依存関係の解析，および，結果フォワーディングのための次サイクルに実行が完了する命令との間の依存関係の解析は，レジスタ・フェッチと平行して，各命令のソースやディスティネーションの物理レジスタ番号を比較することで行われる．レジスタ・フェッチの済んだ命令は実行ステージに送られて実行される．このとき，実行部ではレジスタ・フェッチの際に解析された命令間の依存関係に基づき，ALU 間のバイパス路を開いて Cascading 実行を行ったり，結果フォワーディングを用いた実行を行う．

図 7 に ALU Cascading のためのバイパス回路を追加したレジスタフェッチ・ステージ (RF) と実行ステージ (EX) を示す．op は命令のオペコード，dtag は命令の物理ディスティネーション・レジスタ番号，stagL/R は左右の物理ソース・レジスタ番号，valueL/R は左右のソース・レジスタ値，dvalue は演算結果である．以下，ALU Cascading のためのバイパス回路を Cascading バイパス回路と呼称する．図 7 において，網掛けされた部分が Cascading バイパス回路である．wakeup/select された命令に対しては，レジスタ・フェッチと並行して，結果フォワーディングのための調停と ALU Cascading のための調停が行われる．select された命令の stagL/R は，他の同時に select された命令の dtag と比較される．一致すれば，その dtag を持つ命令の結果を利用することが分かるので，ソース・オペランドの値としてレジスタ・ファイルの値ではなく，図 7 下段の実行ステージにおいて，ALU の出力からパイプライン・レジスタを介さずにバイパスされる値を利用する．バイパスされる値のうち，どれを利用するかは csel に記憶され，ALU 入り口のカスケーディング用マルチプレクサ (Cas. mux) へ送られる．Cas. mux では，バイパスされる値とレジスタ・ファイルからの値のうち，どれを利用するかが選択される．なお，Cas. mux の隣に配置されたフォワード用マルチプレクサ (Fwd. mux) は，結果フォワーディングされてくる値を選択するためのマルチプレクサである．Cas. mux や Fwd. mux の下段には，もう 1 つのマルチプレクサがあり，Cascading バイパス回路からの入力，結果フォワーディング回路からの入力，レジスタ・ファイルの値のうちのいずれかを選択する．このマルチプレクサは，通常の結果フォワーディング回路のみの実行部には存在せず，このマルチプレクサの追加により，結果フォワーディングを用いた演算の遅延時間が伸びる可能性がある．この遅延は，クロック周波数を低下させていない状態において問題となる．この遅延がサイクル時間に与える影響については，4.2 節で評価結果を示す．なお，図 7 の EX ステージ中の 2 本の太い破線は，4.2 節での遅延時間の評価に用いるパスを示している．

提案するスケジューラを用いて ALU Cascading を行う場合，ALU Cascading を行う ALU は専用 ALU ではなく，プロセッサが通常の備えている ALU を利用する．これは，提案するスケジューラでは，命令を select した段階ではどの命令の組が ALU Cascading 可能か分からないため，発行後に ALU 間の接続を決定可能な構成にする必要があるためである．似たような理由により，通常のバイパス回路を Cascading バイパス回路に利用することはできない．これは，通常のバイパス回路を利用して ALU Cascading を行うためには，クロック・サイクル時間の半ばで，先行する演算が行われる ALU の出力側のパイプライン・レジスタを無効化もしくは更新しなくてはならないためである．もし，当該パイプライン・レジスタに前クロック・サイクルの演算の結果値が入っている場合，クロック・サイクル時間の半ばでその値が消えてしまうことになる．このような状況に陥るのを防ぐため，Cascading バイパス回路は専用の回路を設けることになる．

4. 評 価 結 果

本節では，ALU Cascading による IPC 向上と，ALU Cascading のための回路を追加した実行部の遅

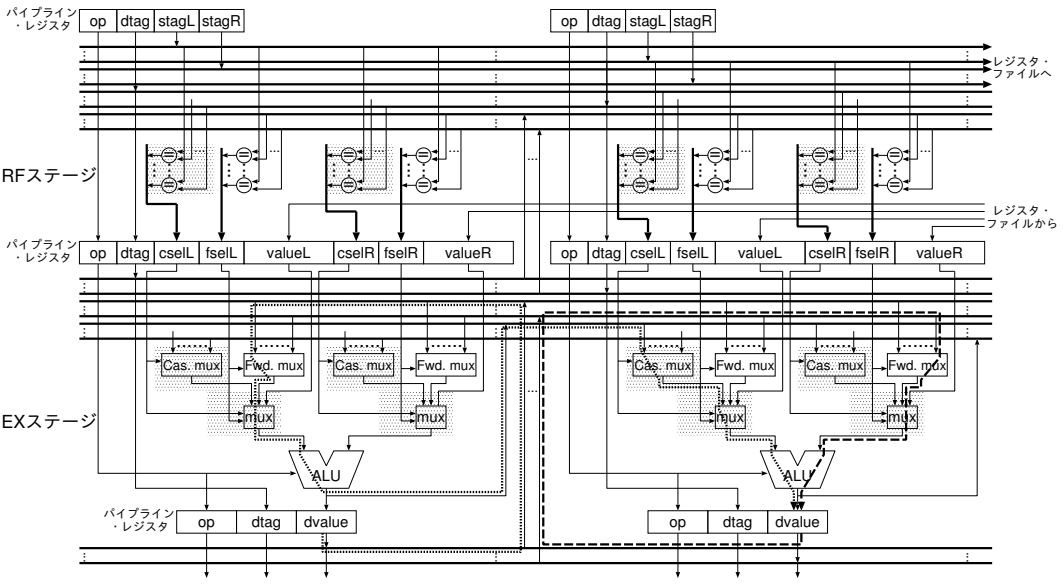

図 7 Cascading 用バイパス回路を追加した RF ステージと EX ステージ

表 1 プロセッサの仕様

命令発行幅	8
命令ウィンドウ	128 エントリ
LSQ	64 エントリ
int ALU	8
fp ALU	4
int mul/div	8
fp mul/div	4
分岐予測機構	8K-entry gshare/6-bit history/ 512-entry BTB/16-entry RAS
分岐予測ミス・ペナルティ	7 サイクル
メモリ・ポート数	8
1 次命令キャッシュ	64KB/32B-line/2-way
1 次データ・キャッシュ	64KB/32B-line/2-way
2 次キャッシュ	2MB/64B-line/4-way

延時間，および ALU Cascading のための回路の消費電力を評価した結果を示す．

4.1 IPC 向上の評価

ALU Cascading による IPC 向上の評価には，SimpleScalar Tool Set[8] を用い，その中に含まれる out-of-order 実行シミュレータを用いた．シミュレーションで仮定するプロセッサの仕様を**表 1** に示す．プロセッサのパイプラインは 10 段であると仮定し，ALU Cascading を適用した時の IPC を測定した．今回のシミュレーションでは，ALU Cascading の対象を整数加減算，シフト，論理演算といった，実行にかかる時間の短い単純な ALU 演算命令に限定した．また，先行する ALU 演算命令に依存する ALU 演算命令が複数ある場合にも複数の後続命令との ALU Cascading を行える，1 対多の ALU Cascading をサポートしている．さらに，比較のため，ALU Cascading を無効にした時の IPC や，同時にデコードされた ALU 演算命令に対して 1 対 1 の ALU Cascading を行う命令グルーピング方式[4] の IPC も測定し，提案する命令スケジューラによる ALU Cascading の効果を評価した．ベンチマーク・プログラムは，SPECint2000 から 8 本を用いた．

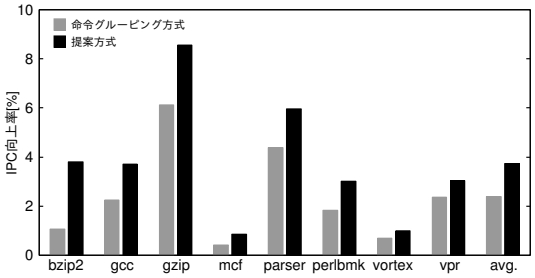

図 8 ALU Cascading による IPC 向上率

図 8 に ALU Cascading による IPC 向上率を示す．グラフ横軸はベンチマーク・プログラム，縦軸は IPC 向上率を表す．各ベンチマークに対する棒グラフは左から，命令グルーピング方式で ALU Cascading を適用した場合，提案方式で ALU Cascading を適用した場合となっている．いずれのベンチマークでも，ALU Cascading を適用することにより IPC の向上が見られた．平均 IPC 向上率は命令グルーピング方式で 2.3%，提案方式で 3.8%となった．この命令グルーピング方式との IPC 向上率の差は，ALU Cascading の対象が同時にデコードされた命令に制限されない点や，1 対

多の ALU Cascading が行える点によって発生するものと考えられる．

必要とされるプロセッサ性能が ALU Cascading 適用前と同じ場合，この IPC 向上を利用して，適用前よりクロック周波数を更に下げることができるため，ALU Cascading を利用した消費電力削減が可能となる．そのため，この IPC の向上は 3.7%の消費電力の削減と取ることも出来る．

4.2 実行ステージの遅延時間の評価

3.4 節で説明したように，ALU Cascading では，異なる ALU 間で Cascading バイパス回路を開くことによって先行命令から後続命令へデータをバイパスする．Cascading バイパス回路の追加によって，結果フォワーディングのためのパスにマルチプレクサが 1 段増えることになり，結果フォワーディングの遅延が増加する．プロセッサのクロック･サイクル時間は整数演算の遅延時間を基準にする場合が多いために実行ステージはクリティカルなステージとなり，さらに，結果フォワーディングは一般的に実行ステージでクリティカル･パスとなるため，クロック周波数を低下させていない状態において，結果フォワーディングの遅延が顕著に増加することは好ましくない．また，ALU Cascading による演算を行う場合，データが Cascading バイパス回路を通る際に，後続の ALU 入口でマルチプレクサを 2 段通過し，これが Cascading バイパス回路の遅延となる．我々の方式では，クロック周波数を最大値の半分まで低下した時点で ALU Cascading を適用することを想定しているが，もし前述の 2 段のマルチプレクサによる遅延が顕著である場合，クロック周波数を最大値の半分まで低下させても ALU Cascading を適用できなくなる恐れがある．従って，このマルチプレクサによる遅延が顕著であることは好ましくない．

本節では，Cascading バイパス回路を有する実行ステージと有しない実行ステージを HDL で記述して論理合成し，両実行ステージの結果フォワーディングの遅延時間の差を評価する．また，Cascading バイパス回路を有する実行ステージにおいて，ALU Cascading による演算の遅延を測定し，前述の Cascading バイパス回路の遅延を含めた 1 演算あたりの遅延を評価する．

回路を評価するため，整数演算部とパイプライン・レジスタを設計した．回路の記述には Verilog HDL を使用し，整数 ALU を 8 個有する実行ステージを仮定した．ALU は not，and，or，xor，add，sub が実行可能な 32-bit ALU とした．設計した回路の論理合成には Synopsys 社の Design Compiler を使用し，論理合成の際のスタンダード･セル･ライブラリとして Oklahoma State University が提供する TSMC 0.18μm プロセスのライブラリ[9] を使用した．結果フォワーディングのパスは，ALU 出力側のパイプライン・レジスタから結果フォワーディングのための配線を通り，ALU で減算が行われた後，結果を ALU 出力側のパイプライン・レジスタへ書き込むパス (図 7 の荒い破線のパス) とした．また，ALU Cascading による演算の遅延は，ALU 出力側のパイプライン・レジスタから結果フォワーディングのための配線を通り，ALU で減算が行われた後，演算結果が Cascading バイパス回路を通って別の ALU へバイパスされ，そこでも減算が行われて結果を ALU 出力側のパイプライン・レジスタへ書き込むパス (図 7 の細かい破線のパス) とした．ALU における演算に減算を選択した理由は，減数の 2 の補数を求める処理が必要となる減算が ALU 内で最も時間のかかる演算であり，実行ステージにおいて最もクリティカルな処理だからである．

表 2 ALU Cascading なしの時の実行ステージの遅延時間

Cascading バイパス回路	なし	あり
結果フォワーディングの配線遅延 (ns)	0.71	0.88
ALU で減算を行う遅延 (ns)	6.63	6.63
合計 (ns)	7.34	7.51
比率	1.00	1.021

表 3 ALU Cascading 適用時の実行ステージの遅延時間

結果フォワーディングの配線遅延 (ns)	0.88
ALU で減算を行う遅延 (ns)	6.63
Cascading バイパス回路の遅延 (ns)	0.40
ALU で減算を行う遅延 (ns)	6.63
合計 (ns)	14.54
1 演算あたりの遅延 (ns)	14.54/2=7.27

表 2 に ALU Cascading なしの時の実行ステージの遅延時間の評価結果を示す．Cascading バイパス回路がある場合，ない場合に比べて 2.1%遅延時間が増加した．クリティカル・パスが実行ステージ以外にあるなどの理由によって，この 2.1%の遅延時間の増加が許容できる場合，ALU Cascading はそのプロセッサに実装可能となる．また，**表 3** に ALU Cascading 適用時の実行ステージの演算の遅延時間の評価結果を示す．ALU Cascading による演算の，1 演算あたりの遅延は 7.27ns となり，ALU Cascading なしの時の実行ステージの遅延時間である 7.51ns より小さくなった．従って，ALU Cascading はクロック周波数を最大値の半分まで低下させれば問題なく適用可能であることが分かる．

表 4 レジスタ・マップ表と等価なキャッシュのパラメータ

ライン・サイズ	2-byte or 5-byte
セット数	64
アソシアティビティ	ダイレクト・マップ
書き込みポート数	16 ポート
読み出しポート数	8 ポート
プロセス・ルール	0.09μm

表 5 レジスタ・マップ表の 1 ラインのビット数

ALU Cascading	なし	あり
Preg	8	8
Output inst. entry	8	8
Prev inst. entry	8 x 0	8 x 2
ALU inst. flag	0	8
合計	16	40

表 6 レジスタ・マップ表の消費エネルギー

ALU Cascading	なし	あり
消費電力 (nJ)	1.42	1.58
比率	1	1.13

4.3 追加回路の消費電力の評価

3.1 節で述べたように，ALU Cascading を適用した命令スケジューラでは，既存のレジスタ・マップ表にエントリを増設する必要がある．これによって消費電力が増加する．また，ALU Cascading を実行する組み合わせ論理も電力を消費する．これらのうち，ALU Cascading の電力消費の大半を占めるのは，前者の増設されたエントリであると考えられる．なぜならば，レジスタ・マップ表は全ての命令のデコード時に使用され，毎サイクル，全ての命令がアクセスするためである．また，ALU Cascading を適用する場合は，適用しない場合よりも 1 クロック・サイクルあたりに実行する命令数が増加するため，動作する ALU の個数が増加する．一見，これは消費電力を増加させるように思える．しかしながら，この演算はプログラムの実行に必要なものであり，ALU Cascading を適用しない場合でも，いずれ，実行されなければならないことに変わりはない．従って，ALU Cascading を適用する場合もしない場合も，プログラムを実行し終わった時点での ALU における総消費電力は変わらない．なお，ALU Cascading を行うことにより，クロック・サイクル時間の後半で演算を行う ALU においてグリッチが大きくなることが考えられる．しかしながら，この問題は，カスケーディング用マルチプレクサの切替えをクロック・サイクル時間の後半の開始に行い，クロック・サイクル時間の前半における入力を固定することによって，容易に回避することができる．

本節の評価では，レジスタ・マップ表をキャッシュに見立て，キャッシュ・アクセスあたりの消費エネルギーを計算するツールである CACTI3.0[10] を用いて，ALU Cascading 用の拡張のあり／なしで，レジスタ・マップ表が消費するエネルギーを評価した．

評価にあたっては，レジスタ・マップ表を**表 4** に示すようなキャッシュと等価であると考え，ライン・サイズのみ変化させて消費エネルギーを計算した．ライン・サイズは Preg のビット数，Output inst. entry のビット数，および Prev inst. entry のビット数により，**表 5** のように決定される．なお，実際には各エントリのビット数はもっと少ないが，CACTI の制限により 8 ビット単位に切り上げた．

表 6 にレジスタ・マップ表の 1 アクセスあたりの消費エネルギーの評価結果を示す．表から分かるように，レジスタ・マップ表の消費エネルギーは ALU Cascading を適用する場合は 1.13 倍の増加にしかならないことが分かる．これは，元のレジスタ・マップ表の消費エネルギーの大半がデコーダで占められていたため，エントリのビット数を増やすことによる消費エネルギーの増加が全体に及ぼす影響が小さいためである．

また，レジスタ・マップ表がプロセッサの消費電力のうちどれだけを占めるかは，Wattch[11] を用いたシミュレーションと，先行文献で調査した．5.1 節の評価条件のもとで，Wattch で SPECint2000 を実行した時のレジスタ・マップ表の消費電力は，プロセッサ全体の 1%であった．また，文献 12) によると，PentiumPro ではレジスタ・マップ表がプロセッサ全体の消費電力の 4%を占めるということが示されている．レジスタ・マップ表の消費電力が 1%しか占めていないならば，消費電力の増加はプロセッサ全体の 1%未満である．また，レジスタ・マップ表の消費電力が 4%を占める場合でも，やはり消費電力の増加は 1%未満である．4.1 節の最後に記したように，IPC 向上を消費電力削減に利用すると 3.7%の削減になるため，追加回路の消費電力の増加を差し引いても ALU Cascading によって消費電力削減は可能ということになる．

5. 関 連 文 献

ある ALU の出力を別の ALU の出力に接続し，1 クロック・サイクル中に複数の演算を行うことを提案している論文は多数ある．古くは，ベクトル・プロセッサの性能向上のために提案されている[5]．また，時代の流れに伴って，当該手法の適用対象も変化し，マルチメディア処理への適用[6]，GALS プロセッサへの適用[1] など提案されている．

しかし，このような演算を行わせる際に，out-of-order 実行を行うスーパスカラ・プロセッサの動的命令スケジューリングを考慮したものは少ない．参考文献 3) では，スーパスカラ・プロセッサにおいて，ALU Cascading と同様に，1 サイクル中に複数のデータ依存関係にある命令を実行する CHAIN という手法を提案している．しかし，そのアルゴリズムは命令列に対する複数回のスキャンやオペランドの出現回数の計数が含まれており，クロック・サイクル時間に大きな影響を与えずにハードウェア化をすることが難しいと考えられる．それに対し，我々はクロック・サイクル時間に大きな影響を与えないように配慮しつつ，ハードウェアの構成まで検討を行った．また，参考文献 4) では，同時にデコードされた命令に対して ALU Cascading で実行できる組を探し，それを 1 組の命令として，命令ウィンドウの 1 エントリに登録する手法が提案されている．この方法には，命令ウィンドウのサイズを増加させることができるという利点もある．しかし，ALU Cascading 可能な命令が同時にデコードされた命令に限定される点や，組となった命令を同時に発行しなくてはならない制限がある．これに対し，我々は上記の制限はなく，その上，1 対多の ALU Cascading 可能という利点もある．

6. ま と め

本論文では，ALU Cascading をスーパスカラ・プロセッサに実装する時に必要となる，任意の命令の組に対する ALU Cascading を行える命令スケジューラを提案した．提案する命令スケジューラを SPECint2000 を用いて評価した結果，ALU Cascading を行うと，IPC が平均で 3.8%向上するという結果になった．必要とされるプロセッサ性能が ALU Cascading 適用前と同じ場合，この IPC 向上を利用して，適用前よりクロック周波数を更に下げることができるため，ALU Cascading を利用した消費電力削減が可能となる．そのため，この IPC の向上は 3.7%の消費電力の削減と取ることも出来る．

また，Cascading バイパス回路を追加した場合，実行ステージの遅延時間の増加は 2.1%であることが分かった．クリティカル・パスが実行ステージ以外にあるなどの理由によって，この 2.1%の遅延時間の増加が許容できる場合，ALU Cascading はそのプロセッサに実装可能となる．また，ALU Cascading のために追加した回路が消費する電力はプロセッサ全体の 1%以下であり，IPC の向上から得られる 3.7%の消費電力の削減を食いつぶすほど大きなものならないという結果になった．

謝辞 本研究の一部は日本学術振興会科学研究費補助金基盤研究 S(課題番号 16100001) による．

参 考 文 献

1) 佐々木広, 近藤正章, 中村宏: GALS 型プロセッサにおける動的命令カスケーディング, 情報処理学会研究報告, Vol. 2005-ARC-164, pp. 67–72 (2005).
2) 尾形幸亮, 嶋田創, 中島康彦, 森眞一郎, 富田眞治: パイプラインステージ統合における ALU Inlining, 平成 18 年度情報処理学会関西支部支部大会講演論文集, pp. 203–206 (2006).
3) 孟林, 小柳滋: スーパースカラプロセッサにおける 動的 RENAME 手法と CHAIN 手法, 情報処理学会関西支部支部大会講演論文集, pp. 207–210 (2006).
4) 佐々木宏, 近藤正章, 中村宏: 命令グルーピングによる効率的な命令実行方式, 情報処理学会研究報告, Vol. 2006-ARC-170, pp. 73–78 (2006).
5) 長島重夫, 稲上泰弘, 阿部仁, 河辺峻: 動的チェイニングによるベクトルプロセッサの実効性能の向上, 電子情報通信学会論文誌, Vol. J74–D1, No. 12, pp. 836–845 (1991).
6) 山崎信行, 伊藤務, 内山真郷, 安西祐一郎: 柔軟なマルチメディア処理機構を有したリアルタイムプロセッサアーキテクチャ, 日本機械学会ロボティクスメカトロニクス講演会 '01, pp. 1–2 (2001).
7) 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: 行列に基づく Out-of-Order スケジューリング方式の評価, 情報処理学会論文誌 : ハイパフォーマンスコンピューティングシステム, Vol. 43, No. SIG 6(HPS 5), pp. 13–23 (2002).
8) Burger, D. and Austin, T. M.: The SimpleScalar Tool Set, Version 2.0, Technical Report CS-TR-97-1342, University of Wisconsin-Madison Computer Sciences Dept. (1997).
9) Stine, J.E., Grad, J., Castellanos, I., Blank, J., Dave, V., Prakash, M., Iliev, N. and Jachimiec, N.: A Framework for High-Level Synthesis of System-on-Chip Designs, *International Conference on Microelectronic Systems Education*, IEEE Computer Society, pp. 67–68 (2005).
10) Shivakumar, P. and Jouppi, N. P.: CACTI3.0: An Integrated Cache Timing, Power, and Area Model, Technical report (2001).
11) Brooks, D., Tiwari, V. and Martonosi, M.: Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, ISCA-27, pp. 83–94 (2000).
12) Manne, S., Klauser, A. and Grunwald, D.: Pipeline Gating: Speculation Control For Energy Reduction, ISCA-25, pp. 132–141 (1998).