計算機アーキテクチャ 150−19
（２００２．１１．２８）

制御フローコードを分離するプロセッサアーキテクチャの提案

吉瀬謙二[†,††] 片桐孝洋[†,††]
本多弘樹[†] 弓場敏嗣[†]

動作周波数と並列性の向上により年率55%という高いプロセッサの性能向上が数十年に渡って維持されてきた。今後も同様の性能向上を維持するために、10億個を越えるトランジスタという豊富なハードウェア資源を用いて、高い命令レベル並列性を抽出する新しいプロセッサアーキテクチャが必要とされている。本稿では、消費電力の削減と高速化を達成するプロセッサアーキテクチャとして、制御フローコードを分離する新しいプロセッサアーキテクチャを提案する。また、制御フローコードに加えて、アドレス計算のためのコードを分離するアーキテクチャの構想を述べる。これらのアーキテクチャは、命令流を明示的に複数の流れとして扱うという意味から**スーパー命令フローアーキテクチャ**と呼ぶことにする。

Processor Architecture to Enhance the Control Flow Code Execution

Kenji Kise,[†,††] Takahiro Katagiri,[†,††] Hiroki Honda[†]
and Toshitsugu Yuba[†]

Microprocessor performance has improved at about 55% per year over the past three decades. To maintain historical performance growth rates, future processors with more than one billion transistors must achieve higher levels of instruction-level parallelism. The aim of this study is to develop a novel processor architecture which enhances the control flow code execution. The architecture makes it possible to design low-power and high-performance processors. We name it a **super instruction-flow architecture** because the architecture processes the multiple instruction-flows efficiently.

1. はじめに

動作周波数と並列性の向上により年率55%という高いプロセッサの性能向上が数十年に渡って維持されてきた。今後も同様の性能向上を維持するためには、動作周波数と並列性をバランスよく向上させる努力が必要となる。文献3)では、現在の2GHzという動作周波数は、パイプライン段数を増やすことにより高々2倍程度にしか向上しないという研究結果から、命令レベルあるいはスレッドレベルの並列性を33%の年率で向上させ、15年後にはサイクル当たり50個の命令を並列に処理する必要があると主張している。Alpha21264を搭載する標準的な高性能サーバDS-10Lにおいて SPEC CPU2000を走らせた際の命令レベル並列性は2程度[1]であり、命令レベル並列性50を達成するためには約25倍の並列性を抽出しなければならないことになる。

近年、プロセッサ性能の向上を目指して、データ値予測などのさまざまな投機技術[10]が提案されている。投機技術は性能向上のために不可欠な手法となりつつあるが、本稿で議論するアーキテクチャは投機技術の利用を前提とするものではない。命令レベル並列性50を目指すためには、その土台を形成する新しいアーキテクチャが必要となる。このような流れにおいて、VLDP[11]、命令レベル分散処理[6]、Gridプロセッサ[7]といった新しいアーキテクチャの可能性が検討されている。

分岐命令や分岐条件を計算するためのコードを**制御フローコード**と呼ぶことにする。また、制御フローコードを形成する個々の命令を制御フロー命令と呼ぶことにする。本稿では、高い命令レベル並列性を達成するための土台となるアーキテクチャの構築を目指して、制御フローコードを分離する新しいプロセッサアーキテクチャを提案する。

プロセッサの内部では、命令パイプラインの早い段階で制御フローコードを、それ以外の命令と分離する。その後、制御フローコードは、それを専用に処理するハードウェアにより高速に処理される。制御フ

† 電気通信大学 大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications
†† 科学技術振興事業団 さきがけ研究21
"Information infrastructure and applications", PRESTO, Japan Science and Technology Corporation(JST)

ローコードを優先的に処理することで、分岐予測ミスのペナルティを軽減する。また、制御フローコードとそれ以外のコードとを分けることにより、それぞれの処理の流れに適した最適化を選択できるという利点が生じる。

2. 制御フローコードを分離するプロセッサアーキテクチャの提案

10 億個のトランジスタをチップ上に集積できる時代が迫っている。これら膨大なハードウェア資源を使い、演算器を多数集積するプロセッサ[7),11)] の可能性が検討されている。しかしながら、これらの強力な実行機構を有効に利用するためには、処理すべき命令を十分に供給し続けなければならない。

現在市販されている全ての高性能プロセッサが、分岐予測を用いて実行すべき制御フローを予測しながら処理を進めている。しかしながら、最先端の分岐予測を用いたとしても予測成功率は 95%程度[2)] であり、一定の割合の予測ミスを避けることはできない。分岐命令によるオーバヘッドを削減するためには、分岐予測ミスの回数を削減する努力に加えて、予測ミスペナルティの削減が重要となる。

本節で提案するアーキテクチャは、分岐予測ミスの回数ではなく、分岐命令の実行タイミングを早めることで予測ミスペナルティの低減を目指す。

2.1 命令セットの定義

Alpha AXP の命令セットアーキテクチャをベースとして、新しいアーキテクチャの命令セットを定義する。

制御フローコードをその他の命令と分離するために、制御フローコードの実行結果を格納するレジスタセット (32 個のレジスタ) をアーキテクチャステートに追加する。これを制御フローレジスタセットと呼ぶことにする。

提案プロセッサのアーキテクチャステートは、R0 から R31 までの整数レジスタセット、F0 から F31 までの浮動小数点レジスタセット、C0 から C31 までの制御フローレジスタセット、プログラムカウンタから構成される☆。

```
add  C6,1,C7  # add: C7 <= C6 + 1
beq  C3, $L5  # branch if C3 equal to zero
move R5, C4   # copy R5 to C4 (bis R5,R5,C4)
```

図 1 制御フロー命令の例
Fig. 1 Example of the control flow instructions.

制御フロー命令の例を図 1 に示す。ここに示した例は、一つの命令列ではなく、異なる 3 つの制御フロー命令である。オペランドとして制御フローレジスタが使われている命令を制御フロー命令として機械的に区別できる。

即値あるいは制御フローレジスタのみが制御フローコードのオペランドとして利用される。ただし、整数 (浮動小数点) レジスタの内容を制御フローレジスタにコピーする命令のみ入力オペランドとして整数 (浮動小数点) レジスタを指定できる。制御フローコードには、浮動小数点演算命令とロードストア命令は存在しないが、それ以外の算術論理演算やシフト命令などは整数レジスタを対象とした命令と同様のものを準備する。メモリからロードした値により分岐するようなコードの場合には、メモリの内容を整数レジスタにロードし、整数レジスタの値を制御フローレジスタにコピーする☆☆。

制御フローレジスタセットと制御フローコードが追加された事を除いて、命令セットアーキテクチャに変更点はない。

```
#define N 1000
int main(){
  int i;
  long long c, w;
  long long a[N],b[N],z[N];
  for(i=0; i<N; i++){
    w    = z[i];
    a[w] = b[i] * c;
  }
}
```

```
     bis C9,C9,C6      ### Control Flow Code ###
     lda C8,999        ### Control Flow Code ###
$L5:
     ldq R1,0(R5)      # R1 <= b[i];
     ldq R2,0(R4)      # R2 <= z[i];
     addq R5,8,R5      # b++;
     addq R4,8,R4      # z++;
     mulq R1,R7,R1     # R1 = b[i] * c;
     s8addq R2,R30,R2  # R2 = R2*8 + R30;
     stq R1,0(R2)      # store R1 to a[w]
     add C6,1,C6       ### Control Flow Code ###
     cmple C6,C8,C3    ### Control Flow Code ###
     bne C3,$L5        ### Control Flow Code ###
```

図 2 ソースコードの例 (上) と対応するアセンブラ (下)
Fig. 2 Sample source code and its assember.

制御フローコードを分離するプロセッサのサンプルコードを図 2 に示す。図 2 上に C のソースコードを、下に対応するアセンブラを示す。このコードは、配列 b の要素を定数 c と掛け合わせ、配列 z により指定された配列 a の要素にストアする。アセンブラの最初

☆ Alpha AXP のアセンブラでは整数レジスタの 1 番を$1 のように、浮動小数点レジスタを$f1 のように記述する。本稿では、提案アーキテクチャにおける記述を簡潔にするために、それぞれ R1, F1 という記述を用いる。

☆☆ レジスタセット間の通信回数を削減するために、制御フローコードにロード命令を追加する方式も考えられるが、議論が複雑になることを避けるため本稿では省略する。

の 2 行と最後の 3 行が制御フローコードとなる。この例では、制御フローレジスタと整数レジスタの間のデータの授受は発生していないので、制御フローコードとそれ以外のコードの処理を同時に進めることができる。

2.2 複数のデータ長を利用する実行ユニット

制御フローコードを処理する実行ユニットでは、インオーダ実行によるシンプルな命令パイプライン構成を採用し、これにより分岐命令の早期実行を目指す。

制御フローコードでは、ループの誘導変数を計算するためのコードや、データ値をゼロと比較する処理が多く現れる。この様なコードにおいては、Alpha AXP や IA-64 など、主流となりつつある 64 ビット長のレジスタを必要としない場合が頻繁に出現する。

レジスタファイルに書き戻されたデータに関して、そのデータ値のビット長の分布を測定した予備評価結果を図 3 に示す。SPEC CINT95, CINT2000, dhrystone の合計 21 本のベンチマークプログラムに関して、全てのレジスタ書き込みの割合を 100%として、書き込まれたデータ長 (符号あり) により、16、32、64 ビットに分類した比率を表示した。Alpha AXP アーキテクチャ、DEC C/C++コンパイラにより実行ファイルを作成した。64 ビット長の値に関しては、書き込まれるデータ値によりアドレス計算に関係するものと、そうでないものに分類して示した。この結果から、16 ビット長に収まるデータが全体の 29%～70%と多いこと、メモリ参照のアドレス計算に関係するデータを除外すると多くのデータが 16 ビット長に収まるという傾向を読み取ることができる。

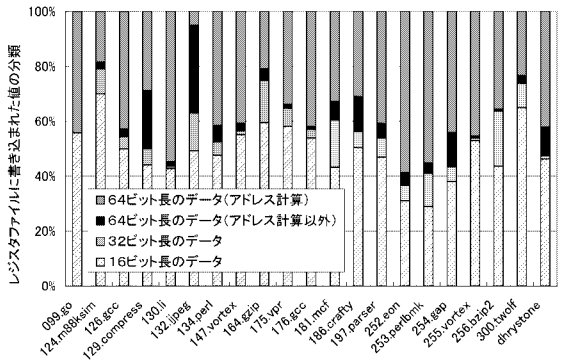


図 3 レジスタファイルに格納されたデータ値のビット長による分類
Fig. 3 The classification of the data stored in the register file by the bit length.

これらの考察から、制御フロー命令が利用するオペランドのレジスタ長により、適切なレジスタと演算器を選択できるようにする。具体的には、C0 から C31 までの 32 本の制御フローレジスタをデータ長に応じて次の 3 つの領域に分割する。C0 から C9 は 16 ビット長のレジスタ、C10 から C19 は 32 ビット長のレジスタ、C20 から C31 は 64 ビット長のレジスタとする。ただし 3 つの領域の最後の番号のレジスタ (C9, C19, C31) の値は 0 に固定され、これに関する書き込みは無視され、読みだしは常に値 0 を返すものとする。

演算に必要となる時間 (ALU の動作速度) は扱うデータのビット長に依存する。これを有効に利用するために制御フローコードの実行ユニットとしてカスケード ALU アーキテクチャ[8] を利用する。複数のデータ長を利用する実行ユニットの構成を図 4 に示す。図が複雑になることを避けるため、16 ビットのデータパスの細かい配線を省略した。16 ビットのデータ長の演算であればデータ依存関係の有無に関係なくサイクル当たり 4 つの命令を処理できる。32 ビットのデータ長の演算であればサイクル当たり 2 つの命令を処理できる。ビット長が少なくなるに従って、カスケードの段数が多くなり、それに従ってレジスタファイルのリードポートとライトポートの数が増加する。しかし、ポート数の多いレジスタファイルは個々のレジスタのビット長が短いので、ポート数の増加によるハードウェア量の急激な増加はおこらない。

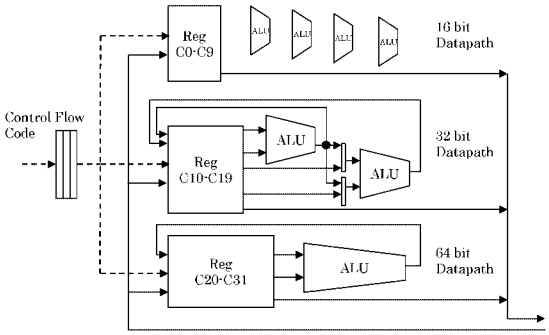


図 4 複数のデータ長を利用する制御フローコードのための実行ユニット (16 ビットのデータパスの詳細を省略)
Fig. 4 The execution unit for the control flow code using three data length.

16 ビット長、32 ビット長、64 ビット長のレジスタを利用する命令の間にはデータ依存関係が存在しないという制約を加えることで、異なるビット長のレジスタを利用する命令を同時に処理するできるようになる。図 4 に示す実行ユニットは、最大でサイクル当たり 7 個の制御フロー命令を処理できる。

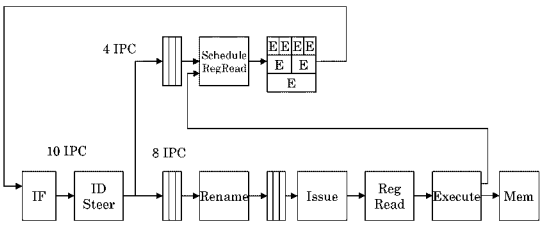
2.3 制御フローコードを分離するプロセッサアーキテクチャ

制御フローコードを分離するプロセッサの命令パイプラインを図 5 上に示す。網掛の部分は図 4 に示した制御フローコードのための実行ユニットを表している。比較のためにスーパースカラプロセッサの命令パイプラインを図下に示してある。

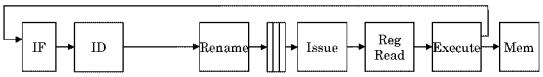
提案アーキテクチャにおいて、フェッチされた命令は続くステアリングステージにおいて制御フローコード

とそれ以外の命令に分けられ、それぞれの命令キューに格納される。制御フローコードの実行ユニットでは、命令キューから制御フロー命令を取り出してインオーダに処理を進める。それ以外の命令は、スーパースカラプロセッサと同様にアウトオブオーダ実行のバックエンドに投入される。

提案アーキテクチャは、レジスタセット間のデータ授受を必要とする move 命令を通信命令の一種として処理する。move 命令は、ステアリングのステージにおいて send と recieve という 2 つの命令に分割され、それぞれが異なる処理機構に送られる。



(a) 制御フローコードを分離するプロセッサ



(b) スーパースカラプロセッサ

図 5 制御フローコードを分離するプロセッサとスーパースカラプロセッサの命令パイプラインの比較

Fig. 5 Comparison of the instruction pipelines of a proposed processor and a superscalar processor.

2.4 サンプルコードの実行タイミング

図 2 に示したサンプルコードの 3 回分のループボディの実行タイミングを図 6 に示す。制御フローコードのみを示し、資源競合は発生しないとする。図 6 上に提案プロセッサにおける実行タイミングを、図下にスーパースカラプロセッサにおける実行タイミングを示した。

サンプルコードでは、ループの回数が 1000 回と固定されておりその値が 16 ビットに収まるので、誘導変数に 16 ビット幅のレジスタ C6 が割り当てられている。図 4 に示した様に、16 ビット長のレジスタを利用する制御フロー命令であれば、サイクル当たり 4 個の命令を実行できるので、提案プロセッサでは 3 回のループ実行が 6 サイクルで終了する。一方、スーパースカラプロセッサでは 10 サイクルを必要とする。このように、インオーダ実行を採用することによる命令パイプラインの短縮の効果と、短いデータ長のレジスタとカスケード ALU を採用することにより、早い時刻に分岐命令を処理できることがわかる。

2.5 コードの生成方針

提案プロセッサの性能はコード生成の手法に大きく依存する。制御フローコードを分離するための基本的

(a) 制御フローコードを分離するプロセッサ

(b) スーパースカラプロセッサ

図 6 サンプルプログラムの実行タイミングの比較

Fig. 6 Comparison of the executions of a sample program.

な方針を以下に示す。

- 分岐命令のオペランドを制御フローレジスタに変更することで分岐命令自身を制御フロー命令に変換する。
- 整数レジスタと制御フローレジスタの間のコピーを削減するように、分岐命令にデータを供給する命令を制御フロー命令に置き換えていく。
- プログラムの正しさを保証するために、必要に応じて move 命令を挿入する。

制御フローコードを分離する際に利用できる情報量から、ユーザが明示的に指示を与えるレベル、コンパイラが解析できるレベル、バイナリトランスレータが解析できるレベルという 3 つの階層に分類して、コードの生成手法を議論する必要がある。コード生成手法とその実現手法を検討は、今後の課題である。

3. 議　　論

3.1 複雑さと消費電力の削減

制御フローコードを分離するプロセッサアーキテクチャは、制御フローコードをメインの実行機構から分離することで、メインのパイプラインの複雑さを軽減する。図 2 に示したサンプルコードの場合には、ループボディを形成する 10 個の命令の中で、7 個の命令のみがアウトオブオーダのパイプラインに投入される。制御フローコードとそれ以外の命令との実行比率を一定に保つことができれば、リネーミングや発行の幅の削減が可能となり、ハードウェアの複雑さを緩和することができる。

制御フローを分離して分岐命令を早期に実行するこ

とで、プロセッサでフラッシュする命令の数を削減できる。このことは、性能向上をもたらすだけではなく、プロセッサ内の無駄な処理の削減を意味し、結果としてアプリケーションを実行するための電力を削減することができる。また、制御フローコードの実行ユニットはインオーダに命令を処理するため、アウトオブオーダの実行機構に命令を投入する場合と比較して、制御フローコードを実行するための電力を節約できる。

3.2 データ局所性の利用と GALS

パイプラインピッチの短縮、プロセスの微細化による配線遅延の増大などの理由により、グローバルクロックの生成が困難になりつつある。このような中、Globally Asynchronous Locally Synchronous(GALS) プロセッサ[4] に代表される分散処理が注目を集めている。提案したアーキテクチャは、制御フローコードを分離して、その計算に必要となるレジスタファイルとデータパスをメインの実行機構から切り離す。つまり、制御フローコードの利用するデータを局所的に閉じ込める分散処理を意味し、GALS との相性がよい。ただし、通信のオーバーヘッドを隠蔽するためには、制御フローレジスタセットと他のレジスタセットとの間の通信をできる限り削減する最適化が必要となる。

3.3 アドレス計算コードの分離

メモリ参照の際のアドレスは、動的に計算されることが多い。この参照アドレスが計算されなければ、メモリデータフローの依存関係を解析することができない。メモリ参照アドレスを早い段階で解決できるとすると、メモリデータフローに関しても、名前替えの手法を用いて、レジスタデータフローと同様に扱うことができる。

先の章では、制御フローコードを分離するプロセッサアーキテクチャを提案した。同様の手法を用いてアドレス計算を分離することを考えることができる。ただし、メモリ参照アドレスの計算がプログラムの本質的な部分であることもある。例えば、一般的なソーティングを考えると、これによりデータ値は変化しない。ソーティングにおける多くの処理は、ある規則に従ってデータの格納されているアドレスを変更することにあるため、その本質的な部分の多くは、アドレス計算と、メモリ間のデータコピーとなる。このような極端な例を除いたとしても、アドレス計算とそれ以外のデータフローとは、適度に分離しつつ、互いに連係しながら処理を進めることが望ましい。

制御フローコードとアドレス計算のためのコードを分離するプロセッサの構成を図 7 に示す。ここでは、アドレス計算のコードとメインのデータフローのための実行機構として命令レベル分散処理[6] の利用を想定して描いた。実行コアとして利用する方式には幾つかの選択肢がある。

3.4 関 連 研 究

文献 9) では、分岐命令に対応するフェッチ命令と呼ばれる命令を事なるパイプラインで処理するフェッチ分岐方式を提案している。本方式は、分岐命令だけではなく、制御フローに関連する複数の命令を分離するという点と、高い並列性をターゲットとしてレジスタセットを拡張するという点において文献 9) の手法を拡張したものと捉えることができる。

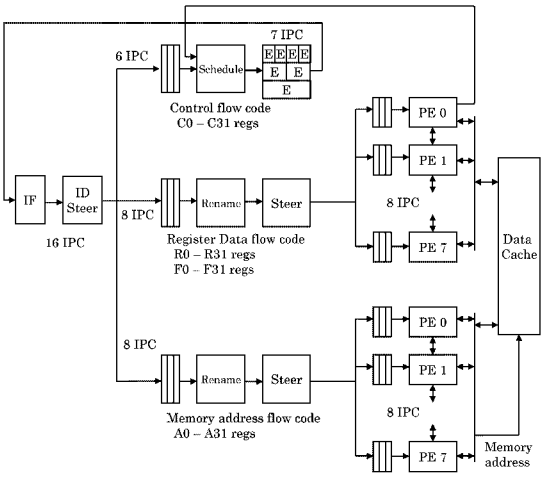


図 7 制御フローとアドレス計算を分離するプロセッサ

Fig. 7 Super instruction-flow architectuer to enhance the control flow code and the address calculation code execution.

4. 課　　題

4.1 分離コードの生成

提案アーキテクチャによるプロセッサの性能は、生成される実行コードの質に大きく依存する。初期の性能評価においては、手作業により得られる実行コード用いて評価することが考えられるが、最終的には、最適化コンパイラ (あるいはバイナリトランスレータ) の出力した大規模なコードを利用して評価する必要がある。

プロセッサの性能を引き出すため、コンパイラ (あるいはバイナリトランスレータ) に以下の要求を満たす仕組みを組み込む必要がある。

- 命令を分割する際に必要となる命令の複製とオーバヘッド命令の挿入を最小限に抑えること。
- レジスタセット間の通信をできるだけ少なくすること。
- 命令列をバランスよく複数のフローに分割すること。

4.2 アーキテクチャの方式検討

スーパー命令フローアーキテクチャの実現に必要となる以下の基本的な方式を検討する必要がある。

- 分岐予測ミスが起こったときに、分散された幾つかのフローの実行ユニットにおいて不必要な命令

のみをフラッシュする仕組み
- 正確な例外処理を実現するために、分離されたそれぞれのフローの命令によって更新される分散されたレジスタにより構成されるアーキテクチャステートを最新の状態に保つ手法
- 複数のフローの間の通信と同期を処理する機構

4.3 アーキテクチャの評価

クロックレベルのソフトウェアシミュレータを開発して、提案したプロセッサアーキテクチャを評価する必要がある。提案プロセッサの本質的な動作記述に加えて、要素技術として、カスケードALUアーキテクチャ[8]、トレースキャッシュ、YAGS分岐予測、Non-Uniform Cache Architecture[5]などを実装し、プロセッサ全体としての性能を評価する必要がある。

提案プロセッサの利点の一つは、命令流を分離してそれぞれのフローに適した処理方式を選択できる点にある。分離されたフローの実行ユニットとして適した方式と、それぞれの方式における適切なハードウェア規模(キャッシュのポート数やALUの個数などのパラメータ)を広大な設計空間の中から選択しなければならない。これらに加えて、本アーキテクチャに適した分岐予測機構、命令キャッシュ機構、データキャッシュ機構の選択が必要となる。

5. おわりに

動作周波数と並列性の向上により年率55%という高いプロセッサの性能向上が数十年に渡って維持されてきた。今後も同様の性能向上を維持するために、10億個を越えるトランジスタという豊富なハードウェア資源を用いて、高い命令レベル並列性を抽出する新しいプロセッサアーキテクチャが必要とされている。

本稿では、消費電力の削減と高速化を目指すプロセッサアーキテクチャとして、制御フローコードを分離する新しいプロセッサアーキテクチャを提案した。また、制御フローコードに加えて、アドレス計算のためのコードを分離するアーキテクチャの構想を述べた。

これらのアーキテクチャを、命令流を明示的に複数の流れとして扱うという意味から、**スーパー命令フローアーキテクチャ**と呼ぶことにする。

今後、本稿で提案したスーパー命令フローアーキテクチャの性能を明らかにするために、4章で述べた課題を検討していく。

参考文献

1) Desikan, R., Burger, D. and Keckler, S. W.: Measuring Experimental Error in Microprocessor Simulation, *Proceedings of the 28th annual international symposium on on Computer architecture*, ACM Press, pp. 266–277 (2001).
2) Eden, A. N. and Mudge, T.: The YAGS branch prediction scheme, *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society Press, pp. 69–77 (1998).
3) Hrishikesh, M. S., Burger, D., Jouppi, N. P., Keckler, S. W., Farkas, K. I. and Shivakumar, P.: The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays, *Proceedings of the 29th annual international symposium on Computer architecture*, IEEE Computer Society, pp. 14–24 (2002).
4) Iyer, A. and Marculescu, D.: Power and performance evaluation of globally asynchronous locally synchronous processors, *Proceedings of the 29th annual international symposium on Computer architecture*, IEEE Computer Society, pp. 158–168 (2002).
5) Kim, C., Burger, D. and Keckler, S. W.: An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches, *Proceedings of the 10th annual international conference on Architectural Support for Programming Languages and Operating Systems* (2002).
6) Kim, H.-S. and Smith, J. E.: An instruction set and microarchitecture for instruction level distributed processing, *Proceedings of the 29th annual international symposium on Computer architecture*, IEEE Computer Society, pp. 71–81 (2002).
7) Nagarajan, R., Sankaralingam, K., Burger, D. and Keckler, S. W.: A design space evaluation of grid processor architectures, *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, pp. 40–51 (2001).
8) Ozawa, M., Imai, M., Ueno, Y., Nakamura, H. and Nanya, T.: Performance evaluation of Cascade ALU architecture for asynchronous superscalar processors, *Proceedings of ASYNC-2001*, pp. 162–172 (2001).
9) 岡本秀輔, 曽和将容: 命令フェッチをプログラム制御するプロセッサ・アーキテクチャ, 情報処理学会論文誌, Vol. 39, No. 8, pp. 2509–2518 (1998).
10) 斎藤史小, 山名早人: 投機的実行に関する最新技術動向, 情報処理学会研究報告 計算機アーキテクチャ研究会 2001-ARC-145, Vol. 2001, No. 116, pp. 67–72 (2001).
11) 中村友洋, 吉瀬謙二, 辻秀典, 安島雄一郎, 田中英彦: 大規模データパスプロセッサの構想, 情報処理学会研究報告 計算機アーキテクチャ研究会 97-ARC-124, Vol. 97, No. 61, pp. 13–18 (1997).