計算機アーキテクチャ 117－13
（1996. 3. 5）

VLIW フェッチ PN 実行を行うプロセッサの設計

岡 本 秀 輔[†] 曽 和 将 容[†]

VLIW と PN に基づいたハイブリットプロセッサを提案する。このプロセッサは、プログラムを VLIW 方式でフェッチした後、内部では PN 方式を用いて実行を行う。実行されるプログラムは、VLIW と類似して、1命令が機能ごとの要素命令からなるが、それら要素命令間には実行のバリアはない。全ての要素命令レベルの先行関係は、他の方法で、明示的に記述されている。つまり、全ての要素命令の実行は静的にスケジュールされている。したがって、プロセッサはフェッチサイクルと実行サイクルを分けて処理を進めて行くことが可能である。本稿では、このプロセッサのアーキテクチャおよびソフトウエアシミュレータによる評価結果について述べる。

Instruction Level Parallel Processor based on VLIW and PN-superscalar

SHUSUKE OKAMOTO[†] and MASAHIRO SOWA[†]

A new hybrid processor based on VLIW and PN-superscalar is proposed. This processor fetches a program in the same way of VLIW processor, and it executes in the same way of PN superscalar processor. A program for this processor is similar to the ordinary VLIW program. But there is no execution barrier among the element instructions in a long word. The control dependency between any two instructions is written explicitly. So the execution order for all instructions is scheduled statically. And since the specification for this is not depended on the way of instruction fetch, the processor can run with the simultaneous execution of the element instructions which are fetched at the different cycle. This paper describes its processor architecture detail as well as the simulation result using software simulator.

1. はじめに

プロセッサの高速化を目的として、スーパースカラーや VLIW といった、命令レベルの並列実行を行うプロセッサが開発されて来ている。PN(Parallel Neumann) プロセッサは、この分類に入る小並列プロセッサである。このプロセッサは、機能別に分類された命令スレッドからなるプログラムに対して、動的に並列性の抽出を行うことなく、それを実行する。実行されるプログラムの並列性は、命令実行の先行関係を指定することにより、明示的に表現されているために、プロセッサは、実行時の並列性の抽出を行わない。したがって、シンプルなハードウエア構成とすることができ、動作周波数の高い、超高速なプロセッサとなる可能性を持っている。

これまでに設計された PN プロセッサでは、各スレッドの命令が別々の命令メモリに格納されており、独立した命令アドレスを用いてフェッチされることになっていた。したがって、実際に CPU をチップとして実現する場合には、命令アドレス用のピンがそれぞれのスレッドに対して必要であり、スレッドが多数となった場合にも、その数だけのピンを用意しなければならない。また、C や PASCAL といった言語プログラムの実行において、手続き呼び出しを行う際には、命令スレッド分のリターンアドレスをスタック上に積む必要があり、これを同時に行うためには、さらなるハードウエアを必要とする。

VLIW プロセッサも、PN 同様に動的な並列性を抽出することなく、命令レベルの並列実行を行う。VLIW のフェッチ方式は1アドレスを用いて行われるために、PN のような上述の問題が生じることはない。

しかしながら、VLIW は実行時において PN よりも不利な点を持っている。VLIW では長形式の1命令ごとに実行のバリアがあるために、その1命令の実行時間は、要素命令の中で最も長く実行時間を必要とするものに制限されてしまう。つまり、同時にフェッチされたのみで、問題に依存関係のない要素命令が、実行段階で他の要素命令から影響を受ける場合が出て来てしまう。PN では各命令に先行関係が記述されているために、VLIW のような問題は生じない。

† 電気通信大学 大学院 情報システム学研究科
The Graduate School of Information Systems,
The University of Electro-Communications

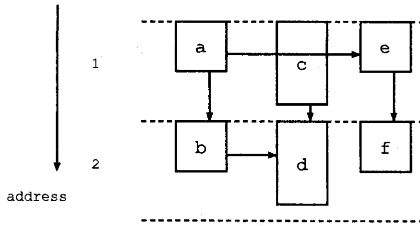

図1 fraction of a VFPE program

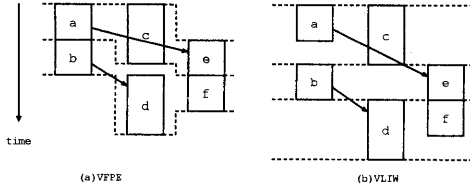

図2 Execution Image

これらの議論から、VLIWのフェッチの単純さとPNの効率的な並列実行を利用することで、フェッチと実行を明確に区別した高速プロセッサが実現可能であると判断し、このプロセッサを設計した。このプロセッサをVFPE(Vliw type Fetch and Pn type Execution)と呼ぶ。本稿では、このプロセッサのアーキテクチャおよびソフトウエアシミュレータによる評価結果について述べる。

2. VFPEプログラム

ここでは、VFPEプログラムの特徴およびスピードアップの要因について例を挙げながら説明する。

2.1 構造と実行イメージ

VFPEプログラムは、通常のVLIWプログラムに対して、各要素命令の実行に関する先行関係を付加した形式になっている。図1はプログラムの一部を示している。この図において、箱が要素命令を、矢印が実行の先行関係を示している。さらに、水平方向に並んだ要素命令は、同一アドレスで指定される。例えば、アドレス1を持った要素命令は、a, c, eであり、aからeへの矢印により先行関係が示されている。これは、a,c,eの3つの要素命令は同時にフェッチするが、eの実行開始はaが完了した後まで遅らされるという意味である。

図2(a)は、その実行イメージである。この図では垂直方向が経過時間を示している。この図では、eの実行開始がaの完了の後となり、次のアドレスを持つ要素命令であるbと 同時となっている。この時点では、b, c, eの要素命令が実行していることになる。つまり、別のアドレス上にある要素命令が、命令境界を越えて同時に実行していることになる。

図2(b)は、(a)と同一先行関係を持った命令を、VLIWで実行した時のイメージである。VLIWのこのプログラムでは、先行関係は要素命令の配置により示されている。したがって、eをaの後に実行開始させるためには、eをbと同じaの次のアドレス上に配置する必要がある。そして、実行時には、命令間のバリアを保証するために、bとeの実行の開始は、依存関係のないcの完了の後となっている。結果として、(a)に比べて時間のロスが生じている。

2.2 命令セット

VFPEプログラムのアセンブリ表現では、垂直方向に並んだ命令(箱)と関連する先行関係(矢印)は、1つのスレッドとなっている。そして、他の先行関係は、PNと同じ制御トークンとして表現されている[3),4)]。

VFPEプロトタイププロセッサ用のプログラムは、データ転送命令、算術命令、分岐制御命令の3つのスレッドからなる。以下の議論において、これらはそれぞれ、TUスレッド、AUスレッド、BUスレッドと呼ぶ。この構成はPNプログラムと同一である。

VFPEの命令セットは、HennessyとPattersonの本[1)]に出て来るDLXプロセッサをベースに設計されている。本の中のdata transfer命令をそのまま、TUスレッドへ割り当て、set conditional命令以外のarithmetic/logical命令をAUスレッドへ、そのset conditional命令とcontrol命令をBUスレッドへ割り当てている。

2.3 スピードアップ要因

上で述べた、" 長形式の1命令の境界を越えての実行" というVFPEの特徴は、要素命令の実行時間が、それぞれ異なる場合に現れる。命令パイプラインを持ったプロセッサでは、要素命令間の実行時間がそれぞれ異なったとしても、実行時間の最少単位は、パイプラインステージの時間スロットで決まる。このスロットの単位は、全てのパイプラインステージにおいて、共通となるので、長形式の1命令の境界を越えた実行は、図2(a)とは、異なった状況で生じる。この小節では、簡単な例を用いてこれを説明する。

まず、以下のようなDLXプログラムを考える。

```
loop:   addi r3,r3,#1
        slli r4,r3,#2
        add  r4,r9,r4
        slli r7,r3,#3
        addi r8,r7,#1
        slti r1,r3,N
        sw 0(r4), r8
        bnez r1,loop
        nop
```

このプログラムは、サイズNの配列a[]に対して、a[i]=i*8+1を計算する。ここで、レジスタ9(r9)は、a[]の先頭アドレス。r3は、配列のインデックスとループカウンタを兼ねている。r4は、ストア先のアドレ

スであり、r7 は、この式の計算結果である。

実行時には、パイプライン・ストールは起こらないために、1イタレーションあたりのパイプライン・サイクルは、9となる。これは、命令数と同じ数である。

次に 同内容の VLIW プログラムを考えて見る。ここで、コードは VFPE と同様に、DLX の命令を機能別に分解して、作成したものである。

```
loop:
 addi r3,r3,#1| nop          | nop
 slli r4,r3,#2| nop          | nop
 add  r4,r9,r4| nop          | nop
 slli r7,r3,#3| nop          | slti r1,r3,N
 addi r8,r7,#1| nop          | bnez r1,loop
 nop          | sw 0(r4),r8| nop
```

このプログラムは、"|" 文字によって区切られている。左端が、arithmetic/logical 命令であり、VFPE の AU スレッドに対応する。中央が、 data transfer 命令であり、VFPE の TU スレッドに対応する。右端が、 control instructions 命令等であり、VFPE の BU スレッドに対応する。

もし、VLIW プロセッサが、次の節の図5で述べるパイプラインと同じ構成持っていたと仮定すると。1イタレーションあたりのパイプライン・サイクルは、6となる。このサイクルも、VLIW 命令の数と同じである。

VLIW プログラムでは、データ依存性は、要素命令の配置によって行う。したがって、命令 *"sw 0(r4),r8"* は、 命令 *"addi r8,r7,#1"* の結果利用するために、この *addi* 命令の以降の長形式命令に配置しなくてはならない。そして、結果として、 *nop* 命令が *addi* 命令の次に挿入される。

最後に、VFPE のコードは以下のようになる。

```
loop:
 addi r3,r3,#1,.B| nop          | nop
 slli r4,r3,#2   | nop          | nop
 add  r4,r9,r4   | nop          | slti r1,r3,N,A.
 slli r7,r3,#3   | nop          | bnez r1,loop
 addi r8,r7,#1,.T| sw 0(r4),r8,A.| nop
```

このプログラムも各スレッドは、"|" 文字で区切られている。左端が、AU スレッド、中央が、TU スレッド、右端が、BU スレッドである。前にも述べたように、異なるスレッド間の制御依存は、PN と同じ制御トークンの通信で表現する[3),4)]。

このコードにおいて、AU スレッドの *"addi r3,r3,#1,.B"* 命令は、BU スレッドへのトークン送信の指定があり、*"addi r8,r7,#1,.T"* 命令は、TU スレッドへのトークン送信の指定がある。これらに対応して、TU スレッドの *sw* 命令と BU スレッドの *slti* 命令には、それぞれ、AU からのトークン受信の指定がある。トークン送受信の表記は、各命令の最後のオペランドとして、 "." 文字よりも前に書かれたものが受信指定、後ろに書かれたものが送信指定である。

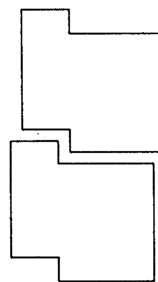

図 3 VFPE's Execution Image of 2 loop iterations

1イタレーションあたりのパイプライン・サイクルは、長形式の命令数と同じ5である。つまり、DLX と VLIW のどちらよりも短い。

この VFPE のプログラムでは、AU スレッドのループの底にある命令から、TU スレッドの *sw* 命令へのデータ依存は、トークン指定を用いて、表現されている。したがって、VLIW の様な不要な *nop* 命令は、AU スレッドには現れない。そして、実行時に、BU スレッドの分岐処理は、AU スレッドの底にある命令より早期に完了するために、TU スレッドの *sw* 命令と AU スレッドのループの先頭にある *"addi r3,r3,#1..B"* 命令は、 同時に実行を開始できる。

図 3はこの VFPE プログラムの実行イメージである。この図では、各多角形が、1イタレーションを表していて、上の多角形の底と他方の多角形の上部がフィットしている。このような実行を行うために、AU スレッドの底の命令から TU スレッドへの依存関係があるにもかかわらず、パイプライン・ストールは起こらずに、長形式の命令数と同じ数のサイクルで実行が行われる。

まとめると、この例では、VFPE は1イタレーションあたり、DLX に比べて4サイクル短く、VLIW と比較しても1サイクル短い。1イタレーションあたりの差であるから、イタレーション数 N に応じて無視できない大きさになる。

3. VFPE プロセッサ

この節では、VPFE プロセッサ・アーキテクチャの詳細について、特に、命令フェッチの機構と命令パイプラインに焦点をあてて述べる。

3.1 基本構造

図 4は、プロトタイプ VFPE プロセッサの基本構造を示している。TU, AU, BU はそれぞれ、スレッド実行ユニットである。これらは、レジスタを共有してデータ通信を行う。

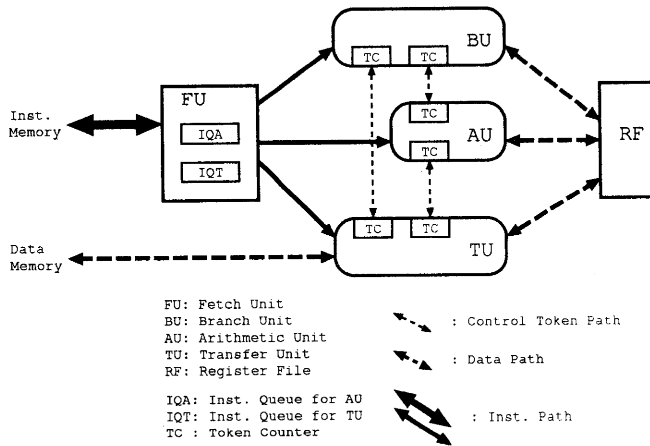

図4 VFPE Basic Structure

VFPE プロセッサでは、実行制御は PN プロセッサと同様に、プログラムカウンタとトークンカウンタ (TC) で行う[3)]。PN プロセッサでは、各実行ユニットごとに、1つのプログラムカウンタが必要なのに対して、VFPEプロセッサでは、全体で1つだけ必要な点が異なる。

FU は、長形式の命令をメモリから VLIW と同方式でフェッチし、それらを3つに分けた後に、各実行ユニットへ供給する。FU には、AU と TU スレッド用に、2つの命令キューがあり、トークン待ちなどである実行ユニットがストールした時に、他方の実行ユニットに対して、途切れなく命令を供給するために、これらのキューが使われる。つまり、ストールしたユニットの命令が、キューに蓄えられ、他方の実行ユニットには命令の供給が続けられる。

ストールから復帰した後は、その実行ユニットはキューの先頭の命令から実行を再開し始める。この時、ストールしなかった方の実行ユニットは、他のフェッチサイクルでフェッチされた命令を実行している。言い替えると、この命令キューの機構を用いて、長形式の命令の境界を越えた実行を、VFPE は行うことができる。

これとは、別に、BU スレッド用の命令キューはない。BU 実行ユニットがストールした時には、FU もストールする。これにより、分岐処理に関連した、命令フェッチを、BU のコードで制御することができる。

まとめると、VFPEプロセッサは、VLIWプロセッサと同方式でプログラムをフェッチし、PN プロセッサと同じ方式で実行を行う。

3.2 命令パイプライン

図5は、VFPE の命令パイプライン構造を示している。IF ステージは、FU で処理されるので、このステージは全てのパイプラインで共有されることになる。このステージでは、上述のように長形式命令が3つに分けられる。AU(または TU) が ID ステージでストールした場合は、関連する命令が命令キューに保存されていく。もし、BU がストールした場合は、FU で行われる IF ステージもストールする。

kind	send stage
AU → TU	ID
AU → BU	EX
TU → AU,BU	MEM(load) ID(store)
BU → AU,TU	ID

表1 stages to send token

各実行ユニットは機能別に分けられているために、各パイプラインはそれぞれに最適な構造を持っている。例えば、BU のパイプラインでは、計算が ID ステージで終了するために、EX ステージはない。BU の WB ステージは set conditional 命令用である。また、AU の EX ステージの結果は、レジスタバイパス機構を用いているので、次のパイプライン・サイクルで利用可能である。加えて、全てのパイプラインにおいて、同サイクルの WB ステージの結果は、ID ステージで利用可能である。

全ての ID ステージは、トークンの受信をチェックし、もし、必要なトークンが到着していない場合は、その ID ステージはストールする。

表1は、トークン送信のためのステージをまとめたものである。AU は TU に対して、ID ステージでトークンを送る。これはレジスタバイパスを考慮にいれているためである。AU から TU へのトークン送信とレジスタバイパスの関係が、図6(a) に示されている。AU の ID ステージで送られたトークンは、次のサイクルまでに、TU へ到着する。TU が即座にこれを受信すれば、TU の次の EA ステージで AU の EX の結果を利用できる。図6(b) は AU の ID ステージから BU へ送る場合である。BU には EX ステージがないために、レジスタバイパスにより AU の EX ステージ

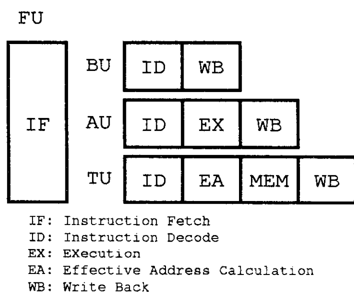

図5 Pipeline Stage for VFPE

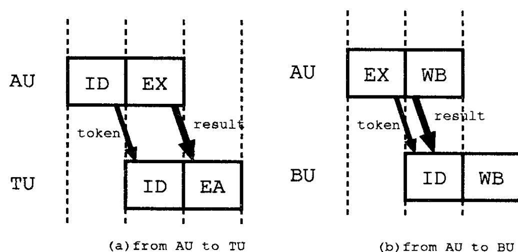

図6 token and register bypass

の結果を次のサイクルで利用することが出ない。したがって、AUからBUへは、EXステージでトークンを送る。AUのWBステージとBUのIDステージが同サイクルとなるので、BUのIDステージはストールすることなく処理を進める。

一方、BUはIDステージで全ての計算を終了するために、IDステージでトークンを送る。TUでは、トークンを送るステージはloadとstoreの命令により異なる。load命令では、TUにレジスタバイパス機構がないために、トークンの送信はMEMステージで行う。しかし、store命令は、レジスタを書き込むことがないために、IDステージでトークンを送信する。

4. 性能評価

提案したプロセッサが正しく動作をすることをテストし、性能を評価するするために、レジスタバイパスを利用した命令パイプラインを持つプロセッサとして動作する、ソフトウエア・シミュレータを実現した。性能の比較として、同じく命令パイプラインを持つDLXプロセッサとVFPEと同じ機能分けを行った実行ユニットを持つVLIWプロセッサのシミュレータを設計実現した。性能評価は、いくつかの簡単なプログラムの実行をトレースすることで行った。

表2は評価に用いたプログラムサイズをビット単位で表したものであり、表3は、実行時間をパイプライン・サイクルで示したものである。DLXに対するスピードアップが括弧で括られている。

プログラムPRIMEは100個の素数を計算するものであり、プログラムQUICKは再帰呼出しを用いて、意図的に作られた100個のデータをソートするものである。プログラムADDは、配列の100個の要素を単純に加算するものであり、ソフトウエア・パイプライン技法を用いて最適化されている。

これらの結果から、VFPEのプログラムサイズは、VLIWよりやや小さいか、ほとんど同じである。そして、VFPEはDLXと比較して、1.6から2.8倍高速に実行していることが分かる。加えて、プログラムADDを除いてVLIWより高速に実行している。プログラムADDのメインループは、ソフトウエア・パイプラインにより、VFPEとVLIWのどちらも*nop*なしで、最適化されているために、結果が同じとなった。

5. まとめ

本稿では、VLIWプロセッサとPNプロセッサの長所を利用たハイブリッド・プロセッサVFPEを提案した。このプロセッサは、VLIW構造を持った特殊なプログラムを、VLIWプロセッサ方式でフェッチし、PNプロセッサ方式で実行する。このプロセッサは、動的な並列性の抽出を行わないために、ハードウエアはシンプルになり、超高速スピードのプロセッサとなる可能性を持っている。性能評価として、シミュレーション結果は、VFPEがDLXに比べて、1.6から2.8倍高速に実行することを示した。そして、いくつかのケースでVLIWより高速な実行を示し、VLIWより遅くなるケースはなかった。

このシミュレーションでは、キャッシュミスを考慮にいれていないので、各命令の実行時間は、トークン待ちのストールを除き一定である。そして、今回は浮動少数点演算は行わないという前提を置いたので、1つのスレッド内の命令の実行時間は全て同じである。キャッシュミスや浮動少数点演算を考慮にいれた場合は、更なる時間のロスが生じる。しかし、命令間の実行時間に差ができ、命令境界を越えた実行というVFPEの特徴が顕著になるので、ロスの総時間は、VLIWよりも短くなるものと考えている。現在この点について、研究を進めている。

program	DLX	VLIW	VFPE
PRIME	2176	5472	5400
QUICK	4160	10176	9612
ADD	1280	3168	3240

表2 program size(bits)

program	DLX	VLIW	VFPE
PRIME	773128	381524(2.03)	290585(2.66)
QUICK	36353	24798(1.47)	22928(1.59)
ADD	638	231(2.76)	231(2.76)

表3 execution cycle

参 考 文 献

1) J. L. Hennessy and D. A. Patterson : "Computer Architecture – A Quantitative Approach", Morgan Kaufmann(1990).
2) M. Johnson: "Superscalar Microprocessor Design" , Prentice Hall(1991).
3) M. Sowa et al. : "Parallel Execution on the Function-Partitioned Processor with Multiple Instruction Streams", Trans. IEICE , D-I, Vol. J73-D-I, No.3, pp.280 – 285(1990).
4) T. Arita et al. : "Performance of the PN Superscalar Processor Estimated by Simulation", Trans. IEICE, D-I, Vol. J74-D-I, No.9, pp.635 – 643(1991).
5) H. Ito et al. : "Access Contention to Shared Memory Units in a Processor Extracting Fine-Grained Parallelism", Trans. IEICE, D-I, Vol.J74-D-I, No.11, pp.781 – 783(1991)