第３章　命令レベル並列処理方式アーキテクチャ "ジェットパイプライン(Jetpipeline)"

§ 3.1　緒言

本章では、第２章で述べたスーパースカラとVLIWの２種類の命令レベル並列処理方式とベクトル処理方式とを融合した、汎用かつ高速なCPUアーキテクチャであるジェットパイプラインを提案し、そのハードウエア、ソフトウエアについて考察する。

第２章でも述べたように、近づきつつある逐次処理の限界を打破するべく各種の並列処理方式が研究され、実用化されつつある。特に現在のプログラミングの考え方を大きく変えることなく高速化が達成可能な命令レベル並列処理方式の中には、現在広く使用されつつあるものもある。命令レベル並列処理方式の代表的なものとして、スーパースカラとVLIWが挙げられるが、これらはそれぞれハードウエアとソフトウエアの間のトレードオフをどちら側におくかにより、特長および欠点が存在している。それらの特長をのばし、欠点を補うべく本論文で提案するのが、ジェットパイプライン・アーキテクチャである。

ジェットパイプライン・アーキテクチャはスーパースカラとVLIWの中間的なハードウエア構成と制御方法を持ち､併せてベクトル処理機能も装備することにより､幅広い応用プログラムに対して高性能を発揮することが期待できる。本章では、まずハードウエアの構成について、各処理ユニットごとに詳細に述べた後、ソフトウエア環境について、特に逐次プログラムの並列化やスケジューリングについて述べる。

§ 3.2　従来の命令レベル並列処理方式とその問題点

第２章で述べたスーパースカラとVLIWの２種類の命令レベル並列処理方式には、それぞれ利点と欠点が存在する。

スーパースカラ・アーキテクチャは、ハードウエアにより命令間の依存関係を検知し並列動作可能な命令を複数の演算器に発行することにより並列処理を行う。この方式は機械語の互換性を保ったまま適用することができ、従来のプログラムコードをそのまま使用してもある程度の高速化が可能である。しかしながら、逐次的な

命令列の中から並列性を検出し、その結果を演算器に振り分けるための回路は非常に複雑になりがちである。

一方、VLIWアーキテクチャは実行時に並列性を検出するハードウエアを持たず、また長命令語内の各フィールドが直接それぞれの演算器を制御する方式をとっているため、これらのデコードも簡単である。従って、スーパースカラと比較してハードウエア構成を単純化することが可能である。しかしながら、長命令語中の各制御フィールドは固定されているため、もしプログラム中に有効な並列性が存在しない場合等においては、柔軟な対応をとることができず、長命令語中に無駄な部分を生じてしまうことがある。

ベクトルアーキテクチャでは、プログラムをベクトル化してパイプライン方式の演算器を効率よく動作させることにより高速な演算能力を得ているが、もしプログラム中でベクトル化できる部分の割合（ベクトル化率）が低い場合は、演算器が本来持つ性能を発揮することが不可能であるという問題点がある。したがって、ベクトル化により高速化可能な部分はこれにより高速化し、それ以外の部分は命令レベル並列処理を併用することにより、より汎用性のある計算機が構成可能であると考えられる。

§ 3.3　ジェットパイプライン・アーキテクチャの基本概念

本節では、前節で述べた各種の高性能計算機アーキテクチャの問題点を解決し、より汎用性のある高速処理を実現するために、代表的な命令レベル並列処理方式であるスーパースカラとVLIW方式の特徴を生かし、さらに両方式の弱点を補うことができ、かつ数値計算の高速化を図るためにスーパーコンピュータのベクトル演算パイプライン方式をも組み合わせた新しいアーキテクチャであるジェットパイプライン・アーキテクチャを提案する[Katahira 92,94a,94b]。ジェットパイプライン・アーキテクチャの概念図を、図3-1に示す。このアーキテクチャは命令パイプラインを複数個持ち、それらに割り付けられた命令がスーパースカラとVLIWの中間的な方式により、整数演算用のALUや浮動小数点演算用の演算パイプラインなどから構成される演算部を制御する形式をとっている。なお、命令パイプラインに投入される命令にはスカラ処理に対応するスカラ命令と、演算パイプラインを最高効率で利用するためのベクトル命令の両方が存在する。本システムは、ある意味では共有メモ

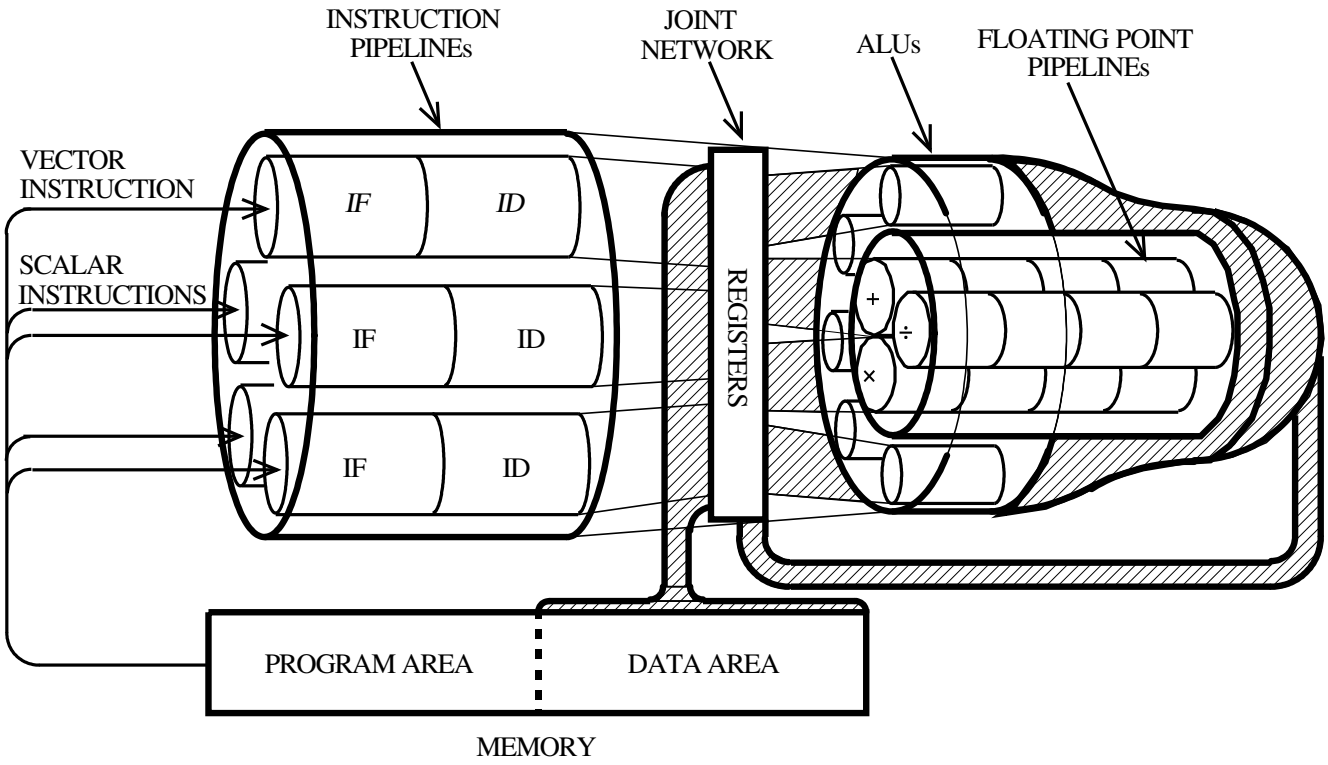


図3-1　ジェットパイプライン・アーキテクチャの概念図

リの代わりにレジスタによって相互通信を行う密結合マルチプロセッサととらえることもできる。

命令パイプラインに投入する命令の種類によって、本システムは並列計算機の各種の動作モードと対応させることが可能である。例えば同種の命令を並列に投入すればSIMD(Single Instruction stream, Multiple Data stream)方式とみなすことができ、別種の命令ならばMIMD(Multiple Instruction stream, Multiple Data stream)方式と見ることができる。プログラムに並列性が存在しない場合には１つの命令パイプラインのみに命令が投入され、それ以外の命令パイプラインにはNOP命令が投入されるが、この場合はSISD(Single Instruction stream, Single Data stream)方式となる。

本ジェットパイプラインからなる計算機のためのコンパイラは、プログラムの持つ並列実行可能な命令をVLIWと同じ考え方で１つにまとめ、命令のスケジューリングを行う。そして、プログラムの実行の際には、１つにまとめられた個々の命令を各命令パイプラインに割り当て、並列処理を行う。したがって、コンパイラが静的に命令のスケジューリングを行うことになる。ハードウエアによって並列動作可能な命令を検出しスケジューリングを行うスーパースカラ方式があるが、これを採用しなかった理由として、以下の３点が挙げられる。

(1) 命令の互換性を考える必要がない。

(2) 並列性の検出に要する複雑なハードウエアを省略することができ、その分をベクトル処理部分に振り分けることができる。

(3) 実行に多くのクロック数を要するベクトル命令をハードウエアによって無駄なくスケジューリングするのは困難である。

また、VLIW方式とも異なる点としては、各命令パイプラインは独立であり、全体として１つの長命令語により制御するのではなく、独立な複数の命令語の組によって制御が行われることがあげられる。したがって、VLIWのように命令中で各部分を制御するフィールドが固定されていないため、より柔軟な並列処理が可能となり、また長命令語中の有効利用されない空きフィールドの減少にもつながる。

なお、表3-1に、ジェットパイプライン・アーキテクチャとスーパースカラ、VLIWの比較を示す。表中に示したとおり、複数の独立した命令を同時にフェッチするのはスーパースカラ的であるが、スケジューリング方式ではソフトウエアによるVLIW方式を採用している。従って、専用コンパイラのサポートが必要であること

表3-1 スーパースカラ、VLIWとジェットパイプラインの比較

	Instruction fetch	Stage with parallel processing	Controling/ Scheduling policy	The support of compiler	Instruction compatibility
Superscalar	Multiple instructions are fetched per cycle	Execution stage e.g. IU, FPU, Load/Store	The scheduling is performed by hardware at run time. (The kinds of concurrent operations are restricted.)	It is unnecessary, but specialized compiler can improve the system performace.	Yes
VLIW	One long instruction is fetched per cycle.	Execution Stage	The scheduling is performed by software at compile time. (The operation of each field in instruction is fixed.)	Necessary	No
Jetpipeline	One group with multiple instructions is fetched per cycle.	Execution Stage	The scheduling is performed by software at compile time. (Any combination of concurrently executed instructions is possible.)	Necessary	No

もVLIWと同様である。

本アーキテクチャを用いることにより、ベクトルアーキテクチャで高速化できるベクトル化可能な部分に加えて、ベクトル化不可能な部分についてもVLIW的な並列化を行うことによって高速化を図ることができる。もちろん、並列性が十分にある場合には、ベクトル化と並列化を組み合わせることによって、より一層の高速化を図れる。

一方、実際のプログラムには、例えば繰り返し間で依存関係のないループ構造のような高レベルの並列性をほとんど含んでいない場合もある。従来のベクトルアーキテクチャのみではこのようなプログラムはまったく高速化できなかったが、本アーキテクチャでは着目する並列化のレベルを柔軟に変更することにより、例えば配列のメモリアドレス計算とその他の演算処理といった低レベルの並列性を有効利用し高速化するといったきめ細かな対応を行うことも可能である。

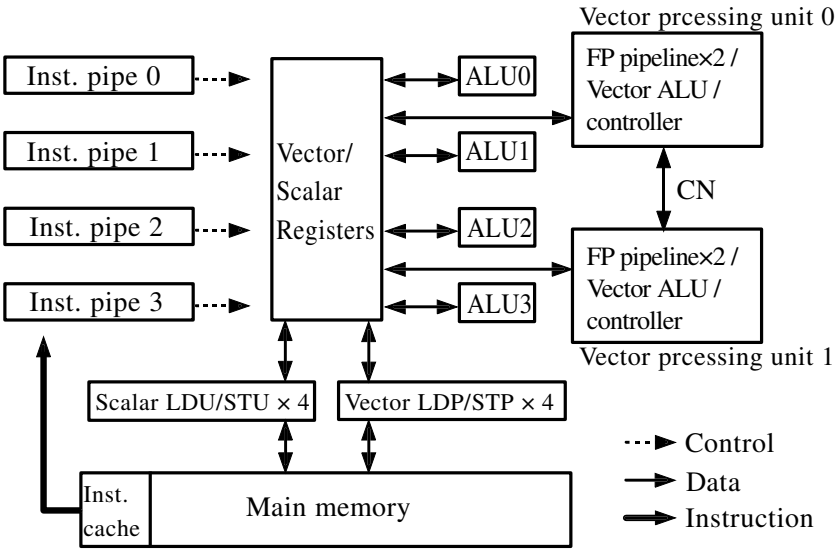
なお、ジェットパイプラインという名称は、図3-1の概念図において、命令を空気に、入力データを燃料に、演算結果を燃焼ガスに見立てたときシステム全体がジェットエンジンのように見えることから名付けられた。

§ 3.4　ハードウエア構成

本節では、前節で述べたジェットパイプライン・アーキテクチャの概念を実現するための具体的な機能ブロックレベルのハードウエア構成について述べる。なお、以下では命令パイプラインを4本持つシステムの各機能ブロックについて、その概略を述べる。これは、実際のプログラムにおいて、基本ブロック内で局所的な命令スケジューリングのみを用いて利用可能な命令レベルの並列性は一般に4以下であることが多いという報告[Jouppi 89][Johnson 91][Saitoh 94]に基づいて決定した。また、ハードウエア構成は図3-2,3-3に示すような初期のプロトタイプアーキテクチャ[Katahira 92,94a]を決定した後に指摘された問題点を解決し、より現実的にするために変更が加えられて、図3-4,3-5に示すような現在のアーキテクチャ[Katahira 94b]に至っている。その相違点についても述べる。

3.4.1　整数ユニット(Integer Unit,IU)

整数ユニットは命令パイプラインと整数演算器、アドレス演算器を含むユニット



Inst. pipe : Instruction pipeline
Inst. cache : Instruction cache
FP pipeline : Floating point calculation pipeline
CN : Interconnection network
LDU/STU : Load unit/Store unit
LDP/STP : Load pipeline/Store pipeline

図3-2 機能ブロックレベルの構成（プロトタイプ）

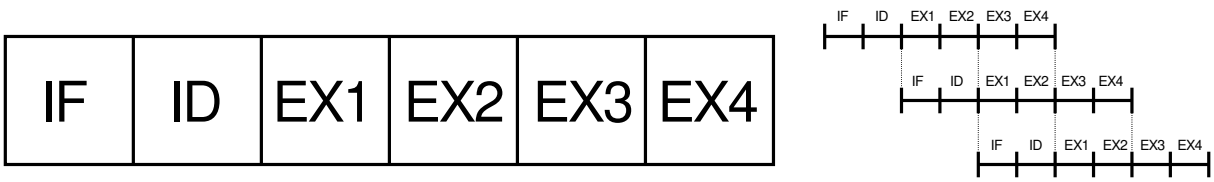
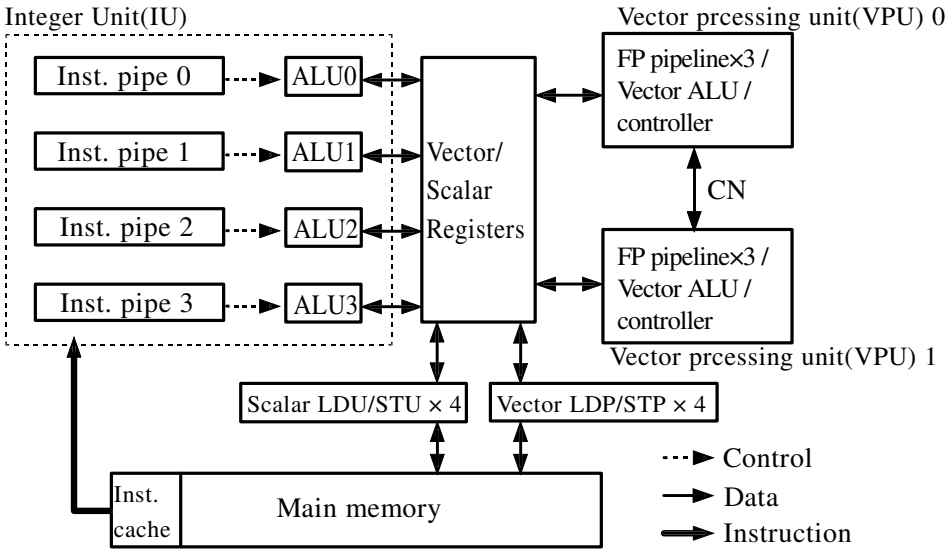


図3-3　命令パイプラインの構成（プロトタイプ）



Inst. pipe : Instruction pipeline
Inst. cache : Instruction cache
FP pipeline : Floating point calculation pipeline
CN : Interconnection network
LDU/STU : Load unit/Store unit
LDP/STP : Load pipeline/Store pipeline

図3-4　機能ブロックレベルの構成図（改良版）

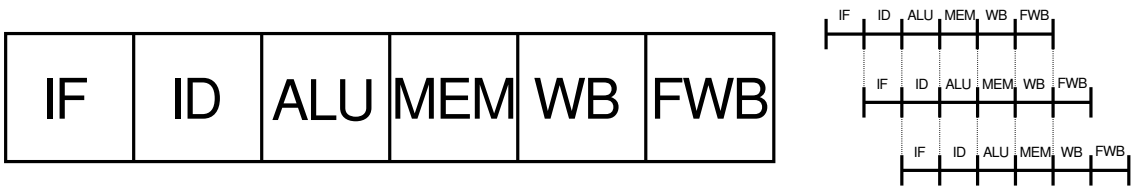


図3-5　命令パイプラインの構成（改良版）

で、ジェットパイプライン・アーキテクチャにおけるすべての命令のフェッチおよびデコードと、スカラ整数演算命令、分岐命令の実行処理を行う。

命令パイプライン(Inst. pipe,IP)は、初期設計においては図3-3のような6段構成で、2段づつオーバーラップして実行するよう構成されていた。これは本パイプラインの設計の基本としたスタンフォードMIPSアーキテクチャ[Hennessy 81,84] [Gross 83]のものと同一である。しかしながら、この構成はアンダーパイプライン[Jouppi 89]と呼ばれ、現在広く使用されているプロセッサにおいてこのような構成をとっているものはなく、性能評価において現実的な比較が困難であるという問題から、MIPS R3000 CPUのアーキテクチャ[Kane 88]等を参考にして設計を見直し、現在では、図3-5に示す一般的なIF, ID, ALU, MEM, WB, FWBといった6段構成で、各段がオーバラップして動作するような構成に変更した。各ステージにおける動作の概略を以下に示す。

IF(Instruction Fetch)ステージは命令フェッチを行う。すなわちPCから命令の存在するアドレスをメモリに送出し、メモリから命令を受け取る。4本のIPがメモリ上の連続する4命令を並列にフェッチする。このとき、IP0のフェッチする命令のアドレスは4で割り切れるものでなければならない。

ID(Instruction Decode)ステージは命令のデコードを行う。IFステージでフェッチされた命令はこのステージで解読され、レジスタをオペランドとする場合はレジスタ値の読み出しも行われる。

ALUステージでは、ALUを用いた演算が行われる。これは整数演算、論理演算ばかりではなく、メモリアクセス命令の実効アドレス計算にも使用される。また、ベクトル命令はこのステージにおいて必要なパラメータがベクトル処理制御ユニットに送られ、実行開始される。

MEMステージでは、ロード・ストア命令におけるメモリ参照が行われる。メモリアクセスを行わない命令の場合、このステージは何も実行しない。

WBステージでは、演算結果やメモリ参照の結果のレジスタへの書き込みが行われる。MEMステージ同様、レジスタに書き込まない命令では何も実行されない。

FWBステージは浮動小数点演算命令の結果のレジスタへの書き込みが行われる。浮動小数点演算命令以外ではこのステージは何も実行しない。

整数演算器(ALU)はスカラ命令の実行において 繁に利用される。たとえば、

MIPSアーキテクチャにおいてもっとも実行 度の高い命令は整数ALU命令で、全体の40 ~ 50%を占めることが報告されている[Gross 88]。そのため、ALUは各命令パイプラインに 1 つづつ対応させて配置した。ALUはパイプラインのALUステージにおいて、整数演算命令、論理演算命令、およびメモリアクセスの際のアドレス計算に使用される。また、ALUの演算結果を後続の命令がすぐに使用できるように、図3-6に示すようなフォワーディングのためのバイパスも用意されている。

分岐命令における分岐先アドレスの計算のためには、専用のアドレス演算装置が用意されている。これは、1 命令で比較と分岐を行うcompare&jump命令に対応するためである。初期の構成では、命令パイプラインのオーバラップが 2 段ごとだったため、分岐先アドレスの計算もALUにより行っていたが、命令パイプラインの変更に伴いALUを使用することが困難になったためアドレス演算器を追加した。また、分岐命令の直後には遅延スロットが存在し、常に分岐命令直後の 1 命令は実行される。分岐命令の実行の様子を図3-7に示す。

3.4.2 ベクトル処理ユニット(Vector Processing Unit,VPU)

ベクトル処理ユニットは、加減算用、乗算用、除算用の 3 つの浮動小数点演算パイプライン(FP calc. pipe)とベクトル整数演算用のALU、及びそれらに併設されるベクトル演算コントローラから構成される。このベクトル処理ユニットは、隣接する 2 つの命令パイプライン間で共用される。従って、隣接する命令パイプライン間においては同種のベクトル命令は同時に実行できない。また、浮動小数点演算パイプラインは、ベクトル命令とスカラ命令の両方で共用されるため、ベクトル浮動小数点演算命令実行中は同種のスカラ浮動小数点演算命令は実行できない。ただし、ベクトル演算命令は、各演算パイプラインに併設されたベクトル演算コントローラによって実行制御が行われるので、いったんIUによりベクトル演算が開始させられた後は独立して動作する。従って、浮動小数点演算以外のスカラ命令とベクトル演算命令の並列実行が可能である。各演算パイプライン間にはチェイニング演算のための相互結合ネットワーク(CN)も存在する。

この浮動小数点演算パイプラインにも初期の構成から変更が加えられている。初期の構成では浮動小数点演算パイプラインは加減算用と乗除算用の 2 本であったが、乗算と除算を一つのパイプラインで演算する仮定は現実的ではなく、また除算

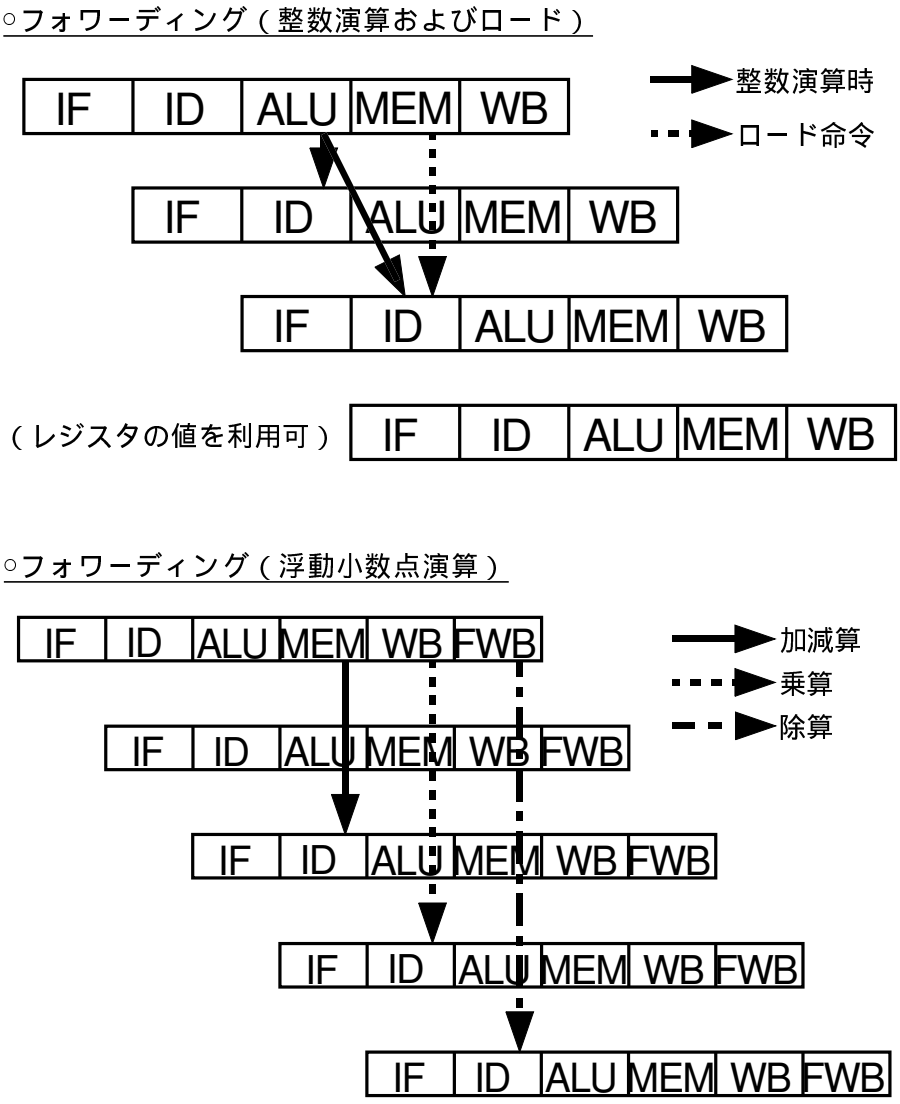
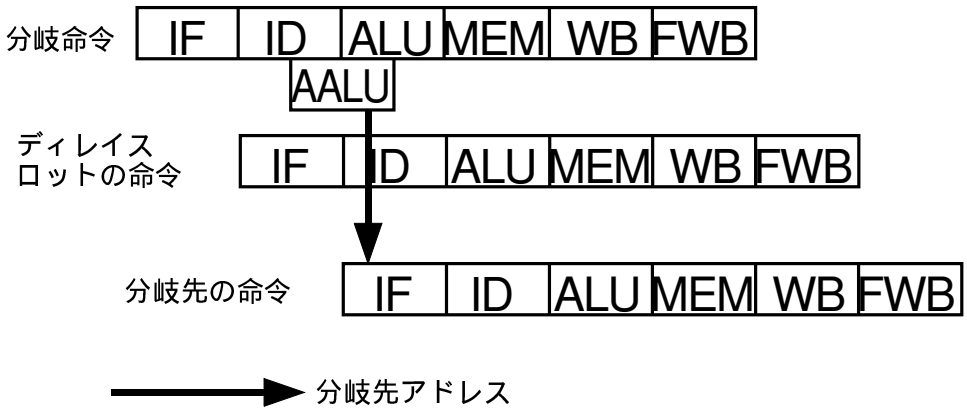


図3-6　フォワーディング

○分岐命令実行時のパイプライン動作



○実際のコード例（**JR**が分岐命令）

```
            JR  L1      (a)
            Inst 1      (b)
            Inst 2      (c)
               .
               .
               .
L1:         Inst 3      (d)
            Inst 4      (e)
```

(a)－(b)－(d)－(e)
の順に実行される

図3-7　ジャンプ命令の実行

には一般に多くの処理クロック数が要求されるため、これらを分離した。スカラ浮動小数点命令ではALU演算命令と異なり、直後の命令では演算結果を使用することはできず、演算結果がパイプラインから出力されるまで待たねばならない。この遅延処理はコンパイラにより適切にスケジューリングされる。

3.4.3 レジスタ構成

ジェットパイプライン・アーキテクチャのレジスタはスカラレジスタとベクトルレジスタから構成される。また、条件文のベクトル化をサポートするためのベクトルマスクレジスタ(VMR)も用意し、ベクトル命令で利用することが可能である。

スカラレジスタにはALUや演算パイプラインなどから同時並列にアクセスが集中することが予想されるにもかかわらず、初期の構成ではすべての命令パイプラインから共有されることを仮定していた[Katahira 92,94a]。この場合、高速にアクセス可能な自由度の高いスイッチ構造はたとえばクロスバがあげられるが、クロスバスイッチを用いてこの構成をとった場合、レジスタ数を１２８個、readとwriteのポートをそれぞれ8個と仮定するとスイッチの総数は以下の式で見積もることができる。

32(bits) × 128(registers) × 16(R/W ports) = 65536

これは明らかに現実的とは言えない。そのため、アクセスの自由度をできるだけ保持したままハードウエア量を減少させることを目的として、現在の構成では図3-8に示すような隣接する命令パイプライン間で一部オーバラップアクセス可能なバンク分けされたレジスタを採用している[Katahira 94b]。この構成では、スカラレジスタは全部で9つのバンクに分割され、それらは全体からアクセス可能なグローバルレジスタ、各命令パイプラインのみからアクセスされるローカルレジスタ、および隣接した2つの命令パイプラインからアクセス可能なオーバラップレジスタの3種類に分けられている。なお、オーバラップレジスタのうち現IPからIP番号の大きな方と共有しているものをフォワードレジスタ、逆にIP番号の小さな方と共有しているものをバックワードレジスタと呼んでいる。ただし、最大のIP番号を持つIPのフォワードレジスタは最小のIP番号を持つIPのバックワードレジスタに対応している。このようなレジスタファイルに対して図3-9に示すような構成をとった場合、ク

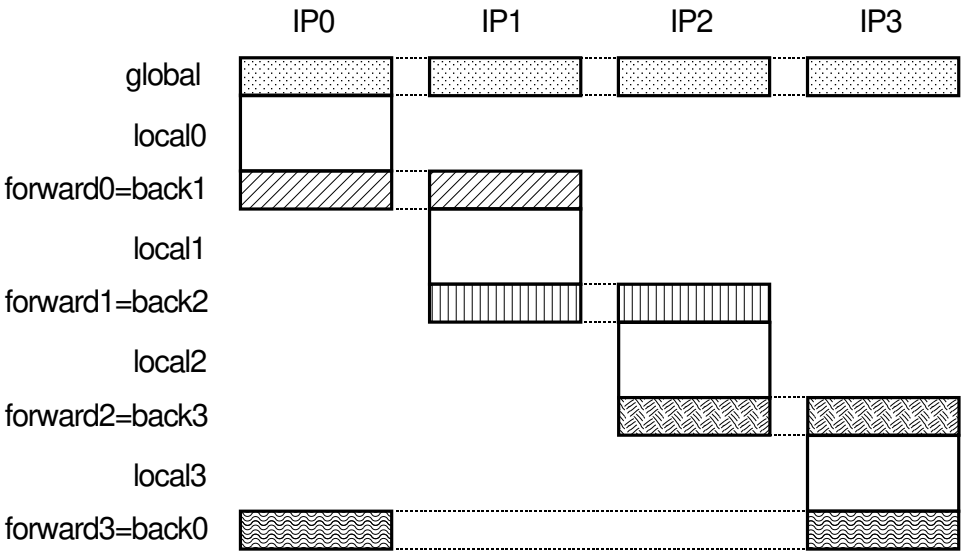


図3-8　バンク化オーバラップレジスタの概念図

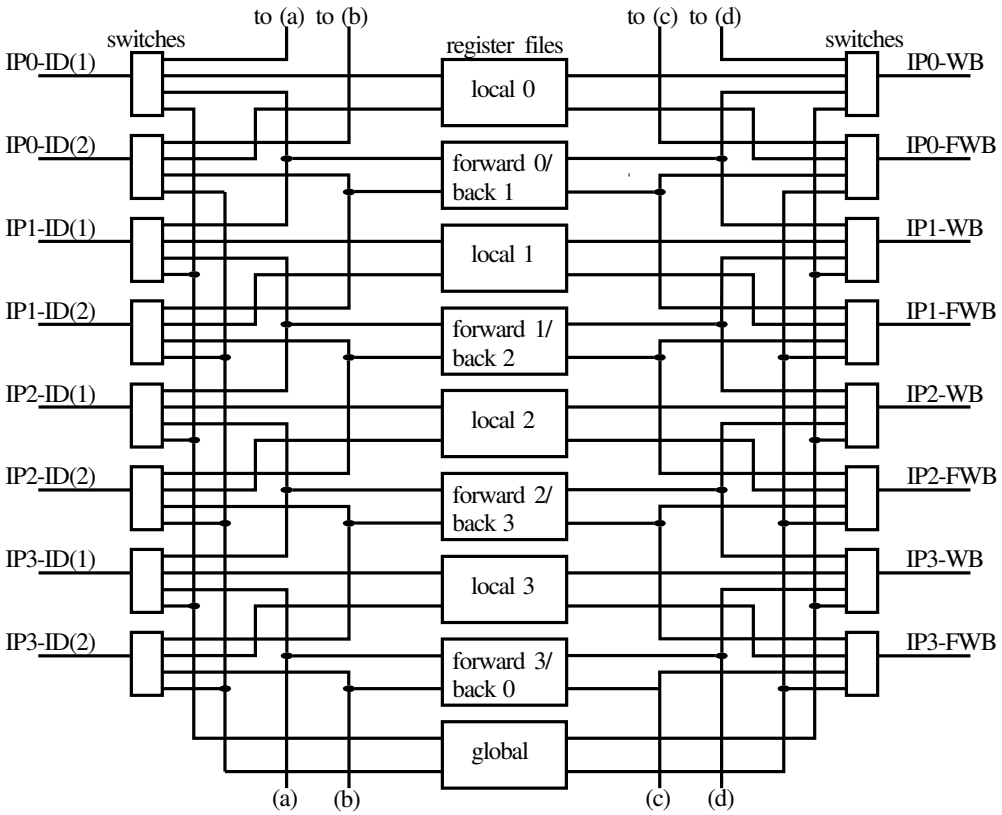


図3-9　バンク化オーバラップレジスタの構成例

ロスバの場合と同様にレジスタ数を128個、readとwriteのポートをそれぞれ8個と仮定し、一般に用いられているreadとwriteのポートをそれぞれ2個もつレジスタファイルを各バンクに使用するとした場合、スイッチの総数は以下の式で見積もることができる。

32(bits) × 4(one-to-four selectors) × 16(R/W ports) = 2048

クロスバの場合と比較して1/32ですみ、ハードウエア量を大きく減少させることが可能である。この場合、隣接したIP間ではオーバラップレジスタを介してデータが共有でき、かつ隣接していない場合でもグローバルレジスタにより共有するか、あるいは中間のIP一つを経由することにより参照可能で、レジスタアクセスの柔軟性の低下を抑えることができる。また、このようなレジスタ構成では、命令スケジューリングにおいて、ソフトウエアパイプライン手法を適用する際に、イタレーション間のデータの移動と各イタレーション内のローカル変数の分離を行う上でも有用であるが、この詳細は次節で述べられる。

現在のジェットパイプライン・アーキテクチャのスカラレジスタ構成を図3-10に示す。なお、図中のSRnはレジスタ指定フィールドで用いられるスカラレジスタ番号である。スカラ命令中のレジスタ指定フィールドは8ビットであるので、256個のレジスタを指定できるが、そのうち前半の128個をデータレジスタとし、バンク化オーバラップレジスタとしている。全IPからアクセス可能なグローバルレジスタは%G0 ~ %G7までの8個、各IPに固有のローカルレジスタは%L0 ~ %L13までの14個、そしてIP間で共有されているフォワードとバックワードレジスタはそれぞれ%B0 ~ %B7, %F0 ~ %F7までの8個である。スカラデータレジスタにおけるレジスタ通し番号と物理レジスタバンクとの対応を、図3-11に示す。

後半の128個のレジスタには各種のスペシャルレジスタが割り当てられている。これらのスペシャルレジスタは各IPから共通にアクセス可能で、またこれらのレジスタは実際に値が代入されることにより有効になるものが多いが、その際にはフォワーディングは働かないので注意する必要がある。それぞれのスペシャルレジスタの働きを以下に示す。

ZR(SR0,%G0): ゼロレジスタ。読み出し時は常にゼロが読み出され、書き込みは無視される。各種の命令のオペランドに指定して、合成

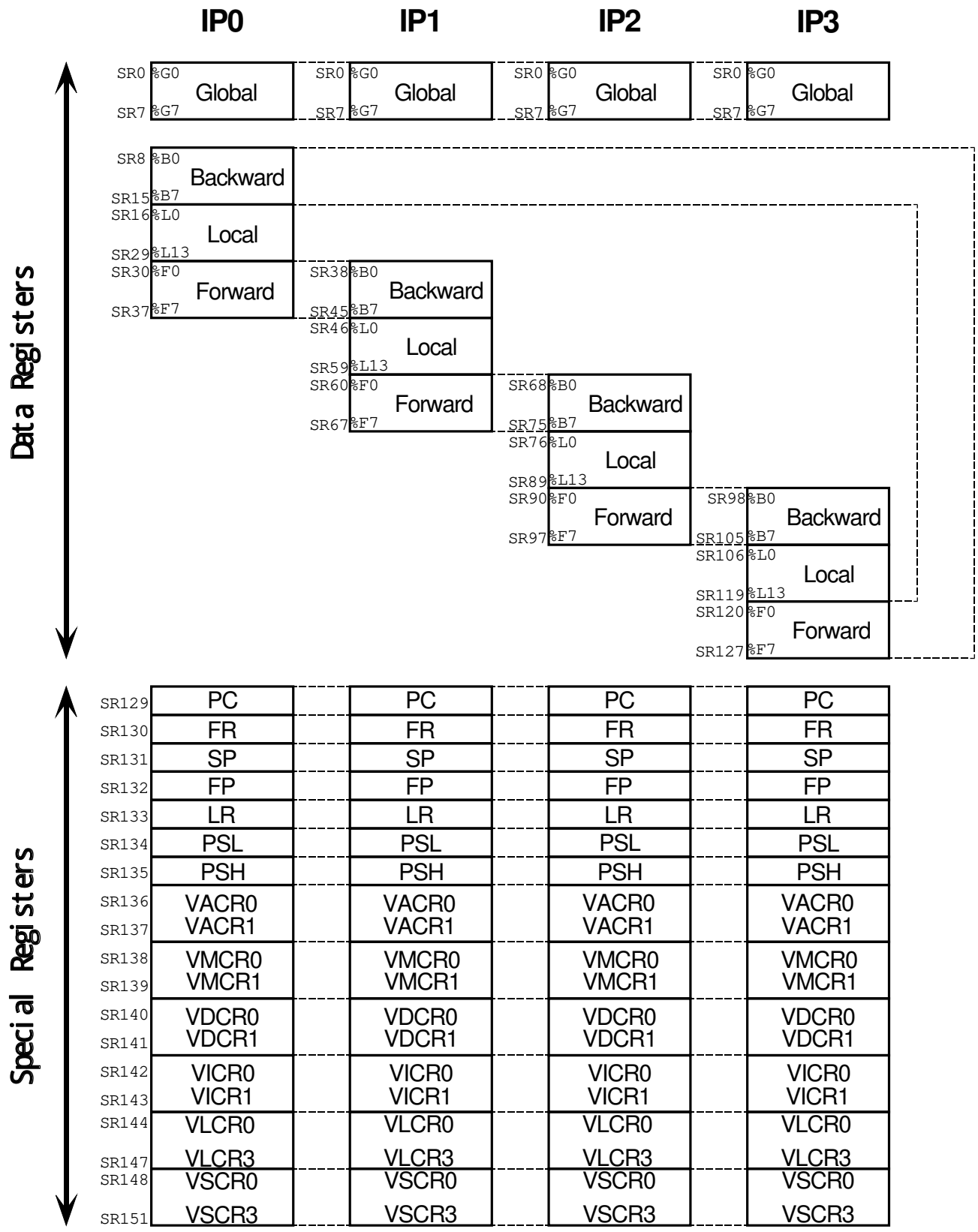


図3-10　ジェットパイプラインアーキテクチャスカラレジスタ

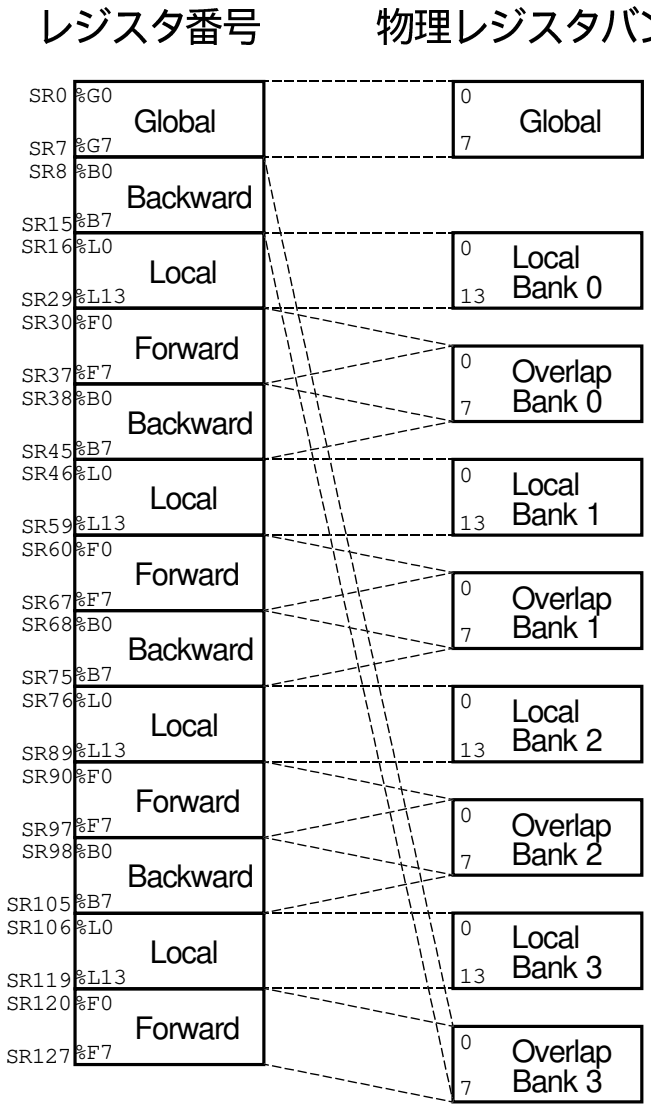


図3-11　スカラデータレジスタ番号と物理レジスタバンクの対応

命令を作るのに用いられることがある。

PC(SR129): プログラムカウンタ。現在フェッチしている命令を指す。

FR(SR130): フラグレジスタ。スカラ演算命令の演算結果により、ゼロフラグとサインフラグがセットされる。なお、各命令パイプラインのフラグは独立している。条件分岐命令はこのフラグレジスタの内容を参照して分岐するか否かを決定する。

SP(SR131): スタックポインタ。

FP(SR132): フレームポインタ。

これらのSP,FPは関数呼び出しの際に使用し、局所変数領域の確保や参照のためのベースレジスタとして用いられる。

LR(SR133): リンクレジスタ。ジャンプアンドリンク命令において、戻り番地が格納される。

PSL(SR134): プログラムセグメント下限レジスタ。

PSH(SR135): プログラムセグメント上限レジスタ。

これらのレジスタに値を設定することにより、PSL以上PSH未満のアドレスに対する書き込みを禁止することができる。プログラムセグメントへの書き込みはセグメンテーションフォールトを発生する。

VACR0(SR136): VPU0用ベクトル浮動小数点加減算命令ベクトル長設定・カウンタレジスタ。

VACR1(SR137): VPU1用ベクトル浮動小数点加減算命令ベクトル長設定・カウンタレジスタ。

VMCR0(SR138): VPU0用ベクトル浮動小数点乗算命令ベクトル長設定・カウンタレジスタ。

VMCR1(SR139): VPU1用ベクトル浮動小数点乗算命令ベクトル長設定・カウンタレジスタ。

VDCR0(SR140): VPU0用ベクトル浮動小数点除算命令ベクトル長設定・カウンタレジスタ。

VDCR1(SR141): VPU1用ベクトル浮動小数点除算命令ベクトル長設定・カウンタレジスタ。

VICR0(SR142): VPU0用ベクトル整数・論理演算命令ベクトル長設定・カウンタレジスタ。

VICR1(SR143): VPU1用ベクトル整数・論理演算命令ベクトル長設定・カウンタレジスタ。

VLCR0(SR144): IP0用ベクトルロード命令ベクトル長設定・カウンタレジスタ。

VLCR1(SR145): IP1用ベクトルロード命令ベクトル長設定・カウンタレジスタ。

VLCR2(SR146): IP2用ベクトルロード命令ベクトル長設定・カウンタレジスタ。

VLCR3(SR147): IP3用ベクトルロード命令ベクトル長設定・カウンタレジスタ。

VSCR0(SR148): IP0用ベクトルストア命令ベクトル長設定・カウンタレジスタ。

VSCR1(SR149): IP1用ベクトルストア命令ベクトル長設定・カウンタレジスタ。

VSCR2(SR150): IP2用ベクトルストア命令ベクトル長設定・カウンタレジスタ。

VSCR3(SR151): IP3用ベクトルストア命令ベクトル長設定・カウンタレジスタ。

それぞれのベクトル命令の実行前にこれらのレジスタに値(1 ~ 128)を設定することにより、ベクトル演算長を設定することができる。また、実行中にこれらのレジスタを参照することにより、ベクトル演算の進行状況を確認できる（0になると実行終了）。なお、0を設定するとデフォルト値の128とみなされる。正しくない値を設定すると実行時エラーとなる。

ベクトルレジスタはVR0 ~ VR63までの64個を指定可能である。各ベクトルレジスタは128個の要素を含んでいる。なお、ベクトルレジスタの各要素に対して、どの程度まで同時にアクセス可能とするかなどの、ベクトルレジスタのハードウエアの詳細は、実現可能なハードウエア量等を考慮して現在検討中である。また、ベクト

ルマスクレジスタはVMR1 ~ VMR7の7個が使用可能であり、それぞれベクトルレジスタ長と同じ128ビットの要素を持っている。ベクトルマスクレジスタはベクトルマスクレジスタ設定命令によりベクトルレジスタの内容に応じて真または偽の値が設定される。ベクトルマスクレジスタを参照するベクトル命令では、ベクトルマスクレジスタが真の要素のみ演算が有効となる。なお、ベクトルマスクレジスタ同士の論理演算も可能であり、その場合にはVMR0をすべての要素が真であるものとして用いることができる。

3.4.4 ロード・ストアユニットおよびメモリシステム

レジスタ-主記憶間のデータ転送のためには、スカラロード・ストア命令用のロード・ストアユニット(LDU/STU)及びベクトルロードストア命令用のベクトルロード・ストアパイプライン(LDP/STP)の両方がそれぞれ命令パイプライン数だけ存在し、ベクトル演算等と並行してデータ転送が行えるようにしている。VPUと同様に、ベクトルロード・ストアパイプラインもIUによって起動された後は独立して動作するようになっている。スカラロード命令はメモリ内のデータを読み込むのがIPのMEMステージであるため、浮動小数点演算命令と同様に直後の命令ではロードデータを使用することはできない。

また、スカラスペシャルレジスタPSL,PSHを用いてプログラムセグメントを保護することができる。すなわち、図3-12に示すように、PSL以上PSH未満のアドレスに対するストア、ベクトルストアは禁止され、もしストアを行った場合にはセグメンテーションフォールトが生じる。

メモリシステムは複数のロード・ストアユニットから並列にアクセスされる可能性があるため、インターリーブ方式を採用するなどして高性能化する。また、命令フェッチの際のアクセス遅延を減少させるために、命令キャッシュ(Inst. Cache)を設けることにする。なお、これらのメモリ周辺のハードウエア構成は、理想化されている部分が残されているため、今後の検討が必要である。

現時点でのハードウエア構成は以上に示したとおりであるが、より現実化した構成をめざして、現在も引き続き検討・変更を行っている。

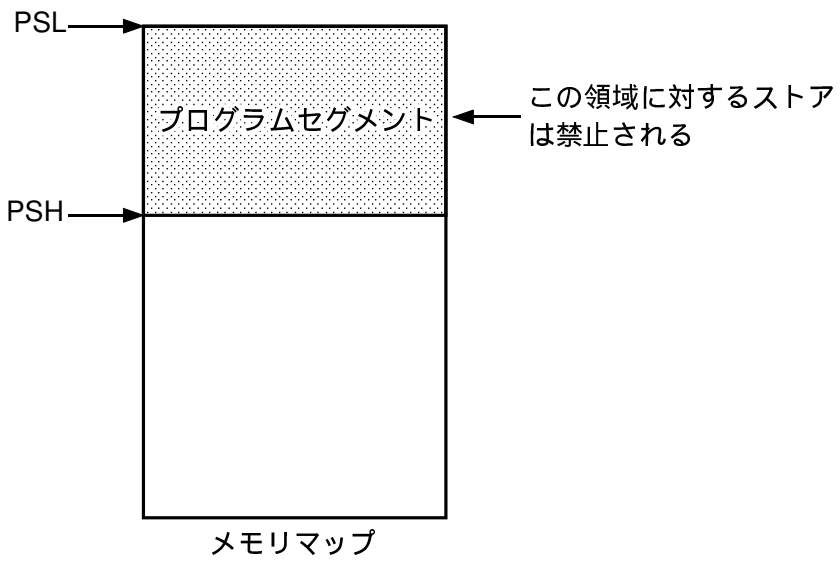


図3-12　PSLとPSHによるプログラムセグメントの保護

§ 3.5　ソフトウエア環境

3.5.1　命令セット

ジェットパイプライン・アーキテクチャの命令セットを以下に述べる。図3-13に典型的な命令のフォーマットを、また表3-2に命令の分類表を示した。

スカラ命令セットはRISCを基本にした32bit固定長形式で、スタンフォードMIPSアーキテクチャの命令セット[Gross 83]を参考にした。具体的には、ロード・ストア命令とレジスタ間演算命令（整数および浮動小数点）、そして分岐命令から構成されている。分岐命令は、一般的な条件コードによる分岐に加えて、MIPSで用いられているコンペアアンドジャンプを用意し、適当な方式を選択できるようにした。また、リンクレジスタに戻り番地を保存した後で分岐するジャンプアンドリンク命令により、プロシージャコールをサポートしている。

ベクトル命令セットも同様に32bit固定長で、一般的なベクトルスーパーコンピュータ[Yasumura 85][Jippo 87]と同様に、ベクトルロード・ストア命令、ベクトルレジスタ間の演算命令（整数演算及び浮動小数点演算）、その他のベクトル命令(マスクレジスタ操作など）から構成される。また、チェイニング演算も命令セットレベルで制御される。

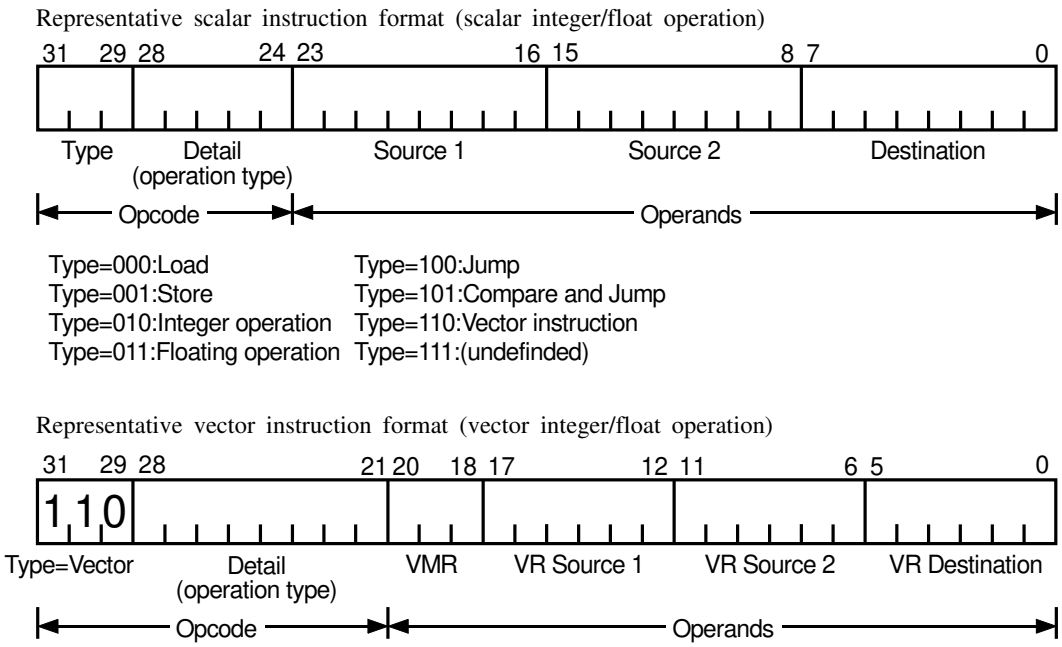


図3-13　命令フォーマットの例

表3-2　ジェットパイプラインの命令

Instruction Type		**Pipe-stage**	**Description**
All instructions		IF	fetch instruction from memory
		ID	decode instruction / source register fetch
Scalar	integer operation	ALU	ALU operation
		MEM	no operation
		WB	write back to distination register
		FWB	no operation
	floating operation	ALU	send source value to floating pipeline
		MEM	no operation
		WB	floating(A/S) pipeline -> dest reg (add/sub)
		FWB	floating(M/D) pipeline -> dest reg (mult/div)
	load/store	ALU	ALU operation (address calculation)
		MEM	memory access
		WB	write back to distination register
		FWB	no operation
	jump/ compare&jump	ID	send jump displacement to address ALU
		ALU	ALU operation (compare & jump)
		MEM	no operation
		WB	write back to distination register (jump&link)
		FWB	no operation
Vector	vector load/store	ALU	parameters(addr etc.)->vector Load/Store Unit initiate VLU/VSU
	vector int/logic operation	ALU	parameters->vector int/logic pipe controller initiate vector int/logic pipe controller
	vector floating operation	ALU	parameters->vector floating pipe controller initiate vector floating pipe controller

3.5.2　基本的命令スケジューリング手法

本システムでは、以下に示すようなパイプライン方式の演算器などのハードウエアに対する命令スケジューリングの全てを、VLIWのようにコンパイラが静的に行う。従ってVLIWの場合と同様に、コンパイラにはハードウエア構成に関する情報が要求される。ジェットパイプライン・アーキテクチャにおいて、コンパイラが命令スケジューリングをする際に考慮する事項を以下に示す。

・先行するスカラ浮動小数点演算命令の結果を利用する命令が存在する場合、パイプラインから結果が出力されるまでの間に、他の順序入れ替え可能な命令あるいはNOP命令を挿入する。

・一般的なRISCシステムと同様に本アーキテクチャでも遅延分岐を採用しているため、分岐命令後のディレイスロットへの命令の挿入を行う。

命令レベルでの並列処理のために、コンパイラがオペランド間の依存関係を検出し、並列動作可能な複数の命令（最大４つ）を同一サイクルで実行されるように４本の命令パイプラインにマッピングする。

ただし、§3.4で述べたようなハードウエア構成上の制限により、同時に実行できる命令には図3-14に示すような制限が存在する。さらに、数値演算プログラム等における単純なループ構造のようなプログラムの場合で、ベクトル命令が使用できるようなときには、スーパーコンピュータのようにベクトル化を行い、ベクトル命令を使用してより高速化することができる。さらに、並列化とベクトル化を組み合わせて、複数のベクトル命令を並列動作させたりチェイニングを行うようなスケジューリングをすることも可能である。

以下にいくつかの例を用いてジェットパイプラインにおける命令スケジューリングの例を示す。

1)プロシージャコール・リターン

ジェットパイプラインにおけるプロシージャコールとリターンは、図3-15に示すようなコードにより実行される。その際のメモリマップを図3-16に示す。コード列とメモリマップを参照すればわかるように、まず現プロシージャの%fpと%lrをスタックに保存し、%fpをその先の番地に設定した後にジャンプアンドリンク命令を使用してプロシージャに分岐する。呼ばれたプロシージャでは%fpの指す番地からロ

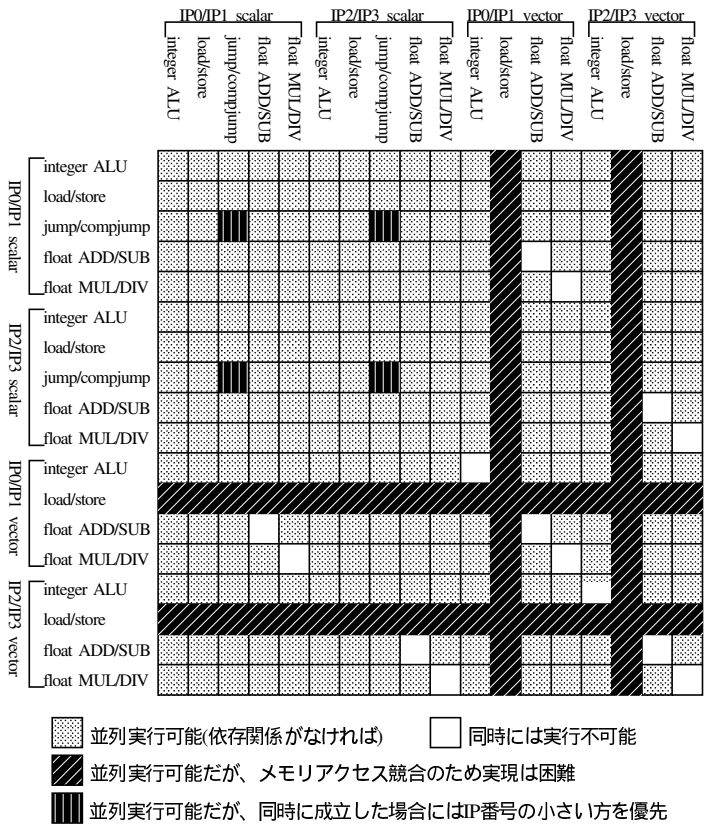


図3-14　並列に実行可能な命令の組み合わせ

```
        stb   %sp,-4,%fp   ;フレームポインタの保存
        stb   %sp,-8,%lr   ;リンクレジスタの保存
        addi  -8,%sp,%fp   ;フレームポインタの更新
        jmpl   proc        ;ジャンプアンドリンク(戻り番地は%lr)
        nop                ;ディレイスロット
```

(a) 呼び出し側のコード列(プロシージャコール)

```
proc:   addi lval,%fp,%sp   ;ローカル変数領域(lval分)の確保

                (プロシージャ本体のコード)

        addi %fp,8,%sp   ;スタックポインタの復帰
        ldb  %fp,4,%fp   ;フレームポインタの復帰
        ldb  %fp,0,%lr   ;リンクレジスタの復帰
        jb   %lr,0       ;%lrの指す番地にジャンプ
        nop              ;ディレイスロット
```

(b) 呼び出された側のコード列(ローカル変数の確保とリターン)

図3-15　ジェットパイプラインにおけるプロシージャコール・リターン

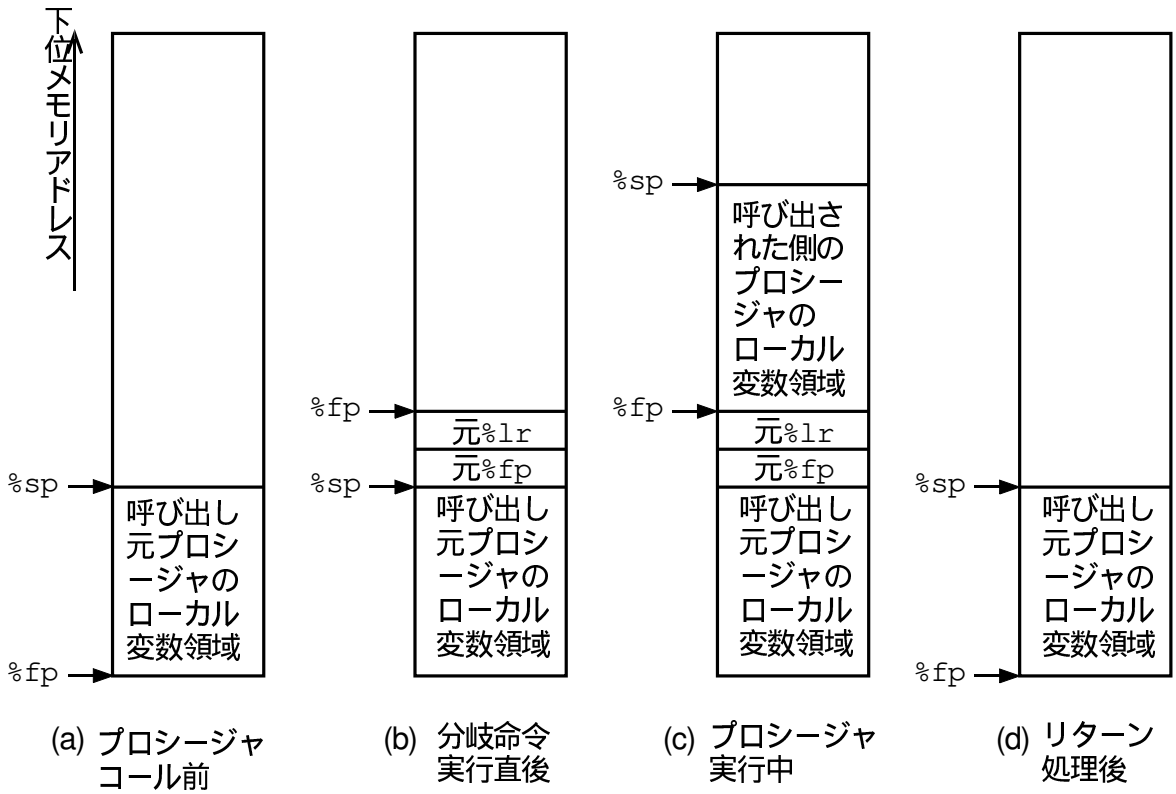


図3-16　ジェットパイプラインにおけるプロシージャコール・リターンのメモリマップ

ーカル変数領域を確保し、%spを更新する。リターン時には%fp,%sp,%lrをスタックから元の値に復帰しつつ、ジャンプアンドリンク命令で%lrに保存されていた番地に復帰する。%lrをロードする命令が復帰用のジャンプ命令より先に配置されているが、メモリからの読み出しはパイプラインの後方のステージにあるため、%lrをロードする命令の次の番地にある%lrの内容へのジャンプ命令の分岐実行時点ではまだ%lrに代入されないため、正しく動作する。%fpのロードと%lrのロードの命令の関係も同様である。

これらのコード列から同時に実行可能な命令を同時に実行するようにIPにマッピングし、またジャンプ命令後のディレイスロットを有効利用するように命令を移動した結果のコードを図3-17に示す。並列化とディレイスロットの利用により、実行に要するステップ数がコール、リターンともに5ステップから2ステップまで減少することがわかる。

2)ループアンローリングを用いた基本的な並列化

ループ構造はプログラム中によく出現するが、ループの各繰り返し（イタレーション）間にデータの依存関係がない場合、ループの各イタレーションは独立であるから並列に実行しても問題はない。このようなループは並列性を非常に多く含んでいると言うことができる。ジェットパイプライン・アーキテクチャ上でそのようなループプログラムに対して基本的な並列化を行った例を図3-18に示す。この図では、オリジナルのループでは1回ごとに繰り返していた各イタレーションを4回アンローリングすることにより4イタレーションごとにループするように変更し、さらにアンローリングした各イタレーションをジェットパイプラインの4本のIPにマッピングして実行している。この場合、それぞれのイタレーションは独立しているため、各IPにおけるレジスタ割り当てはローカルレジスタのみを使用すればよい。また、共通して使用できる値、たとえば配列のベースアドレスなどはグローバルレジスタに割り当てて共用することもできる。

3)基本的なベクトル化

2)で述べたような依存のないループ構造は、ベクトル命令を用いて高速に処理することも可能である。ベクトル命令を用いずに、スカラ命令のみでループを構成す

```
IP0                IP1                IP2                IP3
jmpl  proc         stb  %sp,-4,%fp    stb  %sp,-8,%lr   nop
addi -8,%sp,%fp    nop                nop                nop    ;ディレイスロット
```

(a) スケジューリング後の呼び出し側のコード列(プロシージャコール)

```
IP0              IP1                IP2              IP3
jb  %lr,0        addi %fp,8,%sp     ldb  %fp,0,%lr   ldb  %fp,4,%fp
nop              nop                nop              nop          ;ディレイスロット
```

(b) スケジューリング後のリターン処理コード列

図3-17　ジェットパイプラインにおけるプロシージャコール・リターン
（スケジューリング後）

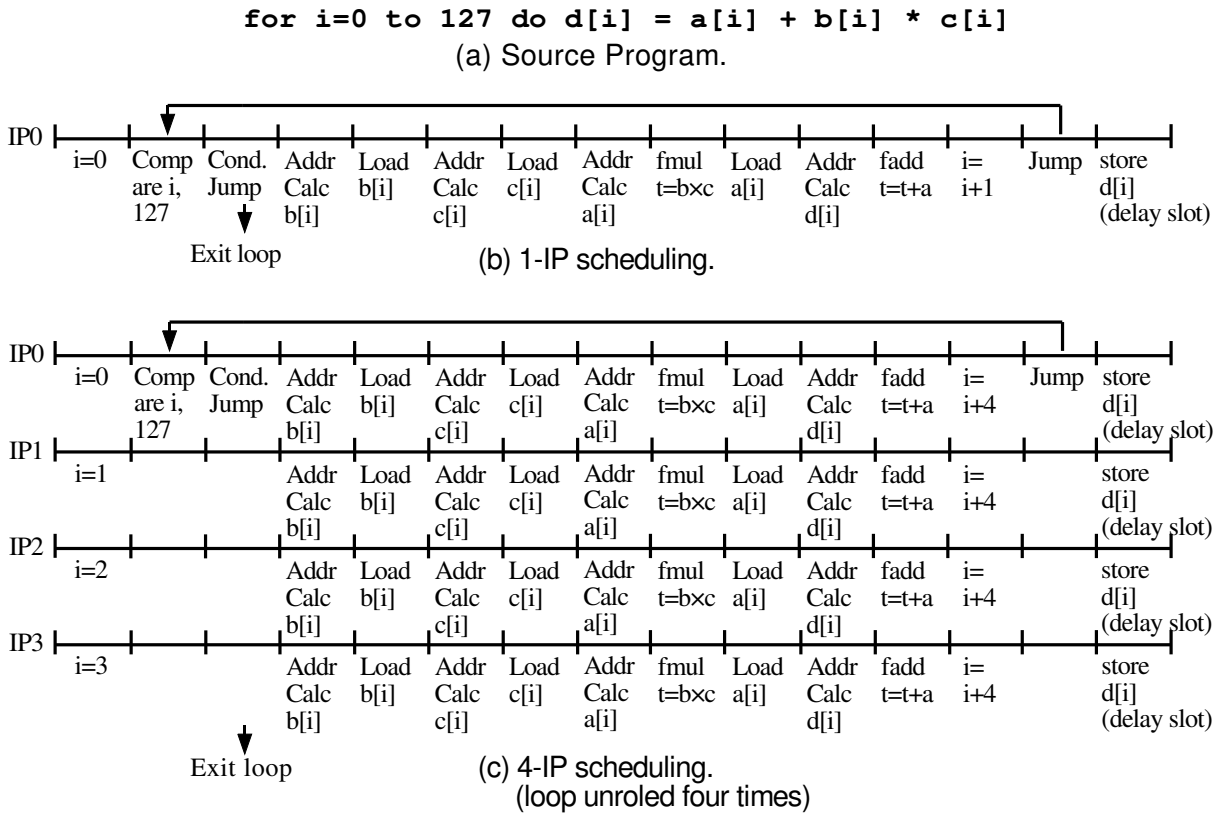


図3-18　ジェットパイプラインのスケジューリング例（スカラ）
（アンローリングによる並列化）

るためには演算命令の他にループカウンタの増加とループ回数との比較命令、および条件分岐が必要となり、たとえば浮動小数点演算を含むループの場合浮動小数点パイプラインに連続してデータを投入することは不可能で、どうしてもパイプラインのセグメントに空きが生じてしまう。一方、ベクトル命令を使用した場合には、ベクトル長(ジェットパイプライン・アーキテクチャの場合には128)までのループはループを構成するための命令が不要である。また、浮動小数点演算を行う場合にはベクトル命令を使用すれば浮動小数点演算パイプラインに連続してデータを投入可能であり無駄な空きが生じない。従って、ベクトル化できるようなループ構造であればベクトル化した方が高速化が可能である。2)で並列化したプログラムをベクトル命令を用いてベクトル化した例を図3-19に示す。図中の(wait)はベクトル命令の実行修了待ちを意味しているが、これは、図3-20に示すような、レジスタ構成の部分で述べたスペシャルレジスタ内のベクトル長設定・カウンタレジスタの値を調べるスカラ命令のループを用いて実現可能である。なお、このループの例の場合、チェイニングなどを用いて同時に実行可能な複数のベクトル命令の実行を並行して行うことにより、さらに高速化可能である。

また、ループ構造の中には依存関係があるため直接ベクトル化できないものがあるが、そのような場合でもオリジナルのループ構造をプログラムの意味が変わらないように変形し、依存がある部分とない部分に分割してベクトル化できる部分だけでもベクトル化することにより、高速化することができる。このような例には図3-21(a)に示すような内積演算のプログラムが挙げられるが、この場合は図3-21(b)のように積を計算する部分と合計を計算する部分とに分割し、ベクトル化できる積の計算部分のみをベクトル化し、残りの合計計算の部分をスカラ処理により実行する方法をとることによって高速化が可能である。

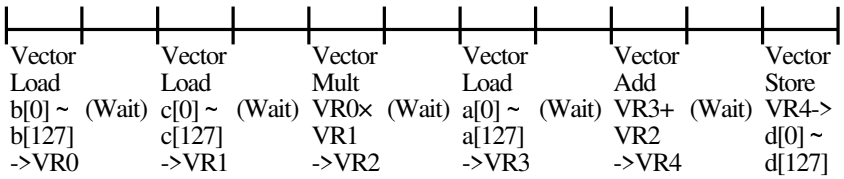
また、ジェットパイプライン・アーキテクチャではベクトル演算とスカラ命令を同時に実行可能な場合があり、ベクトル・スカラ処理を併用する場合は可能であれば並行動作させることによってより高速化することが可能となる。

3.5.3　ディスパッチスタックを用いた並列化手法

前節で述べた基本的な並列化手法は比較的単純なプログラム構造に対して並列化を行った例であるが、実際のプログラムはもっと複雑な場合があり、またそのよう

```
for i=0 to 127 do d[i] = a[i] + b[i] * c[i];
```

(a) Source Program



(b) Vectorized code

図3-19　ジェットパイプラインにおける基本的なベクトル化の例

```
        LDI   100,%VACR0    ;ベクトル長の設定
        VADD  VR0,VR1,VR2   ;ベクトル加算命令
L1:     CMPI  0,%VACR0      ;VACR0が0になるまでループ
        JC    NZ,L1
        NOP                 ;delay slot
```

図3-20　ベクトル長の設定とベクトル処理終了判定の例

```
for i=0 to 127 do s = s + a[i] * b[i];
```

(a) ソースプログラム

```
for i=0 to 127 do t[i] = a[i] * b[i];
for i=0 to 127 do s = s + t[i];
```

(b) 変形したソースプログラム

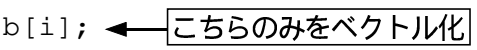


図3-21　内積演算のループ構造変形によるベクトル化

な場合でも並列化を自動的に行えるようにしなければ実用化は不可能である。本節以降では、一般的なプログラムから並列化可能な部分を抽出しスケジューリングするための手法について述べる。

ジェットパイプラインの4本の命令パイプラインを有効に利用し、命令レベルの並列処理を行わせるためのコンパイラには、逐次的なプログラムから自動的に並列性を抽出し並列動作可能な命令を複数の命令パイプラインに割り付けることが要求される。逐次的な命令流から並列性を取り出す方法にはTomasuloのアルゴリズム[Tomasulo 67]などの多くの手法があるが、本節ではそのような方法の一つであるディスパッチスタック法[Acosta 86]について述べる。

ディスパッチスタック法では、逐次的な命令列の各命令のオペランドを比較し、それらが互いに依存関係があるかどうかを検出する。具体的には、先行する命令のディスティネーションレジスタが現命令のソースレジスタと一致するか、および先行する命令のディスティネーションレジスタが現命令のディスティネーションレジスタと一致するか、といった項目が検査され、その結果依存関係がないと判断された命令が同時に発行可能であることになる。ディスパッチスタックの例を図3-22に示す。

ジェットパイプラインの場合、RISCと同様に演算命令はレジスタ間演算に限定されているため、ディスパッチスタックのアルゴリズムを実現する際にはレジスタ番号のみを比較すればよい。また、ジェットパイプラインの最大並列実行可能命令数は4であるため、同時に発行可能な命令が4以上存在する場合には4個ずつディスパッチスタックから取り出されて命令パイプラインに配置される。ただしこのとき、ジェットパイプラインのバンク分けオーバラップレジスタによる命令配置の制限、すなわち命令パイプライン間の距離が2のIPでは共有されているレジスタがないということを考慮しなければならない。また、ディスパッチスタックの欠点として、ハードウエア量が大きくなるという問題点が指摘されている[Johnson 91]が、ジェットパイプラインに今回応用した場合においてはソフトウエア（コンパイラ）で実行するため問題にはならない。

3.5.4　ソフトウエアパイプライニングを用いた並列化手法

ソフトウエアパイプライニング[Lam 88]はハードウエアにおけるパイプライン処

先頭

Tag	OP	S_1	$\alpha(S_1)$	S_2	$\alpha(S_2)$	D	$\alpha(D)$	$\beta(D)$	I^2
I0	AD	R0	0	R1	0	R0	0	0	0
I1	AD	R2	0	R3	0	R2	0	0	0
I2	AD	R0	1	R2	1	R0	1	1	4
I3	AD	R4	0	R5	0	R4	0	0	0
I4	AD	R6	0	R7	0	R6	0	0	0
I5	AD	R4	1	R6	1	R4	1	1	4
I6	AD	R0	2	R4	2	R0	2	2	8

α(R) = レジスタRが先行する未実行の命令でディスティネーションに指定されている回数　例) □←→□ の関係

β(R) = レジスタRが先行する未実行の命令でソースに指定されている回数　例) □←→□ の関係

$I^2 = \alpha(S_1) + \alpha(S_2) + \alpha(D) + \beta(D)$

この値が0の命令を同時に実行できる

例）　上記の場合、I0,I1,I3,I4を同時に実行可能

図3-22　ディスパッチスタックの例

理をループプログラムに応用したもので、ループの立ち上がり（プロローグ）と最後の部分（エピローグ）を除いた定常状態の部分を、ループのイタレーションの数回分を重ね合わせて処理することにより、並列度を増加させ並列化を容易にする方法である。ループ本体を4つのブロックに分けてソフトウエアパイプラインを適用した場合の例を図3-23に示す[Saitoh 94]。

通常の計算機においてソフトウエアパイプライニングを適用する場合、同時にループの異なるイタレーション部分が並列実行されるため、それらの間で変数等が干渉しないようにレジスタのリネーミング等が必要になるが、ジェットパイプラインの場合はそれぞれを各命令パイプラインに配置し、バンク分けされたレジスタを使用することによって容易にソフトウエアパイプライニングを実現することが可能となる[Sasaki 94][Katahira 94b]。この場合、IP間で共有されているオーバラップレジスタを用いて各繰り返し間の変数の受け渡しも容易にできる。

具体的なソフトウエアパイプライニングの例を、以下に示すループプログラムを用いて示す。

```
for k := 1 to n do
   begin
      x[k] := q + y[k] * (r * zx[k+10] + t * zx[k+11]);
   end
```

このプログラムは、以下のようなアセンブラプログラムに変換される。ただし、すべてのレジスタはローカルレジスタ(%L)のみを使用している。

```
 1: L5:ld     %L9,%L1,%L2
 2:    mul    %L5,%L2,%L2
 3:    ld     %L0,%L1,%L3
 4:    mul    %L4,%L3,%L3
 5:    add    %L2,%L3,%L2
 6:    ld     %L4,%L1,%L3
 7:    mul    %L2,%L3,%L2
 8:    add    %L6,%L2,%L2
 9:    st     %L7,%L1,%L2
10:    addi   %L2,1,%L2
11:    cmp    %L2,%L5
12:    jc     LE,L5
13:    addi   %L1,4,%L1    ;delay slot of jc
```

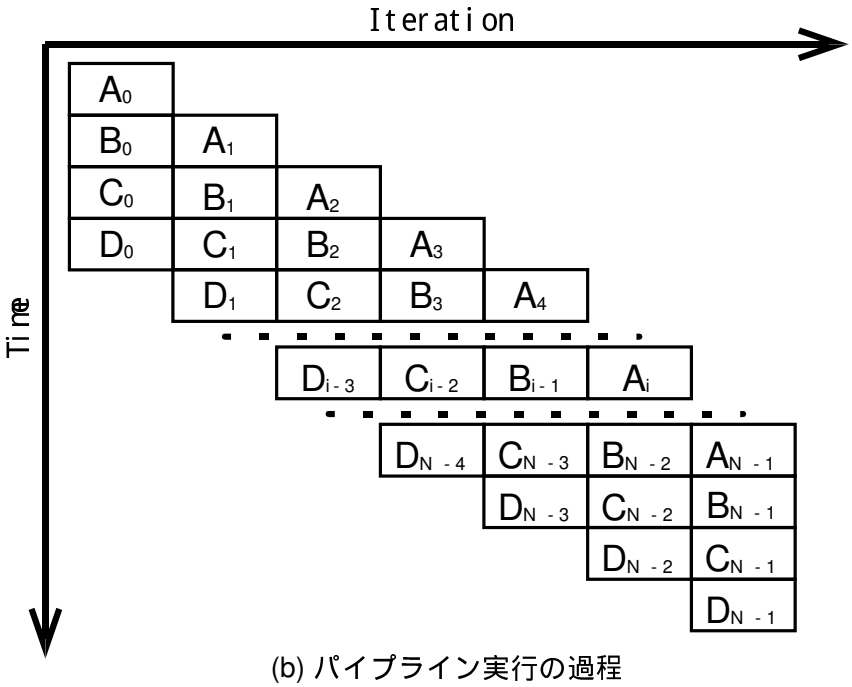
実際のソフトウエアパイプライニングは以下の4ステップに分けて行われる。以下で順を追って説明する。

(1)最内ループボディの抽出（これは上記の例ではすでに済んでいる）

(2)イニシエーション・インターバル(initiation interval,II)の計算

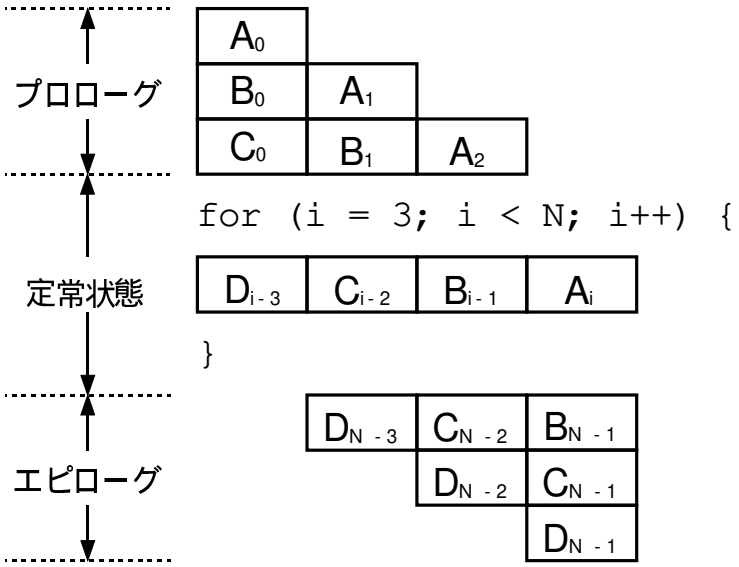
```
for (i = 0; i < N; i++) {
    A
    B
    C
    D
}
```

(a) ループプログラム



(b) パイプライン実行の過程

図3-23　ソフトウエアパイプライニング



(c) ソフトウエアパイプライン化されたループ

図3-23　ソフトウエアパイプライニング (続き)

(3)定常状態のアンローリング

(4)レジスタ番号の付け替え（リネーミング）

ループボディを抽出した後、イニシエーション・インターバル(II)を求める必要がある。IIはリソース制約と先行制約により決定される。ジェットパイプラインの場合、リソース制約は同時に実行可能な命令数より4となる。一方、先行制約は、ループのイタレーション間の依存関係により生じる。先行制約が存在する場合、先行するループのイタレーションにおける結果が得られない限り次のループのイタレーションを開始できない。ただし、例示したプログラムにおいてはループのイタレーション間に依存関係は存在しないため、先行制約はない。従って、IIはこの両制約とループ中の命令数より、次式で求められる。

II = [命令数 / max (リソース制約, 先行制約)]
= [13 / 4]
= [3.25]
= 4

なお、ここで[]は天井関数(ceiling function)を意味する。

ソフトウエアパイプライン化されたループの定常状態においては、もとのプログラムのループの複数のイタレーションが同時に並行して実行されている。従って、元のプログラムでは同一の名前を持つ変数に対して、異なるレジスタを割り当てなければ衝突が生じ、正しい答えを得ることはできない。これを解決するために、ループ本体をアンローリングする必要がある。アンローリング回数は、ループ中でもっとも長いライフタイムを持つレジスタのライフタイムとIIから、次式で求められる。例示したプログラム中でもっとも長いライフタイムを持つレジスタは%L2で、9命令サイクルである。

アンローリング回数 = [max (各レジスタのライフタイム) / II]
= [9 / 4]
= [2.25]
= 3

以上で求められたIIとアンローリング回数を用いて、アセンブリコードを書き直

すと以下のようになる。(ただし、オペコードのみ示してある。)

	IP1	IP2	IP3	IP4
1	addi	st	add	ld
2	nop	addi	ld	mul
3	nop	cmp	mul	ld
4	nop	jc	add	mul
5	addi	st	add	ld
6	nop	addi	ld	mul
7	nop	cmp	mul	ld
8	nop	jc	add	mul
9	addi	st	add	ld
10	nop	addi	ld	mul
11	nop	cmp	mul	ld
12	nop	jc	add	mul

4行ごとのブロック(1~4, 5~8, 9~12)がループの定常状態を示しており、全体として3回アンローリングされている。また、各行のそれぞれの列が対応するIPを表している。

最後に、ジェットパイプラインのバンク化レジスタに対応したレジスタ名の付け替えを行い、ソフトウエアパイプライニングが完了する。これは以下のような簡単な規則により行うことができる。

まず、各ループで独立な変数には、ローカルレジスタ(%L)を割り当てる。これにより、ループの繰り返し間における干渉を考える必要が無くなる。また、各繰り返し間で共通な変数(たとえば配列のベースアドレスなど)はグローバルレジスタ(%G)に割り当てられる。さらに、ループの繰り返し間で、前回の演算の結果を受ける場合はソースレジスタとしてフォワードレジスタ(%F)、次回に演算結果を渡す場合にはディスティネーションレジスタにバックワードレジスタ(%B)をそれぞれ割り付けることにより、ループの繰り返しがIP番号が小さくなる方向に各IPを移動しつつ処理される。その実行の様子を図3-24に示す。図中の網掛けで示したものが隣接するIP間で共有されているフォワード/バックワードレジスタであり、それを利用してデータをIP間で移動しながらループの処理が行われている。

例示したプログラムに対するレジスタのリネーミング結果を以下に示す。以下のコードが3回(アンローリング回数)繰り返されてソフトウエアパイプラインの定常状態を構成している。なお、実際にはこの定常状態の前後にプロローグ部およびエピローグ部が付加されてループ全体に対するソフトウエアパイプライン化コードが完成する。

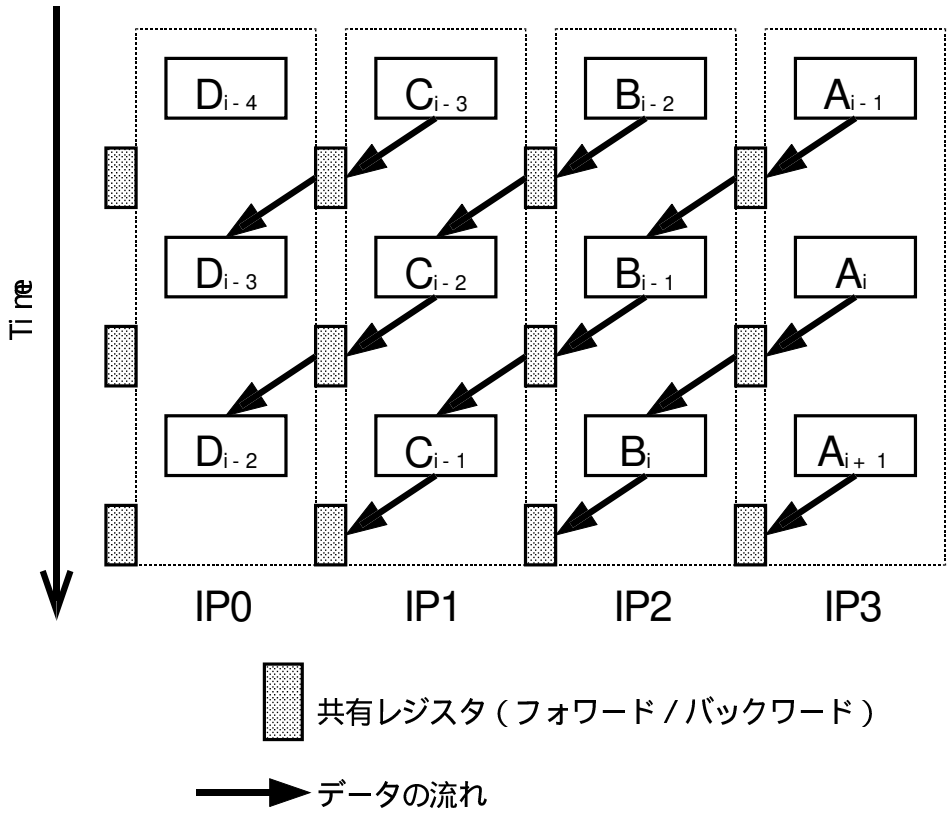
IP1	IP2	IP3	IP4
addi %G1,4,%G1	st %G7,%G1,%F2	add %F2,%F3,%L2	ld %G3,%G1,%L2

```
for (i = 3; i < N; i++) {
```

D_{i-3}	C_{i-2}	B_{i-1}	A_i

```
}
```

(a) ソフトウエアパイプライン化されたループ
（定常状態の部分）



(b) ジェットパイプライン上での実行の様子

図3-24　ジェットパイプライン上でのソフトウエアパイプライニング

```
nop             addi %G2,1,%G2    ld   %G4,%G1,%L3  mul %G5,%L2,%B2
nop             cmp  %G2,%G5      mul  %L2,%L3,%L2  ld  %G0,%G1,%L3
nop             jc   LE,L5        add  %G6,%L2,%B2  mul %G4,%L3,%B3
```

§ 3.6　結言

本章では、代表的な命令レベル並列処理方式であるスーパースカラとVLIWの両方式を融合し、さらにベクトルアーキテクチャも取り入れた新しい高性能プロセッサアーキテクチャであるジェットパイプラインを提案し、そのハードウエアおよびソフトウエアについて述べた。

まずはじめに、これまでの命令レベル並列処理アーキテクチャの長所と欠点について述べ、それらの長所を生かし欠点を補うことのできるジェットパイプライン・アーキテクチャの概念について述べた。次に、ジェットパイプライン・アーキテクチャのハードウエア構成について、各機能ブロックごとに詳細に述べた。最後に、ソフトウエア環境について、命令セット、基本的な命令スケジューリングおよび並列化、ディスパッチスタックを用いた並列化、およびジェットパイプライン・アーキテクチャのためのソフトウエアパイプライニングによる並列化手法について述べた。