(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号

特許第4002554号
(P4002554)

(45)発行日　平成19年11月7日(2007.11.7)　(24)登録日　平成19年8月24日(2007.8.24)

(51)Int.Cl.		F I		
G06F 17/16	(2006.01)	G06F	17/16	G
G06F 9/38	(2006.01)	G06F	9/38	310G
G06F 9/45	(2006.01)	G06F	9/44	322G

請求項の数 5　(全 59 頁)

(21)出願番号　特願2003-586743 (P2003-586743)
(86)(22)出願日　平成15年4月14日 (2003.4.14)
(65)公表番号　特表2006-508414 (P2006-508414A)
(43)公表日　平成18年3月9日 (2006.3.9)
(86)国際出願番号　PCT/US2003/011571
(87)国際公開番号　WO2003/090067
(87)国際公開日　平成15年10月30日 (2003.10.30)
審査請求日　平成18年3月29日 (2006.3.29)
(31)優先権主張番号　10/127,087
(32)優先日　平成14年4月22日 (2002.4.22)
(33)優先権主張国　米国 (US)

前置審査

(73)特許権者　504199127
フリースケール　セミコンダクター　インコーポレイテッド
アメリカ合衆国　７８７３５　テキサス州　オースティン　ウィリアム　キャノン　ドライブ　ウエスト　６５０１
(74)代理人　100116322
弁理士　桑垣　衛
(72)発明者　リー、リー－ウォン
アメリカ合衆国　７８７５８　テキサス州　オースティン　クエイルウッド　ドライブ　９３０３　ナンバーエイ

最終頁に続く

(54)【発明の名称】拡張命令エンコーディングのシステムおよびその方法

(57)【特許請求の範囲】

【請求項１】

繰り返し処理を行うプログラム・ループ命令、プログラム・ループ命令を実行や停止する標準命令、および標準命令を変更する増補命令、からなる複数の命令（４０５）を受け取るステップと、

同複数の命令（４０５）の中から、擬似ベクトル数値命令を識別することによって、繰り返し処理を行うプログラム・ループ（４８０）命令グループを識別するステップと、

データ処理の時に、該プログラム・ループ（４８０）命令グループに属する命令には、３オペランド以上の第１命令サイズを使用するステップと、

データ処理の時に、該プログラム・ループ（４８０）命令グループ以外の命令には、標準命令グループ、および同標準命令グループの命令を増強し補完して、早期終了、継続、または条件実行のうちの少なくとも１つを命令する増補命令グループが使用され、同標準命令グループおよび増補命令グループには、第１命令サイズよりもオペランドの少ない第２命令サイズを使用するステップと、

からなる方法。

【請求項２】

データ処理の時の動作がベクトル・モードである時に、データ処理が、

標準命令を受け取るステップと、

第１デコーダ（４３０）を使用して該標準命令をデコードするステップと、

データ処理の時の動作が擬ベクトル・モードである時に、データ処理システムが、

該標準命令、および外部メモリから増補命令部分を含む命令セットを受け取るステップであって、標準命令グループの命令を増強し補完する該増補命令部分は、早期終了、継続、または条件実行、のうちの少なくとも１つからなる制御情報を含む前記ステップと、
該第１デコーダ（４３０）を使用して該標準命令をデコードするステップと、
第２デコーダ（４４０）を使用して該増補命令部分をデコードするステップと、
からなる方法。
【請求項３】
第１命令サイズを有する第１命令、および第２命令サイズを有する第２命令の、標準命令部分をデコードする第１のデコーダ（４３０）であって、該第１命令サイズが該第２命令サイズと同一であって、該第２命令は擬ベクトル数値命令を含む、前記第１のデコーダと、
該第２命令の擬標準命令部分をデコードする、該第１のデコーダと異なる第２のデコーダ（４４０）であって、該第２命令の第２部分はプログラム・ループを実行するための条件制御情報を含み、同条件制御情報は早期終了、継続、または条件実行、のうちの少なくとも１つを含む前記第２デコーダと、
からなるシステム。
【請求項４】
データ処理システムが、　繰り返し処理を行うプログラム・ループ命令、プログラム・ループ命令を実行や停止する標準命令、および標準命令を変更する増補命令、からなる複数の命令を受け取るステップと、
同複数の命令のうち、擬似ベクトル数値命令を識別することによって、該命令が、繰り返し処理を行うプログラム・ループを表す命令グループを判定するステップと、
該プログラム・ループを表す命令グループ以外の命令を標準命令に変換するステップと、
該命令グループの命令を、標準命令部分と、標準命令グループの命令を増強し補完する増補命令部分とを有する増補命令に変換するステップ、であって、該増補命令部分は、プログラム・ループを実行するための条件制御情報を含み、同条件制御情報は早期終了、継続、または条件実行、のうちの少なくとも１つを含む前記ステップと、
からなる方法。
【請求項５】
データ処理システムが、
コマンドを受け取るステップと、
該コマンドにおいて擬似ベクトル数値命令を識別することによって、該命令が、繰り返し処理を行うプログラム・ループに関連するかどうかを判定するステップと、
該コマンドがプログラム・ループに関連しない場合に、該コマンドを標準命令にコンパイルするステップと、
該コマンドがプログラム・ループに関連する場合に、該コマンドを、標準命令部分および、標準命令グループの命令を増強し補完する増補命令部分を有する増補命令にコンパイルするステップであって、該増補命令部分は、プログラム・ループを実行するための条件制御情報を含み、同条件制御情報は早期終了、継続、または条件実行、のうちの少なくとも１つを含む前記ステップと、
からなる方法。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、一般的にはデータ処理システムでの命令実行に関し、具体的にはループ実行中の拡張長さ命令実行に関する。
【背景技術】
【０００２】
最近、ポケット・ベル、セルラ電話機など、ミッドレンジからローエンドの組み込み応

用向けの低コスト、低電力、高性能プロセッサの設計が注目されている。これらの組込み応用の多くが、ディジタル信号処理（ＤＳＰ）機能など、データ処理システムが高度の繰り返し機能を実行することを必要とし、これらの機能では、大量の命令レベル並列性（Ｉｎｓｔｒｕｃｔｉｏｎ Ｌｅｖｅｌ Ｐａｒａｌｌｅｌｉｓｍ、ＩＬＰ）を利用し得ると同時に、システムが制御集中機能を実行する必要もある。

【０００３】

この必要に対処するために、一部にシステムで、デュアルコア・ソリューションが使用され、この場合に、一方のコアが、制御集中機能を実行し、他方のコアが、特殊化されたＤＳＰ機能を実行する。この手法では、プロセッサ・コアが、共用メモリなどのシステム内で実施される通信チャネルを介して互いに通信する。これらのシステムは、しばしば、実行コアごとに１つのデュアル命令ストリームを使用する。これらのデュアル・コア・システムは、通常は、より高いハードウェア・コストおよび開発コストを有する。

【０００４】

さらに、多くの組込み応用では、一部のループが非常にベクトル化可能であり、他のループが、ベクトル化が困難である。非常にベクトル化可能なループは、「Ｃｒａｙ－１ Ｃｏｍｐｕｔｅｒ Ｓｙｓｔｅｍ Ｈａｒｄｗａｒｅ Ｒｅｆｅｒｅｎｃｅ Ｍａｎｕａｌ」、クレイリサーチ社（Ｃｒａｙ Ｒｅｓｅａｒｃｈ，Ｉｎｃ．）［米国ミネソタ州ブルーミントン（Ｂｌｏｏｍｉｎｇｔｏｎ）］、出版番号２２４０００４、１９７７に記載のものなどの伝統的なベクトル処理パラダイムを使用することによって効率的に処理され得る。これは、ベクトル化可能なループに適用可能であるが、ベクトル化が困難なループには拡張されない。

【０００５】

ベクトル化が困難なループについては、ループ実行の最適化に焦点を合わせたＤＳＰスタイルの処理パラダイムが、より適する。ＡＤＳＰ－２１０６ｘ ＳＨＡＲＣ Ｕｓｅｒ’ｓ Ｍａｎｕａｌ、アナログ・デバイセズ社（Ａｎａｌｏｇ Ｄｅｖｉｃｅｓ Ｉｎｃ．）１９９７年に記載のＳＨＡＲＣ製品が、ループ最適化を使用するシステムの例である。

【非特許文献１】「Ｃｒａｙ－１ Ｃｏｍｐｕｔｅｒ Ｓｙｓｔｅｍ Ｈａｒｄｗａｒｅ Ｒｅｆｅｒｅｎｃｅ Ｍａｎｕａｌ」、クレイリサーチ社（Ｃｒａｙ Ｒｅｓｅａｒｃｈ，Ｉｎｃ．）［米国ミネソタ州ブルーミントン（Ｂｌｏｏｍｉｎｇｔｏｎ）］、出版番号２２４０００４、１９７７

【非特許文献２】ＡＤＳＰ－２１０６ｘ ＳＨＡＲＣ Ｕｓｅｒ’ｓ Ｍａｎｕａｌ、アナログ・デバイセズ社（Ａｎａｌｏｇ Ｄｅｖｉｃｅｓ Ｉｎｃ．）１９９７年

【発明の開示】

【発明が解決しようとする課題】

【０００６】

ベクトル化が困難なループの効率的な性能を提供するが、この手法は、非常にベクトル化可能なループについて効率的でない。

【課題を解決するための手段】

【０００７】

多くの組込み応用例は、その実行時間の大半を、少数のクリティカル・プログラム・ループの実行に費やす。これらのクリティカル・ループは、しばしば、静的コード側のわずかな部分だけを構成する。そのようなシステムでは、性能とシステム・コスト（コード・サイズ）の間の最適トレードオフが、少数のクリティカル・プログラム・ループを除いて、密命令エンコーディング（ｄｅｎｓｅ ｉｎｓｔｒｕｃｔｉｏｎ－ｅｎｃｏｄｉｎｇ）方式をプログラム全体について使用した場合に達成され得ることが多い。上の議論から、命令エンコーディングの改善された方法が必要であることは明白である。

【０００８】

本発明の特定の実施形態を、本明細書で提示する図面に示し、説明する。本発明の様々な目的、長所、特徴、および特性、ならびに構造の関連する要素の方法、動作、および機

能と、部分および製造の経済の組合せは、そのすべてが本明細書の一部を形成する、添付図面を参照して以下の説明および請求項を考慮する際に明らかになる。

【発明を実施するための最良の形態】

【０００９】

１６ビット固定命令長エンコーディングを使用するＩＳＡ（Ｉｎｓｔｒｕｃｔｉｏｎ Ｓｅｔ Ａｒｃｈｉｔｅｃｔｕｒｅ）を有する普通のシステムでは、しばしば、命令エンコーディングに対する様々な制限によって性能が悪影響を受ける。たとえば、ＩＳＡでは、通常は、３オペランド命令フォーマットではなく、２オペランド命令フォーマットが使用される。この場合に、命令は、通常、オペコード、Ｒａ、Ｒｂというフォーマットを有し、ここで、Ｒａは、ソース・オペランドならびにデスティネーション・オペランドである。その結果、Ｒａの元の値を保存する必要がある場合には、この命令のオペランド自己破壊的性質に起因して、追加の「ｍｏｖｅ」命令が必要である。このＩＳＡの命令は、「即」値フィールドをエンコードするのに限られた幅も有する。さらに、ＩＳＡは、条件実行機能（たとえば、条件コードがある値である場合に命令を条件的に実行し得ること）をエンコードし得ない。この機能は、短い順方向分岐を除去するのに有利である。命令エンコーディングでのこれらの制限を有するので、１６ビット固定命令長エンコーディングを使用するＩＳＡは、通常、３２ビット固定命令長エンコーディングを使用するＩＳＡより低い性能を有する。しかし、前者は、後者より高いコード密度（したがって低いシステム・コスト）を有する。

【００１０】

クリティカル・ループについて、本発明では、拡張された命令幅エンコーディングを使用して、大量のプログラム・コードに使用される狭い命令エンコーディングに関連する性能劣化を防ぐ。下記の例を検討されたい。組込みシステムは、その実行時間の約４０％を、少数のクリティカル・ループの実行に費やす。これらのループは、プログラムの静的コード・サイズの約１％を構成する。これらのループだけについて命令エンコーディングを拡張することによって、これらのループのサイズを１００％だけ増やす。それと同時に、このループの実行時間を１００％だけ改善することも可能である。全体として、２０％だけ性能が向上する。この例の静的コード・サイズは、１％増えるのみである。

【００１１】

本発明は、統合されたデータ・パスでスカラ機能およびベクトル機能を実行する方法を提供する。スカラ機能は、制御機能とみなすことも可能であり、ベクトル機能は、通常は、反復プログラム・ループを置換し、実行するためのものである。この方法では、ループの性質に応じて、従来のベクトル処理パラダイムまたはＤＳＰスタイルの処理パラダイムの間で選択する。プログラム・ループに関するこれらの処理パラダイムの両方を実行する能力を提供することによって、個々のパラダイムより高い性能改善を達成し得る。さらに、ベクトル機能の命令に関連する機能を、増補命令部分の使用を介して拡張し得る。増補命令部分は、ベクトル機能に関連する命令と共に提供される。

【００１２】

本明細書で使用する用語「バス」は、データ、アドレス、制御、または状況など、１つまたは複数の様々なタイプの情報を転送するのに使用し得る、複数の信号または導体を指すのに使用される。用語「アサート」および「ネゲート」は、信号、状況ビット、または類似する装置を、論理的に真の状態および論理的に偽の状態にすることを指す時に使用される。論理的に真の状態が、論理レベル１である場合に、論理的に偽の状態は、論理レベル０である。論理的に真の状態が、論理レベル０である場合には、論理的に偽の状態は、論理レベル１である。

【００１３】

大括弧は、バスの導体または値のビット位置を示すのに使用される。たとえば、「バス６０［０から７］」または「バス６０の導体［０から７］」は、バス６０の８つの下位導体を示し、「アドレス・ビット［０から７］」または「アドレス［０から７］」は、アドレス値の下位８ビットを指す。数字の前の記号「＄」は、その数が１６進数または１６を

底とする形で表されることを示す。数字の前の記号「％」は、その数が２進数または２を底とする形で表されることを示す。

【００１４】

ベクトル処理パラダイムで、データは、メモリまたはベクトル・レジスタから継続的に流れ、非常にパイプライン化された形の一連の機能ユニットによって処理される。処理されたデータは、メモリに継続的に流れる。上で述べたＣｒａｙアーキテクチャが、このパラダイムを使用するシステムの初期の例である。

【００１５】

ベクトル処理パラダイムの威力を、次の例のループを使用して示し得る。

【実施例１】

【００１６】

【数１】

```
L1:
addi        R2,2        // update stride value
ld.h        R7,(R2)     // load with stride of 2
addi        R3,1        // update stride value
ld.b        R6,(R3)     // load with stride of 1
mul         R7,R6       // multiply
st.h        R7,(R2)     // store with stride of 2
dec.ne      R1          // decrement loop index r1
                        // set c bit if r1 not equals zero
bt          L1          // branch to L1 if c bit is set
```

このループは、２つのベクトルに対して要素単位の乗算を実行する。このループは、ベクトル形式で、ベクトルＡ、Ｂ、ＣについてＣ［ｉ］＝Ａ［ｉ］＊Ｂ［ｉ］、ｉ＝０、．．．、ｎ－１を実行する。

【００１７】

この例では、中間値が、作られ、即座に消費される。これらの値は、連続的にレジスタｒ６およびｒ７に書き戻され、読み取られる。これらは、作られ、１回だけ消費されるので、これらの値を、しばしば限られているレジスタ記憶空間に蓄積することは非効率的である。この状況は、残念ながら、ベクトル動作がｌｏａｄ－ｓｔｏｒｅ　ＩＳＡ（Ｉｎｓｔｒｕｃｔｉｏｎ　Ｓｅｔ　Ａｒｃｈｉｔｅｃｔｕｒｅ）を使用して表される時に、不可避である。

【００１８】

より効率的な手法は、複数の機能ユニットを一緒にチェーンにし、各ユニットが特定のタスクを実行することである。この形で、中間値が機能ユニットによって作られる時に、その値が、チェーン内の次の機能ユニットに直接に渡され、これによって、値のストアおよび検索に関連するレジスタ・ファイルへの読取トラフィックおよび書込トラフィックが回避される。

【００１９】

ベクトル動作またはベクトル実行をパイプライン化し、すべてのサイクルに１つの結果を作り得るようにすることも可能である。一般に、ベクトル動作を実行するのに必要な時間は、Ｔｓ＋ｎ／Ｔｒによって与えられ、ここで、Ｔｓは、初期セットアップ・コストであり、ｎは、ベクトルの長さであり、Ｔｒは、各サイクルに作られる結果の個数単位のスループット・レートである。スカラ計算機が、同等のプログラム・ループの１つの反復を実行するのにｍサイクルを要する場合に、ベクトル計算機を使用することによる速度向上

は、ｎｍ／（Ｔｓ＋ｎ／Ｔｒ）またはＴｒ＝１の場合にｎｍ／（Ｔｓ＋ｎ）によって与えられる。最大の速度向上は、Ｔｓが十分に小さく、ｎが十分に大きい時に達成し得る。この場合に、速度向上がｍに達し、これは、スカラ計算機がプログラム・ループの１つの反復を実行するのに要するサイクル数である。

【００２０】

ベクトル処理は、複数の効果を有する。様々な動作を単一のデータ・ストリームでパイプライン化し、改善された性能をもたらし得る。効率的なデータのストレージおよび移動がもたらされる。というのは、大量の一時データが作られ、レジスタ・ファイルまたはメモリ・システムを通らずに、隣接する機能ユニットによって消費されるからである。さらに、ベクトル処理では、小さいルーティング区域が使用される。というのは、データ・パス全体にブロードキャストするのではなく、機能ユニットが、デスティネーション機能ユニットに直接にルーティングされるからである。効率的なデータ移動およびより小さいルーティング区域によって、電力消費が減る傾向がある。さらに、ベクトル実行中に命令を要求する必要がないので、より低い命令フェッチ帯域幅が達成される。ベクトル処理パラダイムの威力によって、ベクトル処理パラダイムが、低コスト、低電力組込みコンピュータ・システムに非常に適するようになる。

【００２１】

あるベクトルＡ、Ｂ、およびＣについて、

Ｃ［ｉ］＝Ａ［ｉ］＊Ｂ［ｉ］、ｉ＝０、．．．、ｎ－１

によって記述されるベクトル演算を実行するループは、非常にベクトル化可能なループである。ループ・ベクトル化のしやすさは、通常は、ハードウェアおよびシステム構成の関数である。一般に、実行される算術関数を決定するのに実行時情報に依存しないループは、ベクトル化しやすい。本明細書で使用されるＣＶＡ（Ｃａｎｏｎｉｃａｌ　Ｖｅｃｔｏｒ　Ａｒｉｔｈｍｅｔｉｃ）は、非常にベクトル化可能なループによって実行し得るベクトル算術を表す。下に、ＣＶＡのもう１つの例を示す。

【００２２】

Ｃ［ｉ］＝４＊Ａ［ｉ］＋（Ｂ［ｉ］＞＞１）、ｉ＝０、．．．、ｎ－１

このＣＶＡは、次のように複数のＣＶＡに分解し得る。ある一時ベクトルＴ１およびＴ２について、

Ｔ１［ｉ］＝４＊Ａ［ｉ］；Ｔ２［ｉ］＝Ｂ［ｉ］＞＞１；Ｃ［ｉ］＝Ｔ１［ｉ］＋Ｔ２［ｉ］、ｉ＝０、．．．、ｎ－１

ＤＳＰアルゴリズムまたはＤＳＰ機能は、組込み計算機で実施される時に、しばしばプログラム・ループに変換される。最適化コンパイラは、ループを再構成し、すべての可能な並列性を計算機によって簡単に利用し得るようにする。しかし、そのようなプログラム・ループは、ベクトル化が簡単でない。そのようなプログラム・ループは、コンパイラによってベクトル処理パラダイムに「あてはまる」ように変換された後にベクトル化可能になる可能性がある。これらの変換には、マスク生成、動作の収集および分散などが含まれる可能性がある、ある追加のベクトル動作の追加が含まれる。

【００２３】

たとえば、

Ｃ［ｉ］＝（Ａ［ｉ］＞Ｂ［ｉ］）？Ａ［ｉ］^{2}：Ａ［ｉ］＋Ｂ［ｉ］、ｉ＝０、．．．、ｎ－１

によって記述されるベクトル動作を実行するループは、ベクトル化が困難であるか高コストである。具体的に言うと、このループは、条件

Ａ［ｉ］＞Ｂ［ｉ］

に動的に頼って、結果の要素Ｃ［ｉ］を得るために実行される算術関数を決定する。このタイプの算術を、本明細書ではＰＶＡ（Ｐｓｅｕｄｏ－Ｖｅｃｔｏｒ　Ａｒｉｔｈｍｅｔｉｃ）と称する。

【００２４】

ＤＳＰタイプの計算機は、プログラム・ループの実行を最適化することによって、ＰＶ

Ａ算術を効率的に実行する。これらの計算機で、（ｉ）ループ制御機構、（ｉｉ）定数ストライド・ロード、および（ｉｉｉ）定数ストライド・ストアに関連するオーバーヘッドの多くを除去することによって性能が改善される。

【００２５】

伝統的なＤＳＰプロセッサの命令によって、複数の計算動作およびメモリ動作を並列に実行することを指定し得る。そのようなプロセッサの動作は、複数の動作が並列に発行されるＶＬＩＷ（Ｖｅｒｙ　Ｌｏｎｇ　Ｉｎｓｔｒｕｃｔｉｏｎ　Ｗｏｒｄ）プロセッサの動作に非常に類似する。

【００２６】

本発明は、ベクトル・タイプ処理を使用するベクトル化可能ループおよびＤＳＰタイプの処理を使用するベクトル化が困難または不可能なループの処理の手段を提供する。ループのタイプに応じて、計算機は、ある条件の下ではベクトル・プロセッサのように振る舞い、他の条件の下ではＤＳＰプロセッサのように振る舞う。さらに、この計算機は、単一データ・パスを使用して、プログラムのすべてのベクトル算術ならびにスカラ部分（すなわち非ループ部分）を実行し、同一のハードウェア・リソースの効率的な再利用を可能にする。

【００２７】

本発明は、ベクトル・タイプ処理またはＤＳＰタイプ処理のどちらを使用するかを決定する判断機構としてベクトル化を組み込む。前者を、ＣＶＡ実行とも称する。後者を、ＰＶＡ実行とも称する。この新しい処理パラダイムを、本明細書では「擬似ベクトル計算機（Ｐｓｅｕｄｏ－Ｖｅｃｔｏｒ　Ｍａｃｈｉｎｅ）」と称する。

【００２８】

この新しい処理パラダイムによれば、実行モデルが、２つのモードすなわち、（ｉ）制御機能に関するスカラ実行モードおよび（ｉｉ）命令レベル並列性（ｉｎｓｔｒｕｃｔｉｏｎ　ｌｅｖｅｌ　ｐａｒａｌｌｅｌｉｓｍ、ＩＬＰ）を利用するベクトル実行モードで動作し得る。図１に、本発明の１実施形態の実行モードを示す。２つの基本モードが、スカラおよびベクトルであり、ベクトル・モードには、さらに特定の実行モードが含まれる。ＣＶＡおよびＰＶＡは、ベクトル・モードでのみ使用可能であり、ＣＶＡモード実行には、（ｉ）コンパウンド（ｃｏｍｐｏｕｎｄ）、（ｉｉ）リダクション（ｒｅｄｕｃｔｉｏｎ）、および（ｉｉｉ）ハイブリッド（ｈｙｂｒｉｄ）という３つのタイプがある。それに対応して、この計算機には、２つのベクトル命令すなわち、ＣＶＡ命令およびＰＶＡ命令がある。

【００２９】

図１の実行モデルでは、スカラ・モードおよびベクトル・モードが、時間的にオーバーラップしない。モデルは、単一の命令ストリームを使用して、単一のデータ・パスでこの２つのモードを実行する。このストリーム内の各命令は、スカラ命令またはベクトル命令のいずれかとして分類し得る。ベクトル命令がフェッチされ、デコードされる時に、計算機は、ベクトル実行モードに入る。計算機は、少数の事前定義の機構を介してのみベクトル・モードから出る。

【００３０】

非常にベクトル化可能であるループに関して、計算機は、ＣＶＡ実行モードすなわち、「真の」ベクトル処理パラダイムを使用して、ループを処理する。ベクトル化が困難または不可能であるループに関して、計算機は、ＰＶＡ実行モードすなわち、ＤＳＰスタイルの処理に似た「擬似」ベクトル処理パラダイムを使用して、ループを処理する。最適化コンパイラは、この場合に、所与のプログラム・ループにどの実行モードが最も適するかを判断する。一般に、コンパイラは、まず、ベクトル処理パラダイムの低電力態様および高性能態様を利用することを試みて、真のベクトル命令またはＣＶＡ命令を使用するループのベクトル化を試みる。これが可能でない場合には、コンパイラは、ＰＶＡ命令または両方の組合せを使用して、ＤＳＰスタイルのループベース実行にフォール・バックする。

【００３１】

普通の意味でのベクトル化は、ベクトル化可能プログラム・ループを識別し、ある同等のベクトル演算と置換することを指す。さらに、本発明は、プログラム・ループを識別し、ＤＳＰスタイルのループ構造と置換するベクトル化を提供する。これには、ＤＯ　ＵＮＴＩＬ命令またはＤＯ　ＷＨＩＬＥ命令を含め得る。そのようなループは、普通の意味ではベクトル化が困難または不可能である可能性がある。擬似ベクトル計算機で、ＰＶＡ命令が、ＤＳＰタイプ・プログラム・ループの構成およびベクトル化に使用される。

【００３２】

プログラム・ループが、１つまたは複数のベクトル命令（ＣＶＡ命令および／またはＰＶＡ命令）からなる同等のコードに置換される時に、そのプログラム・ループを、ベクトル化されたという。このベクトル化されたコード・セグメントを、元のスカラ・プログラム・ループと同等の機能を実行するので、元のスカラ・プログラム・ループのベクトル同等物とも称する。ベクトル化は、アセンブリ・レベルまたはソース・コード・レベルで行い得る。

【００３３】

ループが、ＣＶＡ構成を使用してベクトル化可能である場合に、これをＣＶＡベクトル化可能という。ループが、ＰＶＡ構成を使用してベクトル化可能である場合に、これをＰＶＡベクトル化可能という。ループが、ＣＶＡベクトル化可能である場合に、そのループは、ＰＶＡベクトル化可能でもある。しかし、逆は一般に真でない。ＰＶＡ構成は、より一般的なベクトル化機構を表す。同等のハードウェア・コストで、ＣＶＡ実行は、通常は、非常にベクトル化可能であるループのより高い性能の利益を提供する。ベクトル化が不可能またはコストが高すぎるループについて、ＰＶＡ実行が、よりよい性能の利益を提供する。

【００３４】

ＤＳＰタイプ実行および／またはＶＬＩＷタイプ実行と、ベクトル・タイプ実行との間の区別を、図２および３に示す。図２に、ＤＳＰタイプ実行の動作を示すが、ここでは、複数の独立の動作が、複数の機能ユニットに同時に発行される。これらの独立の機能ユニットによって作られた結果が、あるアーキテクチャ的レジスタおよび／またはメモリ・システムに書き戻される。図２に示された動作によって、ＶＬＩＷ計算機の動作も記述されることに留意されたい。

【００３５】

図３に、伝統的なベクトル実行を示すが、ここでは、複数の機能ユニットが、一緒にチェーンにされて、依存する動作を実行する。チェーンされた機能ユニットの間の一時的結果は、アーキテクチャ的レジスタに書き戻されず、メモリ・システムにも書き込まれない。さらに、ベクトル実行は、ベクトル命令の最初のフェッチおよびデコードの後に、そのベクトル実行の残りについてさらなる命令要求をまったく行わないという事実も特徴とする。

【００３６】

本発明は、ＤＳＰタイプ実行とベクトル実行を１つの統合されたデータ・パスで組み合わせる方法を提供する。本明細書で使用されるＰＶＡ実行は、ＤＳＰタイプ実行であり、ＣＶＡ実行は、ベクトル実行である。したがって、本発明は、１つの処理システム内の各タイプの実行の利益を利用する。

【００３７】

図４に、本発明の１実施形態による処理システム２を示す。処理システム２には、単一のデータ・パスでスカラ実行、ＣＶＡ実行、およびＰＶＡ実行を実行する処理アーキテクチャが組み込まれる。実行コア４に、第１ロード・ユニットL_0　６および第２ロード・ユニットL_1　８が含まれる。情報は、メモリＭ０　１４からL_0　６に、およびメモリＭ１　１６からL_1　８にロードされる。

【００３８】

本発明の１実施形態で、Ｍ０　１４およびＭ１　１６が、ランダム・アクセス・メモリ（ＲＡＭ）ブロックであるが、他のタイプのメモリ・ストレージ・デバイスを使用して実

施し得る。メモリＭ０　１４は、Ｍ０＿ｄｂｕｓを介してデータ情報を、Ｍ０＿ａｂｕｓを介してアドレス情報を、処理システム２の残りと通信する。同様に、Ｍ１　１６は、Ｍ１＿ｄｂｕｓを介してデータ情報を、Ｍ１＿ａｂｕｓを介してアドレス情報を通信する。

【００３９】

実行コア４に、プログラム・シーケンサ２４、Ｍ０＿ｄｂｕｓ、およびＭ０＿ａｂｕｓに結合されたループ・キャッシュ２２も含まれる。プログラム・シーケンサ２４は、Ｍ０＿ｄｂｕｓおよびＭ０＿ａｂｕｓにも結合され、プログラム・シーケンサ２４には、さらに、カウント・インデックス・レジスタ（ＣＩＲ）５０が含まれる。ＣＩＲ５０には、２つの独立のレジスタすなわち（ｉ）カウント・レジスタ（ＣＲ）および（ｉｉ）インデックス・レジスタ（ＩＸＲ）が含まれる。ＣＩＲ５０は、図１１にも示されており、この図では、ＣＲ５１が、ＣＶＡ実行のベクトル長またはＰＶＡ実行の反復回数を指定するのに使用される。ループ・キャッシュは、Ｍ０＿ｄｂｕｓを介してデータを受け取るように結合される。プログラム・シーケンサ２４は、Ｍ０＿ａｂｕｓを介してＭ０　１４にアドレス情報を提供する。

【００４０】

レジスタ・ファイル（ＲＦ）２６も設けられ、データが、$Ｌ_0$ ６からＬ０＿ｄｂｕｓを介して、および$Ｌ_1$ ８からＬ１＿ｄｂｕｓを介してＲＦ２６に供給される。ＲＦ２６、$Ｌ_0$ ６、および$Ｌ_0$ ８のすべてが、マルチプレクサ２８、３０、および３２に結合される。マルチプレクサ２８は、ｘｂｕｓを介して主算術ユニット（Ｐ）３４に情報を供給する。１実施形態で、Ｐ３４は、汎用機能ユニットである。Ｐ３４は、基礎になるＩＳＡで定義されたスカラ算術機能のすべてを実行し得る。マルチプレクサ３０は、ｙｂｕｓを介してＰ３４に情報を供給する。

【００４１】

Ｐ３４の結果は、ｐ＿ｂｕｓを介して副算術／ストア・ユニット（Ｓ）３６に供給される。Ｐ３４の結果は、マルチプレクサ４０にも供給される。メモリ・ストア動作を実行するほかに、Ｓ３６は、「ａｄｄ」、「ａｎｄ」、「ｏｒ」、「ｘｏｒ」などの単純な算術論理ユニット（ＡＬＵ）算術も実行し得る。マルチプレクサ３２は、ｚｂｕｓを介してラッチ３８に情報を供給し、この情報は、ラッチ３８からマルチプレクサ４１に供給される。マルチプレクサ４１の出力は、ｚｓ＿ｂｕｓを介してＳ３６に情報を供給する。Ｓ３６のデータ出力は、ｓ＿ｄｂｕｓを介してマルチプレクサ４０、マルチプレクサ４１、Ｍ１＿ｄｂｕｓ、Ｍ０＿ｄｂｕｓ、および一時メモリ（商標）２０に供給される。Ｓ３６のアドレス出力は、ｓ＿ａｂｕｓを介してＭ０＿ａｂｕｓ、Ｍ１＿ａｂｕｓ、および一時メモリ（ＴＭ）２０にも供給される。マルチプレクサ４０の出力は、ｒｅｓｕｌｔ＿ｂｕｓを介して、ＲＦ２６、$Ｌ_0$ ６、$Ｌ_1$ ８、ＳＳＲ４２、マルチプレクサ２８、およびマルチプレクサ３０に情報を供給する。

【００４２】

スカラ・モードで実行する時に、実行コアは、単一発行パイプライン式計算機のように振る舞う。実行コアは、スカラ計算にレジスタ・ファイルＲＦ２６およびＰ３４を使用し、メモリ・ロード／ストア動作に$Ｌ_0$ ６、$Ｌ_1$ ８、およびＳ３６を使用する。具体的に言うと、これらのロード／ストア動作のメモリ・アドレスが、Ｓ３６によって、ＴＭ２０、Ｍ０　１４、およびＭ１　１６を含むメモリ・システムに供給される。メモリ・ストア動作について、データが、Ｓ３６ユニットによって供給される。メモリ・ロード動作について、データは、メモリ・システムによって、$Ｌ_0$ ６および$Ｌ_1$ ８を介し、Ｌ０＿ｄｂｕｓおよびＬ１＿ｄｂｕｓを介して、ＲＦ２６およびＰ３４に供給される。

【００４３】

ベクトル・モードで実行する時に、ベクトル命令すなわち、ＣＶＡ命令またはＰＶＡ命令によって、任意選択として、２つまでの入力データ・ストリーム$Ｌ_0$ および$Ｌ_1$ および１つのデータ・ストリームをイネーブルし得る。

【００４４】

ベクトル・モードで実行する時に、データを、Ｍ０　１４から$Ｌ_0$ ６およびＬ０＿ｄｂ

ｕｓを介して継続的にフェッチし、ＲＦ２６、Ｐ３４、またはＳ３６への複数のパスのいずれかによって供給し得る。第１パスは、Ｌ０＿ｄｂｕｓを介してＲＦ２６にデータをストリーミングするのに使用される。第２パスは、マルチプレクサ２８を介してＰ３４にデータをストリーミングするのに使用され、第３パスは、マルチプレクサ３０を介してＰ３４にデータをストリーミングするのに使用される。第４パスは、マルチプレクサ３２、ラッチ３８、およびマルチプレクサ４１を介してＳ３６にデータをストリーミングするのに使用される。ＰＶＡ実行に関して、第１パス、第２パス、および第３パスの１つまたは複数を、ＰＶＡ命令に応じて使用し得る。ＣＶＡ命令に関して、第２パス、第３パス、および第４パスの１つまたは複数を、ＣＶＡ命令に応じて使用し得る。これらのデータ・ストリームのどれをも、入力データ・ストリームL_0と称する。

【００４５】

同様に、ベクトル・モードで実行する時に、データを、$L_1$８およびＬ１＿ｄｂｕｓを介してＭ１　１６から継続的にフェッチし、複数のパスのいずれかによってＲＦ２６、Ｐ３４、またはＳ３６に供給し得る。第１パスは、Ｌ１＿ｄｂｕｓを介してＲＦ２６にデータをストリーミングするのに使用される。第２パスは、マルチプレクサ２８を介してＰ３４にデータをストリーミングするのに使用され、第３パスは、マルチプレクサ３０を介してＰ３４にデータをストリーミングするのに使用される。第４パスは、マルチプレクサ３２、ラッチ３８、およびマルチプレクサ４１を介してＳ３６にデータをストリーミングするのに使用される。ＰＶＡ実行に関して、第１パス、第２パス、および第３パスの１つまたは複数を、ＰＶＡ命令に応じて使用し得る。ＣＶＡ命令に関して、第２パス、第３パス、および第４パスの１つまたは複数を、ＣＶＡ命令に応じて使用し得る。これらのデータ・ストリームのどれをも、入力データ・ストリームL_1と称する。

【００４６】

また、ベクトル・モードでは、データを、Ｓ３６からメモリ・モジュールＭ０　１４、Ｍ１　１６、またはＴＭ２０の１つに継続的にストアし得る。この出力データ・ストリームを、出力データ・ストリームＳと称する。

【００４７】

実行コア４に、さらに、対応するロード／ストア動作がイネーブルされる場合にL_0ストリーム、L_1ストリーム、およびＳストリームのストライドおよびオペランド・サイズを指定するストライド・サイズ・レジスタ（ＳＳＲ）４２が含まれる。ＳＳＲ４２は、$L_0$６、$L_1$８、ＲＦ２６およびＳ３６に結合される。ＳＳＲ４２は、さらに、ｒｅｓｕｌｔ＿ｂｕｓを介して情報を受け取るためにマルチプレクサ４０に結合される。一時メモリ・ストレージ・ユニット、ＴＭ２０が、Ｓ３６、$L_0$６、および$L_1$８に結合される。ＴＭ２０は、一時ベクトルまたは任意のデータのストアに使用し得る。ＴＭ２０にストアされるベクトルは、固定ベクトル長に制限されない。

【００４８】

処理システム２で実行されるＣＶＡの３つのタイプを表すデータ依存性グラフを、図５に示す。図５に示されたＣＶＡの３つのタイプのすべてで、２つの入力の近くで実行される第１算術を、主算術と呼び、ｐ＿ｏｐと表す。この算術は、図３のＰ３４によって実行される。出力の近くで実行される第２算術を、副算術と呼び、ｓ＿ｏｐと表す。この算術は、図３のＳ３６によって実行される。

【００４９】

図５に、本発明の１実施形態に関する、ＣＶＡの入力データ・ストリームL_0およびL_1とソース・オペランドＸ、Ｙ、およびＺの間の関係を示す。オペランドＸは、L_0ストリーム、L_1ストリーム、またはレジスタｒ４からソーシングし得る。オペランドＹは、L_0ストリーム、L_1ストリーム、またはレジスタｒ５からソーシングし得る。オペランドＺは、L_0ストリーム、L_1ストリーム、またはレジスタｒ６からソーシングし得る。Ｘオペランドおよびオペランドは、主算術ｐ＿ｏｐで使用される。主算術の結果が、副算術ｓ＿ｏｐに転送される。ＣＶＡのタイプに応じて、副算術ｓ＿ｏｐは、オペランドＺまたはｓ＿ｏｐ自体の出力からソーシングし得る。

【００５０】
図５の（Ａ）に示されたコンパウンドＣＶＡは、ベクトル計算の結果としてデスティネーション・ベクトルを作る。コンパウンドＣＶＡの一般形式を次のように表し得る。
ソースＸおよびソースＹがすべてベクトルである場合に
$R_i = (X_i \ p\_op \ Y_i)\ s\_op \ Z_i$、ｉ＝０、．．．、ｎ－１
ここで、ｎは、ベクトルの長さを表し、ｐ＿ｏｐは、主算術を表し、ｓ＿ｏｐは、副算術を表し、R_iは、デスティネーション・ベクトルの第ｉ要素を表し、X_i、Y_i、およびZ_iは、それぞれベクトルＸ、Ｙ、およびＺの第ｉ要素である。
【００５１】
ソースＸが、スカラｘ定数であり、ソースＹがベクトルである場合には
$R_i = (x \ p\_op \ Y_i)\ s\_op \ Z_i$、ｉ＝０、．．．、ｎ－１
である。
【００５２】
ソースＸが、ベクトルであり、ソースＹが、スカラ定数ｙである場合には、
$R_i = (X_i \ p\_op \ y)\ s\_op \ Z_i$、ｉ＝０、．．．、ｎ－１
である。副算術ｓ＿ｏｐを、「ノー・オペレーション」にすることも可能であることに留意されたい。
【００５３】
リダクションＣＶＡについて、図５の（Ｂ）に示されているように、Ｘオペランドおよびオペランドのソースは、コンパウンド・タイプと同一であるが、Ｚオペランドは使用されない。副算術の出力が、ｓ＿ｏｐ自体への入力としてフィードバックされる。この場合に、Ｓストリームはディスエーブルされる。リダクションＣＶＡは、ベクトル・リダクション演算を実行し、この動作では、１つまたは複数のベクトルが、ベクトル演算の結果として、スカラ結果に換算される。リダクションＣＶＡの一般形式を、次のように表し得る。
【００５４】
ソースＸおよびソースＹのすべてがベクトルである場合に、
$S_0 = (X_0 \ p\_op \ Y_0)$
$S_i = (X_i \ p\_op \ Y_i)\ s\_op \ S_{i-1}$、ｉ＝１、．．．、ｎ－１；
$r = S_{n-1}$
ここで、S_iは、ｉ番目の部分的結果を表し、ｒは、ベクトル・リダクション演算のスカラ結果を表す。
【００５５】
ソースＸが、スカラ定数ｘであり、ソースＹがベクトルである場合には、
$S_0 = (x \ p\_op \ Y_0)$
$S_i = (x \ p\_op \ Y_i)\ s\_op \ S_{i-1}$、ｉ＝１、．．．、ｎ－１；
$r = S_{n-1}$
ソースＸが、ベクトルであり、ソースＹが、スカラ定数ｙである場合には、
$S_0 = (X_0 \ p\_op \ y)$
$S_i = (X_i \ p\_op \ y)\ s\_op \ S_{i-1}$、ｉ＝１、ｎ－１；
$r = S_{n-1}$
である。
【００５６】
副算術に関連するフィードバック・パスは、部分的結果を計算し、累算し、最終的に最後のスカラ結果を作る責任を負う。そのようなリダクション演算の例が、Σ_i（Ａ［ｉ］＊Ｂ［ｉ］）によって記述される、２つのベクトルの内積である。この場合に、主算術は、「乗算」関数であり、副算術は、累算的「加算」関数である。
【００５７】
図５を継続すると、（Ｃ）として示されたハイブリッドＣＶＡを用いると、リダクションの場合と同一の、ＸオペランドおよびＹオペランドのソースならびに副算術のフィード

バックが可能になるが、ハイブリッドの場合には、Ｓストリームがイネーブルされる。ハイブリッドＣＶＡは、中間の部分的結果が、Ｓストリームを介してデスティネーション・ベクトルに継続的に書き込まれることを除いて、リダクションＣＶＡと同一である。ハイブリッドＣＶＡの一般形式は、部分的結果$Ｓ_{ｉ}$、ｉ＝０、．．．、ｎ－１も、デスティネーション・ベクトルＲを形成し、$Ｒ_{ｉ}＝Ｓ_{ｉ}$、ｉ＝０、．．．、ｎ－１であることを除いて、リダクションＣＶＡと同一である。ハイブリッドＣＶＡについて、２つのデスティネーションすなわち、スカラ・デスティネーションおよびベクトル・デスティネーションがある。

【００５８】

ハードウェア・コストを制限するために、本発明の１実施形態では、複数の制約を課す。第１に、コンパウンドＣＶＡでは、組み合わされる２つの算術が、２つまでの別個のベクトルからソーシングすることだけが可能である。しかし、この制約は、２つまでの別個のベクトルからソーシングする限り、３つのすべてのソース・オペランドがベクトルである場合を除外しない。たとえば、Ｃ［ｉ］＝Ａ［ｉ］＊Ａ［ｉ］＋Ｂ［ｉ］によって記述されるベクトル演算は、２つの別個のベクトルＡおよびＢからソーシングする、３つのソース・ベクトル・オペランドを有する。別個のソース・ベクトルの数に関するこの制約によって、最大メモリ帯域幅要件が、１サイクルあたり３つのデータ・フェッチから１サイクルあたい２つのデータ・フェッチに減る。

【００５９】

上記の制約のほかに、副算術を、少数の単純な可換ＡＬＵ関数に制限し得る（ＡＬＵ関数ｏｐは、すべてのスカラｘおよびｙについてｘ　ｏｐ　ｙ＝＝ｙ　ｏｐ　ｘである場合に、可換という）。この関数に、ａｄｄ、ａｎｄ、ｏｒ、またはｘｏｒなどが含まれる。

【００６０】

ＣＶＡの少数の例を、下の表に示す。

【表１】

表１．ＣＶＡの例

例	ベクトル算術	説明	CVA ベクトル命令	イネーブルされるストリーム		
				L0	L1	S
(i) コンパウンドCVA						
(a)	C[i]=sA[i]+B[i]	ベクトル定数の乗算および加算	CVA mul r4、@L0、@P、add @P、@L1、@S;	Y	Y	Y
(b)	$C[i]=(A[i])^2+B[i]$	要素単位の二乗および加算	CVA mul @L0、@L0、@P、add @P、@L1、@S;	Y	Y	Y
(c)	$C[i]=(A[i])^2$	要素単位の二乗	CVA mul @L0、@L0、@S;	Y	N	Y
(d)	C[i]=abs(A[i])	要素単位の絶対値	CVA abs @L0、@S;	Y	N	Y
(e)	C[i]=A[i]	ベクトル代入	CVA mov @L0、@S;	Y	N	Y
(f)	C[i]=0	メモリ・ブロック初期化	CVA mov 0、@S;	N	N	Y

例	ベクトル算術	説明	CVA ベクトル命令	イネーブルされるストリーム		
				L0	L1	S
(ii)リダクション CVA						
(g)	$IP=\Sigma_i(A[i]*B[i])$	ベクトル内積	CVA mul @L0、@L1、@P、add r3、@P、r3;	Y	Y	N
(h)	$Norm^2=\Sigma_i(A[i])^2$	ベクトルＡの「ノルム」の二乗	CVA mul @L0、@L0、@P、add r3、@P、r3;	Y	N	N
(i)	$Sum=\Sigma_i A[i]$	合計を介するベクトル・リダクション	CVA mov @L0、@P、add r3、@P、r3;	Y	N	N

例	ベクトル算術	説明	CVA ベクトル命令	イネーブルされるストリーム		
				L0	L1	S
(iii) ハイブリッド CVA						
(j)	C[i]=A[i]*B[i]; $IP=\Sigma_i(A[i]*B[i])$	ベクトル乗算およびベクトル内積	CVA mul @L0、@L1、@P、add r3、@P、{@S, r3};	Y	Y	Y

この例のそれぞれに、対応するＣＶＡ命令ならびにＬ$_0$ストリーム、Ｌ$_1$ストリーム、およびＳストリームのイネーブル設定およびディスエーブル設定も詳細に示されている。このＣＶＡ命令で、「@」は、データ・ストリームを表す。具体的に言うと、「@Ｌ０」は、メモリからの第１入力データ・ストリームＬ$_0$を表し、「@Ｌ１」は、メモリからの第２入力データ・ストリームＬ$_1$を表し、「@Ｐ」は、主算術ｐ＿ｏｐによって作られる中間結果ストリームを表し、「@Ｓ」は、メモリへの出力データ・ストリームＳを表す。１実施形態で、Ｌ$_0$ストリーム、Ｌ$_1$ストリーム、およびＳストリームのすべてが、一定のストライドのメモリ動作である。

【００６１】
コンパウンドＣＶＡ動作に関して、ＣＶＡ命令で、上の表１の（ａ）および（ｂ）に示されているように、主算術と副算術の両方を指定し得る。この２つの算術は、ＣＶＡ命令で、それらを区切るコンマを用いて指定され、主算術が先に指定され（この例では、主算術が、３番目のコンマの左に、単語「ａｄｄ」の前に配置される）、副算術がそれに続く（この例では、副算術が、３番目のコンマの右に、単語「ａｄｄ」から始めて配置される）。命令は、「；」記号で終わる。この例では、「＠Ｐ」ストリームが、主算術のデスティネーションとして現れ、副算術にもソースとして現れる。
【００６２】
コンパウンドＣＶＡ動作に関して、ＣＶＡ命令で、上の表１の（ｃ）から（ｆ）のように、主算術を指定するが、副算術を指定しないことが可能である。副算術、この例ではｓ＿ｏｐは、「ノー・オペレーション」であり、主算術によって作られた結果が、「＠Ｓ」ストリームを介してメモリに直接にストアされる。「＠Ｐ」ストリームは、これらの命令で指定されない。
【００６３】
リダクションＣＶＡについて、ＣＶＡ命令では、上の表１の（ｇ）から（ｉ）のように、主算術と副算術の両方が指定される。この例では、「＠Ｐ」ストリームが、主算術のデスティネーションとして現れ、副算術にもソース・オペランドの１つとして現れる。副算術のデスティネーションおよび第２ソース・オペランドは、レジスタＲ３である。リダクションＣＶＡに関して、Ｒ３が、リダクション演算の部分的結果ならびに最終スカラ結果をストアするように指定される。
【００６４】
副算術は可換なので、速記表記を使用して、リダクションＣＶＡを記述し得る。この場合に、副算術式全体が、ｓ＿ｏｐの関数名によって置換される。たとえば、内積を計算するＣＶＡ命令（上の表１の例（ｇ））を、下記のように書き得る。
【００６５】
ＣＶＡ　ｍｕｌ　＠Ｌ０、＠Ｌ１、ａｄｄ；／／リダクションＣＶＡの速記表記
ハイブリッドＣＶＡについて、命令構文は、副算術が２つのデスティネーションすなわちＳストリームおよびレジスタＲ３を有することを除いて、リダクションＣＶＡの構文に類似する。２つのデスティネーションは、ＣＶＡ命令では「｛＠Ｓ、Ｒ３｝」という形で現れる。ハイブリッドＣＶＡの速記表記はない。
【００６６】
図６に、１実施形態による、図４の実行コア４によって３タイプのＣＶＡ（図５に図示）がどのように実行されるかを示す。ストリームL_0が、Ｌ０＿ｄｂｕｓを介して$L_0$６に供給され、ストリームL_1が、Ｌ１＿ｄｂｕｓを介して$L_1$８に供給される。図６では、オペランドＸ、Ｙ、およびＺが、ストリームL_0、ストリームL_1、またはＲＦ２６の指定されたレジスタからソーシングし得る。具体的に言うと、Ｚオペランドが、マルチプレクサ３２、ラッチ３８、マルチプレクサ４１、およびｚｓ＿ｂｕｓを介してＳ３６に供給される。さらに、ｓ＿ｄｂｕｓ、マルチプレクサ４１、およびｚｓ＿ｂｕｓが、リダクションＣＶＡおよびハイブリッドＣＶＡの部分的結果を累算するフィードバック・パスとして使用される。リダクションＣＶＡおよびハイブリッドＣＶＡについて、これらの部分的結果は、常に、ｓ＿ｄｂｕｓおよびｒｅｓｕｌｔ＿ｂｕｓを介してレジスタＲ３に書き戻される。
【００６７】
図７に、本発明の１実施形態による、ＣＶＡ命令のフォーマットを示す。この命令には、イネーブル・フィールドが含まれ、E_0、E_1、およびE_Sが、それぞれL_0ストリーム、L_1ストリーム、およびＳストリームのイネーブル・ビットである。V_{x0}／V_{x1}、V_{y0}／V_{y1}、およびV_{z0}／V_{z1}ビットは、それぞれ、ＣＶＡ命令のオペランドＸ、Ｙ、およびＺがその入力をどのようにソーシングし得るかを指定する。具体的に言うと、これらのビットによって、ストリームL_0、L_1、または指定されたレジスタからの

これらのオペランドのソーシングが選択される。V_{z0}ビット、V_{z1}ビット、およびE_Sビットの組合せによって、実行されるＣＶＡのタイプが定義される。ｐ＿ｏｐフィールドおよびｓ＿ｏｐフィールドによって、それぞれＰ３４およびＳ３６で実行される主算術および副算術のオペコードが指定される。

【００６８】

ＣＶＡ実行に関して、カウント・レジスタ（ＣＲ）５１と称する特殊なレジスタが、ベクトル長の指定に使用される。ＣＲ５１の１実施形態が、図４のＣＩＲ５０の下位半分として図１１に示されている。ＣＩＲ５０の上位半分は、インデックス・レジスタ（ＩＸＲ）７０と称し、ＰＶＡ実行だけに使用される。

【００６９】

１実施形態で、ＣＩＲ５０は、特殊な「ｍｏｖｅ－ｔｏ－ｃｏｎｔｒｏｌ－ｒｅｇｉｓｔｅｒ」命令または「ｍｏｖｅ－ｆｒｏｍ－ｃｏｎｔｒｏｌ－ｒｅｇｉｓｔｅｒ」命令を使用してソフトウェア・プログラムによってアクセスされる特殊な制御レジスタとして実施される。他の実施形態では、ＣＲ５１およびＩＸＲ７０を、２つの別々のレジスタとして実施し得る。

【００７０】

ＣＶＡ実行について、ＣＲ５１を、ＣＶＡ実行の前に適当なベクトル長を用いてソフトウェアによって初期化しなければならない。ＣＶＡ実行中に、処理されるベクトル要素ごとに、ＣＲ５１が、自動的に１つ減分される。ＣＲ５１が０に達し、ベクトル全体が処理されていることが示される時に、ＣＶＡ実行が終了する。計算機は、命令ストリームで、ＣＶＡ命令またはＰＶＡ命令のいずれかである次のベクトル命令に出会うまで、スカラ・モードに入る。

【００７１】

あるスカラｓに関するベクトル演算Ｃ［ｉ］＝ｓＡ［ｉ］＋Ｂ［ｉ］を、次のようにＣＶＡ命令を使用してベクトル化し得る。

【００７２】

【数２】

```
Some initialization code
// assign L0 to vector A; assign L1 to vector B; assign S to vector C
// initialize R5 with the scalar s
CVA    mul @L0, R5, @P,    add @P, @L1, @S;
```

これは、コンパウンドＣＶＡである。この例では、L_0ストリーム、L_1ストリーム、およびＳストリームのすべてがイネーブルされる。ストリームL_0は、ベクトルＡに割り当てられ、ストリームL_1は、ベクトルＢに割り当てられ、ストリームＳは、ベクトルＣに割り当てられる。レジスタＲ５は、ＣＶＡ実行の前に、スカラｓに初期化される。主算術は、「乗算」関数であり、副演算は、「加算」関数である。

【００７３】

ＣＶＡ命令の前に、L_0ストリームをベクトルＡに割り当て、L_1ストリームをベクトルＢに割り当て、Ｓストリームをベクトルに割り当てる初期化コードがある。この「割り当て」は、ある特別に指定されたレジスタを、ベクトルＡ、Ｂ、およびＣにアクセスするための適当な開始ベクトル位置、ストライド値、および要素サイズに初期化することを指す。

【００７４】

次のプログラム・ループの例を検討されたい。

【実施例２】

【００７５】

【数３】

```
L6:
ldw         R10,(R14)
addi        R14,4
mov         R7,R10
lsr         R7,R9
mov         R3,R10
lsl         R3,R8
or          R7,R3
stw         R7,(R13)
addi        R13,4
dec.ne R4
bt          L6
```

このベクトル演算は、次の式によって記述し得る。

Ｃ［ｉ］＝（ｌｓｒ（Ａ［ｉ］、Ｒ９）｜ｌｓｌ（Ａ［ｉ］、Ｒ８））

このループでは、１時に１要素ずつベクトルＡを読み込み、各要素に対して、右論理シフト（ｌｓｒ）演算および左論理シフト（ｌｓｌ）演算を実行する。２つのシフト演算の結果に対して「ｏｒ」関数を実行し、その結果をデスティネーション・ベクトルＣに書き込む。このループは、次のようにベクトル化し得る。

【００７６】

【数４】

```
Some initialization code
// assign L0 to vector A; assign S to a temporary vector
mov R5, R9
CVA    lsr @L0, R5, @S
Some initialization code
// assign L0 to the temporary vector; assign L1 to vector A.
// assign S to vector C.
mov R5, R8
CVA    lsl @L0, R5, @P,    or @L1, @P, @S
```

例２のループは、２つのコンパウンドＣＶＡ命令を使用することによってベクトル化される。第１のＣＶＡ命令は、ソース・ベクトルＡに対して「ｌｓｒ」演算を実行し、一時ベクトルを作る。第２のＣＶＡ命令は、主算術として「ｌｓｌ」、副算術として「ｏｒ」を実行する。後者の命令は、L_0 を介して一時ベクトルを読み取り、これに対して「ｌｓｌ」演算を実行する。この命令は、L_1 を介して元のソース・ベクトルＡを読み取り、主算術から作られた結果との「ｏｒ」関数を実行する。この命令は、その後、Ｓを介して結果をベクトルＣに書き戻す。この実施形態で、両方のＣＶＡ命令のソース・オペランドＹが

、必ずレジスタＲ５からソーシングされることに留意されたい。したがって、Ｒ５を初期化するＣＶＡ実行の前に、追加の「ｍｏｖ」命令が必要である。

【００７７】

ＣＶＡ実行は、ＣＲが０になる前に終了することも可能である。ＣＶＡ命令は、図７に示されているように、E_T ビットおよびC_T ビットを含む早期終了フィールドも有する。E_T ビットは、ＣＶＡ計算の早期終了機能をイネーブルまたはディスエーブルする。C_T ビットは、この機能がイネーブルされる（E_T ＝１）場合に、これを行う条件コードを指定する。次の例に、この機能をベクトル計算についてどのように使用し得るかを示す。この例では、「ｃ」ビットが、条件コードを指す。

【実施例３】

【００７８】

【数５】

```
L1:
addi        R2,2
ld.h        R7,(R2)      // load A[i]
addi        R3,1
ld.b        R6,(R3)      // load B[i]
cmplt  R6,R7 // compare less than: is A[i]>B[i]?
bt          EXIT         // if so, exit the loop
dec.ne R1           // decrement loop index R1
                         // set c bit if r1 not equals zero
bt          L1           // branch to L1 if c bit is set
EXIT
```

このループの対応する高水準ソース・コードを下に示す。

【００７９】

【数６】

```
for (i=0; i<n; i++) {

  if (A[i] > B[i]) {break;}

}
```

このループは、ベクトルＡとＢの間の要素単位の比較を実行する。このループは、あるｉについてＡ［ｉ］＞Ｂ［ｉ］になるや否や終了する。そのような要素の対がない場合には、ループを終了する前に、ベクトルＡとＢのすべての要素が処理されている。

【００８０】

あるプログラム・ループが、固定長ベクトルに対してある算術関数を実行し、ベクトルの最後の要素が処理される前であっても計算が終了し得る場合に、そのような動作を、早期終了を有するベクトル算術と称する。

【００８１】

早期終了を有するベクトル算術には、２つの終了条件すなわち（ｉ）ソース・ベクトルのすべての要素が処理された時、および（ｉｉ）ある算術条件が満たされる時がある。後者の条件は、最後のベクトル要素が処理される前に満たされる可能性があり、通常は、データ依存であり、先験的には判定されない。

【００８２】
例３に示されたループは、早期終了を有するベクトル算術である。このループは、ＣＶＡ構造を使用して下記のようにベクトル化し得る。
【００８３】
【数７】

```
<Some initialization code>
// assign L0 to B, and L1 to A
CVA   cmplt.ct=1   @L0, @L1;
```

このＣＶＡ命令では、$Ｌ_0$ストリームおよび$Ｌ_1$ストリームの両方がイネーブルされるが、Ｓストリームはディスエーブルされる。具体的に言うと、$Ｌ_0$ストリームは、ベクトルＢに割り当てられ、$Ｌ_1$ストリームは、ベクトルＡに割り当てられる。副算術は、指定されない（すなわち、「ノーオペレーション」である）。命令は、$Ｅ_T$＝１および$Ｃ_T$＝１を有する。ＣＶＡ命令の構文「．ｃｔ＝ｘ」によって、$Ｅ_T$＝１であることがアセンブラに指示される。
【００８４】
主算術「ｃｍｐｌｔ」またはｃｏｍｐａｒｅ－ｌｅｓｓ－ｔｈａｎは、$Ｌ_0$ストリームと$Ｌ_1$ストリームの先頭を継続的に比較する。実際には、ベクトルＡとベクトルＢを要素単位で比較している。あるｉについてＢ［ｉ］＜Ａ［ｉ］である場合に、条件コードに１がセットされ、ＣＶＡ実行が終了する。そのような要素対が見つからない場合には、２つのソース・ベクトルを使い果たすまで実行が継続する。
【００８５】
早期終了をイネーブルされた（$Ｅ_T$＝１）ＣＶＡ実行を実行するために、主算術ｐ＿ｏｐ（図４の機能ユニットＰ３４で実行される）は、条件コードを変更し得る算術関数である。ベクトル実行の過程中に、条件コードに事前に指定された値（$Ｃ_T$ビットによって与えられる）がセットされる場合に、ベクトル実行は、即座に終了する。この終了は、ベクトルの全長が処理される前すなわち、ＣＲ５１が０になる前であっても、発生し得る。この実施形態では条件コードが単一のビットであるが、代替実施形態で、条件コードを表す複数のビットおよび／またはエンコーディングを使用し得ることに留意されたい。
【００８６】
ＰＶＡ命令は、伝統的なＤＳＰプロセッサでのＤＯ　ＵＮＴＩＬ命令またはＤＯ　ＷＨＩＬＥ命令によく似ている。そのＤＳＰ対応物と同様に、ＰＶＡ命令を使用して構成されるプログラム・ループは、ＰＶＡ命令と、それに続く複数のスカラ命令から構成されるループ本体からなる。
【００８７】
図８に、ＰＶＡ命令のフォーマットを示す。図９に、ＰＶＡ命令を使用して構成されるプログラム・ループの構造を示す。ＰＶＡモードで実行する時に、実行コア４は、ＰＶＡ命令が、任意選択として、２つまでの入力データ・ストリーム$Ｌ_0$および$Ｌ_1$ならびに出力データ・ストリームＳをイネーブルして、ＰＶＡループ実行中に自動的に実行され得ることを除いて、単一発行パイプライン式計算機のように振る舞う。これらのデータ・ストリームは、一定ストライドのメモリ・ロード／ストア動作である。ＰＶＡ実行について、$Ｌ_0$、$Ｌ_1$、およびＳのいずれかまたはすべてをディスエーブルし得る。
【００８８】
ＣＶＡ実行に似て、入力ストリーム$Ｌ_0$および$Ｌ_1$は、それぞれ、ロード・ユニット$Ｌ_0$６および$Ｌ_1$８を介して、メモリＭ０　１４およびＭ１　１６からＲＦ２６および／またはＰ３４にデータをストリーム・インする。しかし、ＰＶＡ実行では、ストリーム$Ｌ_0$およびストリーム$Ｌ_1$を介してプリフェッチされるデータは、それぞれＲＦ２６のレジスタＲ０およびＲ１に書き込まれる。これらのデータを、Ｐ３４に直接にフィードフォワー

ドすることも可能である。ＰＶＡ実行では、マルチプレクサ３２、ラッチ３８、およびマルチプレクサ４１は、使用されない。

【００８９】

図８を参照すると、ＰＶＡ命令に、E_0、E_1、およびE_Sを有するイネーブル・フィールドが含まれ、これらは、それぞれデータ・ストリームL_0、L_1、およびＳのイネーブル・ビットである。ＰＶＡ命令は、E_TビットおよびC_Tビットを有する早期終了フィールドも有する。ＣＶＡ命令のE_Tビットに似て、E_Tビットは、ＰＶＡ実行の早期終了機能をイネーブルまたはディスエーブルする。C_Tは、この機能がイネーブルされる（E_T＝１）場合に、これが行われる条件コードを指定する。

【００９０】

ＰＶＡ命令のＬｏｏｐ＿ｓｉｚｅフィールドは、スカラ命令の個数単位で、ループ本体のサイズを指定する。ＰＶＡ命令のｃｓ－ｓｔｏｒｅ－ｉｎｄｅｘフィールドは、ループ本体の中で、実行される時に定数ストライド・ストア（ｃｓ－ｓｔｏｒｅ）動作を自動的に開始する命令を指定する。このｃｓ－ｓｔｏｒｅ動作のデータは、この命令によって書き戻されるデータである。１実施形態で、ＰＶＡループの最初の命令のインデックス値が、１であり、２番目の命令のインデックス値が、２であり、以下同様である。ＰＶＡループ内の最後の命令のインデックス値は、Ｌｏｏｐ＿ｓｉｚｅである。この実施形態のインデックス値は、ＰＶＡ命令に関する命令位置である。他の命令インデクシング方式も可能である。

【００９１】

ＰＶＡプログラム・ループでは、条件分岐および無条件分岐が、ループ本体内で許容される。ループ本体の中で分岐が採用され、分岐のターゲットがまだループ本体の中にある場合に、ＰＶＡ実行が継続される。ループ本体の中で分岐が採用され、分岐のターゲットがループ本体の外にある場合に、ＰＶＡ実行が自動的に終了する。

【００９２】

図１１でＣＩＲ５０の下位半分として示されたＣＲ５１も、ＰＶＡ実行に使用される。具体的に言うと、これは、実行されるループ反復の回数を指定するのに使用される。そのＣＶＡ対応物に似て、ＣＲ５１は、その実行の前に、適当な反復カウントを用いてソフトウェアによって初期化されることを必要とする。ＰＶＡ実行中に、ＣＲ５１は、実行される反復ごとに１つだけ自動的に減分される。ＣＲ５１が０に達した時に、ＰＶＡ実行が終了する。計算機は、命令ストリームで次のベクトル命令に出会うまで、スカラ・モードに入る。

【００９３】

ＰＶＡ実行は、３つの機構すなわち（ｉ）ＣＲ５１が０に達する時、（ｉｉ）ＰＶＡ命令のE_TビットおよびC_Tビットが早期終了条件を示す時、または（ｉｉｉ）分岐がループ本体内で採用され、分岐のターゲットがループの外にある時の１つを介して終了し得る。３つの終了条件のすべてが、単一のプログラム・ループ内に共存し得る。すなわち、ループは、実行時条件に基づいて、上の３つの終了機構の１つを介して終了し得る。終了機構（ｉｉ）および（ｉｉｉ）を、集合的に、ＰＶＡ実行の早期終了と称する。

【００９４】

例３に示されたプログラム・ループは、上で説明した早期終了機能（ｉｉ）を使用して、ＰＶＡ構成を使用してベクトル化し得る。

【００９５】

【数８】

```
<Some initialization code>
// assign L0 to B, and L1 to A
PVA     @L0, @L1, ct=1, #1;          // PVA instruction
cmplt   R0, R1;                      // loop body with one instruction
```

記号「＠Ｌ０」および「＠Ｌ１」は、両方ともＰＶＡ命令で指定される。これによって、アセンブラに、$Ｌ_0$ストリームと$Ｌ_1$ストリームの両方がイネーブルされることが示される。Ｓストリームは、ループ本体に「ｃｓ－ｓｔｏｒｅ」ラベルが現れないので、ディスエーブルされる。この命令では、$Ｅ_T$＝１かつ$Ｃ_T$＝１である。$Ｅ_T$＝１をセットすることによって、上で説明した早期終了機能（ｉｉ）がイネーブルされる。ＰＶＡ命令の構文「ｃｔ＝ｘ」によって、アセンブラに、$Ｅ_T$＝１であることが指示される。

【００９６】

ＰＶＡプログラム・ループのサイズは、ＰＶＡ命令の「＃１」表記によって指定されるように、１命令である。このループには、単一のスカラ命令「ｃｍｐｌｔ」が含まれ、この命令は、レジスタＲ０およびＲ１から継続的に読み取り、この２つの値を比較する。

【００９７】

ＰＶＡ実行について、ループ本体内でＲ０（またはＲ１）から読み取ることによって、データ項目が$Ｌ_0$（または$Ｌ_1$）入力ストリームから自動的にデキューされる。$Ｌ_0$（または$Ｌ_1$）からのデータ項目のデキューによって、デキュー動作によって残された空のスロットを埋めるために、定数ストライド・ロード（ｃｓ－ｌｏａｄ）動作が自動的に実行される。したがって、ループ本体でレジスタＲ０（またはＲ１）から継続的に読み取ることによって、Ｍ０　１４（またはＭ１　１６）からのデータの連続的ストリームのフェッチおよびレジスタＲ０（またはＲ１）へのロードがトリガされる。Ｒ０（またはＲ１）は、単一のループ反復で複数回読み取り得る。

【００９８】

ＰＶＡ実行について、$Ｌ_0$（または$Ｌ_1$）ストリームがイネーブルされる時に、Ｒ０（またはＲ１）が、読取専用レジスタになり、このレジスタへの書込は、ハードウェアによって無視される。

【００９９】

実質的に、上のループ本体の「ｃｍｐｌｔ」命令は、ベクトルＡおよびＢを要素単位で比較する。この例では、Ｒ０の内容がＲ１の内容より小さい（すなわちＢ［ｉ］＜Ａ［ｉ］である）場合に、条件コードに１がセットされ、事前に指定された値$Ｃ_T$＝１と等しくなり、ＰＶＡループ実行が即座に終了する（$Ｅ_T$＝１なので）。この終了は、ＣＲ５１が０に達する前であっても発生し得る。

【０１００】

代替案では、例３に示されたプログラム・ループを、下のようにＰＶＡ構成を使用してベクトル化することも可能である。この例では、前に説明した早期終了機構（ｉｉｉ）を使用して、どのようにしてＰＶＡループを終了し得るかを示す。

【０１０１】

【数９】

```
<Some initialization code>
// assign L0 to B, and L1 to A
PVA     @L0, @L1, #2;        // PVA instruction
cmplt   R0, R1;              // part of PVA loop
bt EXIT                      // part of PVA loop
EXIT
```

この代替案では、ＰＶＡ命令でE_T＝０である。ループ本体には、２つの命令「ｃｍｐｌｔ」および「ｂｔ」がある。第１の命令が、ベクトルＡとＢの間の比較動作の結果として条件コードをセットする場合に、第２の命令（「ｂｔ　ＥＸＩＴ」）が採用される。この分岐のターゲットは、ループ本体の外にあるので、ＰＶＡ実行が即座に終了する。そうではなく、条件コードがセットされず、分岐命令「ｂｔ」が採用されない場合に、実行は、次の反復の第１命令（この場合では「ｃｍｐｌｔ」命令）から継続される。

【０１０２】

前に示したＰＶＡベクトル化されたループの第１版（終了機構（ｉｉ）を使用する）は、上で示したＰＶＡベクトル化されたループの第２版（終了機構（ｉｉｉ）を使用する）より効率的である。前者は、ループ本体の中に１つのスカラ命令だけを有するが、後者は、ループ本体の中に２つのスカラ命令を有する。

【０１０３】

「ｅｘｉｔ－ｂｙ－ｃｏｎｄｉｔｉｏｎａｌ－ｂｒａｎｃｈ（条件分岐による終了）」手法が、通常、条件コードも使用するループ本体内で条件実行と共にプログラム・ループによって使用される。

【０１０４】

例３に示された元のプログラム・ループでは、反復ごとに８つの命令が、ループ実行中に実行コアによって要求される。ＰＶＡ構成を使用してループをベクトル化した後には、反復ごとに２つから３つの命令だけが、ループ実行中に実行コアによって要求される。

【０１０５】

代替案では、早期終了をもたらすために、ＰＶＡ命令によって、図２３に示されているように、ループを早期に終了し得る条件設定命令のオフセットを指定し得る。図２３のＰＶＡ命令を使用して構成されるプログラム・ループの構造を、図２４に示す。図２３では、非０のC_{offset}フィールドによって、早期終了機能がイネーブルされることが示される（図８のフォーマットでE_T＝１をセットすることに似る）。このフィールドによって、条件設定命令のオフセット（ＰＶＡ命令からの命令数単位）が指定される。条件コード（またはｃビット）に、条件設定命令の実行の結果として、ＰＶＡ命令のC_Tフィールドによって指定される値がセットされた時に、ＰＶＡ実行が即座に終了する。下に例を示す。

【０１０６】

【数１０】

```
Some initialization code // assign A[i] to L0; assign B[i] to L1.
PVA  @L0, @L1, #5, CT=1, Coffset=4
cmplt R0, R1              //part of loop body

bt SKIP                   //part of loop body
sub R3, R4                //part of loop body
SKIP
cmpnei R3, #2             //part of loop body, check R3
add R5, R3                //part of loop body
EXIT
```

この代替案では、ＰＶＡ命令のＣ$_{offset}$＝４である。ループ本体には５つの命令（「ｃｍｐｌｔ」、「ｂｔ」、「ｓｕｂ」、「ｃｍｐｎｅｉ」、および「ａｄｄ」）がある。ｃｍｐｌｔ命令は、Ｒ０およびＲ１の値を比較し、条件コード（またはｃビット）をセットする。「ｂｔ」命令は、結果のｃビットの値を検査し、セットされている場合に、「ｓｕｂ」命令を迂回して分岐する。ループの４番目の命令は、比較命令であり、これは、減算命令の結果（レジスタＲ３内）を比較して、それが値「２」と等しくないかどうかを調べる。比較の結果が真である場合には、ｃビットに「１」がセットされ、ＰＶＡ命令のＣ$_{T}$フィールドにも「１」がセットされているので、ＰＶＡ実行は即座に終了する。そうでない場合には、実行は、次の命令「ａｄｄ」から継続される。ａｄｄ命令に続いて、実行は、次のループ反復（「ｃｍｐｌｔ」）で継続される。ＰＶＡ命令のＣ$_{offset}$フィールドによって、ループ終了の制御に「ｃｍｐｎｅｉ」命令（ＰＶＡ命令に関してループ内で４のオフセット）が使用されることと、ループ終了に影響せずに、ループ内の他の命令（この場合には「ｃｍｐｌｔ」命令）によってｃビットが影響を受ける可能性があることが示されることに留意されたい。この機能は、ループ終了に影響せずにループ内で複数の条件をテストしなければならない時に有用である。

【０１０７】

下の例を使用して、Ｓストリームを介してｃｓ－ｓｔｏｒｅ動作をＰＶＡ実行についてどのように実行し得るかを示す。

【実施例４】

【０１０８】

【数１１】

```
        mov     R4,4
        mov     R8,8
        mov     R6,16
L1:
        ld.w    R7,(R3)     // load A[i]
        addi    R3,4
        mov     R2, R8
        cmplt   R6,R7       // is A[i] > 16?
        movt    R2, R4      // conditional move:
                            // R2 = (A[i]>16)? R4 : R8;
        st.w    R2,(R10)    // store result to C[i]
        addi    R10, 4
        dec.ne  R5          // decrement loop index R5
                            // set c bit if R5 not equals zero
        bt      L1          // branch to L1 if c bit is set
```

上のループの高水準ソース・コードを下に示す。

【０１０９】

【数１２】

```
for (i=0; i<n; i++) {
        if (A[i] > 16) C[i] =4; else C[i] = 8;
}
```

このループは、ＰＶＡ構成を使用して、下記のようにベクトル化し得る。

【０１１０】

【数１３】

```
<Some initialization code>
// assign L0 to vector A; assign S to vector C.
mov     R4, 4
mov     R8, 8
mov     R6, 16
PVA     @L0, @S, #3;     // PVA instruction
mov     R2, R8
cmplt   R6, R0           // is A[i] > 16?
cs-store:
movt    R2, R4           // conditional move:
                         // R2 = (A[i]>16)? R4:R8;
                         // cs-store performed here
```

このループでは、L_0ストリームおよびＳストリームがイネーブルされるが、L_1ストリームはイネーブルされない。ＰＶＡループ・サイズは、３命令である。各反復で、「ｃｍｐｌｔ」命令は、Ｒ０（または要素Ａ［ｉ］）を読み取り、Ｒ６にストアされた定数値１６と比較する。「ｍｏｖｔ」命令は、ｃビット（または条件コード）がセットされている場合に、Ｒ４をＲ２へ条件付きで移動する。そうでない場合に、「ｍｏｖｔ」命令は、Ｒ２をＲ２に移動し、値は変化しない。

【０１１１】

「ｍｏｖｔ」命令は、ループ本体の中の「ｃｓ－ｓｔｏｒｅ」ラベルに配置されている。Ｓストリームを介するｃｓ－ｓｔｏｒｅ動作は、この「ｍｏｖｔ」命令が実行される時に、必ず自動的に開始される。このｃｓ－ｓｔｏｒｅ動作に使用されるデータは、「ｍｏｖｔ」命令によって書き戻されるものと同一のデータである。このＰＶＡループを実行する時に、「ｍｏｖｔ」命令によって作られる結果が、Ｐ３４によってｐ＿ｂｕｓに駆動され、マルチプレクサ４０およびｒｅｓｕｌｔ＿ｂｕｓを介してＲＦ２６に常に書き戻される。Ｓ３６は、ｐ＿ｂｕｓでこの結果を取り込み、これらの取り込まれたデータを使用してメモリ・システムへのｃｓ－ｓｔｏｒｅ動作を実行する。

【０１１２】

この例では、「ｍｏｖｔ」命令に関連するストア動作が、アセンブリ・コードで、ループ本体の中のプログラム・ラベル「ｃｓ－ｓｔｏｒｅ」を使用して指定される。しかし、機械コード（またはアセンブルされたバイナリ・コード）では、そのようなストア動作は、「ｍｏｖｔ」命令にエンコードされない。そうではなく、ストア動作は、ループを構成するのに使用されるＰＶＡ命令で暗黙のうちに指定される。この場合では、ＰＶＡ命令が、E_S＝１にされ、その「ｃｓ－ｓｔｏｒｅ－ｉｎｄｅｘ」フィールドが、「ｍｏｖｔ」命令をポイントしている。このタイプのストア動作を、「暗黙ストア」動作と称する。このストア動作をトリガし得るループ本体内の命令（この例では「ｍｏｖｔ」）を、暗黙ストア命令と称する。

【０１１３】

上の実施形態では、ＰＶＡ命令内に「ｃｓ－ｓｔｏｒｅ－ｉｎｄｅｘ」フィールドが１つだけあり、したがって、１つの「ｃｓ－ｓｔｏｒｅ」ラベルだけが、ループ本体の中で許容される。その結果、１つの暗黙ストア動作だけが、各反復で許容される。代替実施形態では、複数の暗黙ストア動作を、特殊なマスク・レジスタを使用することによって実行

し得る。そのような代替実施形態では、マスク・レジスタを１６ビット・レジスタとすることが可能であり、このレジスタの各ビットは、ループ本体内のスカラ命令に対応する。このレジスタの各ビットによって、命令内の対応するスカラ命令に関するｃｓ－ｓｔｏｒｅ動作がイネーブルまたはディスエーブルされる。したがって、ループ本体の１６個までの命令によって、各反復でｃｓ－ｓｔｏｒｅ動作を開始し得る。ソフトウェアは、ＰＶＡ実行の前にこのマスク・レジスタを初期化する。ＰＶＡ実行中に、命令が結果を書き戻し、それに対応するマスク・レジスタ内のビットをセットする時に、その命令によって書き戻されたデータを使用して、ｃｓ－ｓｔｏｒｅ動作が開始される。１６個のｃｓ－ｓｔｏｒｅデータ・アドレス、ストライド、およびオペランド・サイズを指定するために、ハードウェア・サポートが必要である。

【０１１４】

暗黙ストア動作を使用するもう１つの実施形態では、ｃｓ－ｓｔｏｒｅ－ｉｎｄｅｘではなく、ｃｓ－ｓｔｏｒｅ－ｒｅｇ－ｄｅｓｉｇｎａｔｏｒ（ｃｓ－ｓｔｏｒｅレジスタ指定子）フィールド（レジスタ・フィールドとも称する）をＰＶＡ命令で指定し得る。たとえば、図２５に、ｃｓ－ｓｔｏｒｅ－ｒｅｇ－ｄｅｓｉｇｎａｔｏｒフィールドを有するＰＶＡ命令の１実施形態を示し、図２６に、図２５のＰＶＡ命令フォーマットを使用して構成されるプログラム・ループを示す。このｃｓ－ｓｔｏｒｅ－ｒｅｇ－ｄｅｓｉｇｎａｔｏｒフィールドによって、ループ本体の内部の命令のデスティネーション・レジスタの関数としての暗黙ストア動作が定義される。たとえば、ｃｓ－ｓｔｏｒｅ－ｒｅｇ－ｄｅｓｉｇｎａｔｏｒフィールドによってレジスタ３が示される場合に、デスティネーションとしてレジスタ３を有するすべての命令が、暗黙ストアを呼び出す。例の命令が、「ａｄｄ　Ｒ３、Ｒ５」であり、これは、「命令デスティネーション、ソース」というフォーマットを有する。このａｄｄ命令では、レジスタ３がデスティネーション・レジスタとして使用され、したがって、このａｄｄ命令は、暗黙ストア動作に対応する。ループ本体の中の複数の命令が、デスティネーション・レジスタとしてレジスタ３を使用する場合には、複数の命令が、暗黙ストア動作を開始し得る。しかし、「ｃｍｐｎｅｉ　Ｒ３、＃２」などの命令は、この例ではＲ３がデスティネーション・レジスタとして使用されないので、暗黙ストア動作に対応し得ない。「ｃｍｐｎｅｉ」命令（ならびに他の命令）は、デスティネーション・レジスタに書き込まれる結果値を生成せず、このフォーマットは、この例ではソース値の対を指定している。さらに、代替実施形態では、ＰＶＡ命令で複数のレジスタを定義することを可能にして、より多くの暗黙ストアを可能にし得る。

【０１１５】

もう１つの実施形態で、ＰＶＡ命令に、継続機能を組み込むことも可能である。この機能を有するＰＶＡ命令のフォーマットを、図２０に示す。このＰＶＡ命令には、早期継続フィールド内の２つの追加ビットすなわちE_CビットおよびC_Cビットがある。E_Cビットによって、早期継続機能がイネーブルまたはディスエーブルされ、C_Cビットによって、この機能がイネーブルされる場合に、この早期継続が行われる条件コードの値が指定される。

【０１１６】

次の高水準コードに、プログラム・ループの残りの実行をスキップするのにこの早期継続機能をどのように使用し得るかを示す。

【０１１７】

【数１４】

```
for (i=0; i<n; i++) {
  if (A[i] < s) continue;
  <more code>
}
```

この例では、ベクトルＡの要素が、スカラｓと比較される。あるｉについてＡ［ｉ］＜ｓである場合に、上で「＜さらなるコード＞」と示されたプログラム・ループの残りの部分が、スキップされ、実行は、次の反復の先頭から継続される。これは、事前に指定された条件が満たされる時にループの実行が即座に終了する、前に説明した「早期終了機能」と対照的である。

【０１１８】

このループを、次のようにベクトル化し得る。

【０１１９】

【数１５】

```
<Some initialization code>
// assign L0 to A; initialize R1 with scalar s.

PVA     @L0, cc=1, #Loop_size;     // PVA instruction
cmplt   R0, R1;                    // A[i] < s?

<more code>                        // The rest of the loop body
```

このＰＶＡ命令では、E_C＝１かつC_C＝１である。ＰＶＡ命令の構文「ｃｃ＝ｘ」によって、E_C＝１であることがアセンブラに指示される。「ｃｍｐｌｔ」命令によって条件コードに１がセットされる時に、必ず、そのＡ［ｉ］とｓの間の比較動作の結果として、実行が、即座に次の反復から継続され、「＜さらなるコード＞」の実行がスキップされる。

【０１２０】

早期終了機能および早期継続機能の両方がイネーブルされた（E_T＝E_C＝１）ＰＶＡ命令では、ループ本体内の命令が、条件コードを変更することが可能であり、これによって、現在の反復の実行が終わる。条件コードに、C_Tビットによって指定される値がセットされている場合には、ループ実行が、完全に終了する。条件コードに、C_Cビットによって指定される値がセットされている場合には、ループ実行は、次の反復から継続される。条件コードに、C_TビットおよびC_Cビットの両方によって指定される値がセットされている（両方が条件コードについて同一の値を指定する）場合には、ループ実行が終了する。すなわち、この実施形態では、早期終了機能が、早期継続機能の地位を奪う。他の実施形態では、早期継続機能が、早期終了機能の地位を奪う。図示されていないが、早期継続機能を、早期終了について図２３に示したものに似た形で設けることも可能である。$C_{offset-c}$フィールドおよびC_Cビットを命令フォーマットに加えることが可能であり、あるいは、C_{offset}指定子およびC_T指定子と置換し得る。$C_{offset-c}$フィールドおよびC_Cビットによって、条件設定命令のＰＶＡ命令からの命令単位のオフセットと、指定された命令がＰＶＡ命令でセットされたC_Cビットと一致する条件コードを生成する時にループの早期継続をもたらすのに使用される条件値を指定し得る。

【０１２１】

ＰＶＡ構成は、一般に、（ｉ）ループ制御機構、（ｉｉ）ｃｓ－ｌｏａｄを介する入力データ・ストリーム化、および（ｉｉｉ）ｃｓ－ｓｔｏｒｅを介する出力データ・ストリーム化に関連するある種の実行オーバーヘッドを除去し得る。ＰＶＡ命令に、システムの必要に応じて、上で説明した諸フィールドの組合せを、適当に含めることも可能である。たとえば、代替実施形態で、暗黙ストア動作を実行する、上で説明したものと異なるフィールドおよび方法を使用し得る。

【０１２２】

図１０に、データ・ストリームL_0、L_1、およびＳのストライド値およびオペランド

・サイズを指定するＳＳＲ４２の１実施形態を示す。このレジスタは、ＣＶＡ実行とＰＶＡ実行の両方に使用される。この実施形態では、レジスタが、３つのデータ・ストリーム$Ｌ_0$、$Ｌ_1$、およびＳのそれぞれに１つの３つの部分に区分される。ストライド・フィールド$ＳＴＲ_0$、$ＳＴＲ_1$、および$ＳＴＲ_S$は、９ビット幅である。サイズ・フィールド$ＳＺ_0$、$ＳＺ_1$、および$ＳＺ_S$は、２ビット幅であり、それぞれによってバイト、ハーフワード、またはフルワードが指定される。

【０１２３】

２つのベクトルの内積、$Σ_i$（Ａ［ｉ］＊Ｂ［ｉ］）を実施する下記の例を検討されたい。この動作は、ＣＶＡ命令を使用して下記のように実行し得る。この例では、ＳＳＲ［$ＳＴＲ_0$］が、ＳＳＲレジスタの$ＳＴＲ_0$フィールドを表し、他も同様である。

【０１２４】

ＣＲをベクトル長に初期化する。

ＳＳＲ［$ＳＴＲ_0$］をベクトルＡのストライド値に初期化する。

ＳＳＲ［$ＳＴＲ_1$］をベクトルＢのストライド値に初期化する。

【０１２５】

Ｒ０をベクトルＡの開始アドレスに初期化する。

Ｒ１をベクトルＢの開始アドレスに初期化する。

ＣＶＡ　ｍｕｌ　@Ｌ０、@Ｌ１、ａｄｄ；

この例では、$Ｌ_0$および$Ｌ_1$がイネーブルされ、Ｓがディスエーブルされ、したがって、これはリダクションＣＶＡである。このリダクション演算の主算術は、「ｍｕｌ」関数であり、副算術は、「ａｄｄ」関数である。ＣＶＡ実行中に、部分的結果が、継続的にレジスタＲ３に書き戻される。ベクトル計算が完了した時に、最終結果すなわち内積が、Ｒ３に暗黙ストアされる。この例のＣＶＡは、$Ｖ_{x1}$／$Ｖ_{x0}$＝０１、$Ｖ_{y1}$／$Ｖ_{y0}$＝１０、$Ｖ_{z1}$＝$Ｖ_{z0}$＝１１、$Ｅ_S$＝０、$Ｅ_T$＝０、$Ｃ_T$＝「ドント・ケア」という設定を有する。

【０１２６】

この例では、ベクトル実行が割り込まれる場合に、レジスタＲ３に、内積計算の中間の部分的結果が含まれる。割込みからリターンする時に、計算が、中断されたところから継続され、最初の部分的結果としてレジスタＲ３の内容が使用される。

【０１２７】

もう１つの例で、すべてのｉに関するベクトル算術Ｃ［ｉ］＝ｓＡ［ｉ］を実施する。

ＣＲをベクトル長に初期化する。

ＳＳＲ［$ＳＴＲ_0$］をベクトルＡのストライド値に初期化する。

【０１２８】

ＳＳＲ［$ＳＴＲ_S$］をベクトルＣのストライド値に初期化する。

Ｒ０をベクトルＡの開始アドレスに初期化する。

Ｒ２をベクトルＣの開始アドレスに初期化する。

【０１２９】

Ｒ５をｓに初期化する。

ＣＶＡ　ｍｕｌ　@Ｌ０、Ｒ５、@Ｓ；

これはコンパウンドＣＶＡである。第２算術は「ノー・オペレーション」である。この場合に、$Ｌ_0$ストリームおよびＳストリームは、イネーブルされるが、$Ｌ_1$ストリームはイネーブルされない。ベクトル実行の前に、Ｒ５が、スカラ定数ｓによって初期化される。この例のＣＶＡ命令は、$Ｖ_{x1}$／$Ｖ_{x0}$＝０１、$Ｖ_{y1}$／$Ｖ_{y0}$＝００、$Ｖ_{z0}$／$Ｖ_{z1}$＝００、$Ｅ_S$＝１、$Ｅ_T$＝０、$Ｃ_T$＝「ドント・ケア」という設定を有する。

【０１３０】

図１２から１４に、$Ｌ_0$６、$Ｌ_1$８、およびＳ３６の１つの可能な実施形態を示す。$Ｌ_0$６、$Ｌ_1$８、およびＳ３６のそれぞれが、それぞれレジスタＲ０、Ｒ１、およびＲ２のローカル・コピーを有する。このローカル・コピーを、Ｌ＿Ｒ０　１１４、Ｌ＿Ｒ１　１２４、およびＬ＿Ｒ２　１３６と表す。

【０１３１】
スカラ実行中に、$L_0$６（または$L_1$８）ユニットは、レジスタＲ０（またはＲ１）への書き戻しアクティビティについてｒｅｓｕｌｔ＿ｂｕｓを常時スヌープし、レジスタのローカル・コピーＬ＿Ｒ０　１１４（またはＬ＿Ｒ１　１２４）を更新する。同様に、Ｓ３６は、レジスタＲ２への書き戻しアクティビティについてｐ＿ｂｕｓを常時スヌープし、レジスタのローカル・コピーＬ＿Ｒ２　１３６を更新する（図１４参照）。
【０１３２】
ＣＶＡ実行中またはＰＶＡ実行中に、これらのローカル・レジスタの内容が、メモリ・ブロックのロード・アドレスまたはストア・アドレスとして適当に使用される。これらのレジスタは、対応するデータ・ストリームがイネーブルされている時に、適当なストライド値を加算することによって、ハードウェアによって常時更新される。すなわち、Ｌ＿Ｒ０　１１４、Ｌ＿Ｒ１　１２４、およびＬ＿Ｒ２　１３６は、それぞれ、実行されるｃｓ－ｌｏａｄ動作またはｃｓ－ｓｔｏｒｅ動作のそれぞれについて、ＳＳＲ［STR_0］、ＳＳＲ［STR_1］、およびＳＳＲ［STR_S］によって指定される量と合計される。
【０１３３】
ベクトル命令（ＣＶＡ命令またはＰＶＡ命令）が、そのE_0（またはE_1）ビットをセットされて最初にデコードされる時に、第１ロード・アドレスとしてＬ＿Ｒ０　１１４（またはＬ＿Ｒ１　１２４）を使用して、ｃｓ－ｌｏａｄが即座に開始される。その後、メモリからプリフェッチされるデータごとに、Ｌ＿Ｒ０　１１４（またはＬ＿Ｒ１　１２４）が更新される。
【０１３４】
図１２および１３に示された実施形態では、ロード・ユニット$L_0$６および$L_1$８内のデータ・キューが、２つだけの深さである。$L_0$６への入力に、Ｍ０＿ｄｂｕｓおよびＴＭ＿ｄｂｕｓが含まれる。この２つの入力が、マルチプレクサ１１０に供給される。マルチプレクサ１１０の出力は、データ・キューの末尾１１１に供給される。データ・キューの末尾１１１は、マルチプレクサ１１３およびデータ・キューの第２エントリ１１２にデータを供給する。この第２エントリ１１２も、マルチプレクサ１１３にデータを供給する。データは、マルチプレクサ１１３からＬ０＿ｄｂｕｓに供給される。
【０１３５】
ロード・ユニット$L_0$６内で、Ｍ０＿ｄｂｕｓおよびＴＭ＿ｄｂｕｓからのデータを、マルチプレクサ１１０を介し、末尾エントリ１１１およびマルチプレクサ１１３を介し、第２データエントリ１１２をバイパスして、Ｌ０＿ｄｂｕｓに供給し得る。データを、第２エントリ１１２からマルチプレクサ１１３を介してＬ０＿ｄｂｕｓに供給することも可能である。
【０１３６】
図１２を継続すると、加算器１１６が、ＳＳＲ４２、ＳＳＲ［STR_0］、およびロード・アドレスをＬ＿Ｒ０　１１４から受け取る。加算器１１６の出力は、マルチプレクサ１１５に結合される。ｒｅｓｕｌｔ＿ｂｕｓも、マルチプレクサ１１５に結合される。マルチプレクサ１１５の出力は、Ｌ＿Ｒ０　１１４に結合され、Ｌ＿Ｒ０　１１４は、Ｌ０＿ａｂｕｓに結合される。ベクトル実行中に、ＳＳＲ［STR_0］によって指定される量が、ストリームL_0を介して実行されるｃｓ－ｌｏａｄ動作ごとに、Ｌ＿Ｒ０　１１４に加算される。マルチプレクサ１１５およびＬ＿Ｒ０　１１４を用いると、ロード・ユニット$L_0$６が、レジスタＲ０へのすべての書き戻しについてｒｅｓｕｌｔ＿ｂｕｓをスヌープし得るようになる。スカラ実行中には、Ｌ＿Ｒ０　１１４が、ＲＦ２６のレジスタ０と同一の値に維持される。同様の動作および接続性が、図１３に示されたロード・ユニット$L_1$８にもあてはまる。
【０１３７】
図１４を参照すると、ストア・ユニットＳ３６内で、ｐ＿ｂｕｓによって、データがラッチ１３０に供給される。このラッチおよびｚｓ＿ｂｕｓの出力によって、ＡＬＵ１３１にデータが供給される。ＡＬＵ１３１は、ラッチ１３２にデータを供給し、ラッチ１３２

は、マルチプレクサ１３３にデータを供給する。マルチプレクサ１３３は、ラッチ１３０およびラッチ１３２からデータをとり、ｓ＿ｄｂｕｓにデータを供給する。

【０１３８】

データを、ｐ＿ｂｕｓから、ラッチ１３０およびマルチプレクサ１３３を介してｓ＿ｄｂｕｓに供給することも可能である。このパスによって、ストア・ユニットＳ３６内のＡＬＵ１３１の計算がバイパスされる。データを、ＡＬＵ１３１の出力からラッチ１３２およびマルチプレクサ１３３を介してｓ＿ｄｂｕｓに供給することも可能である。このパスでは、ＡＬＵ１３１の結果が、受け取られ、ｓ＿ｄｂｕｓに渡される。

【０１３９】

ベクトル算術では、メモリから定数ストライド・ロードを介してロードされるほとんどのデータが、単一の反復で消費され、絶対に再利用されないという意味で、一時的でしかない。ベクトル算術への割込みが許容される場合に、ベクトル実行に関連するすべての定数ストライド・ロード／ストア・アドレスを保存して、割込みからのリターンの後にロード／ストア動作を再開し得るようにする必要がある。

【０１４０】

メモリからプリフェッチされるすべての一時的なものならびにロード／ストア・アドレスを、汎用レジスタ・ファイルまたは制御レジスタなどのアーキテクチャ的に可視の記憶空間を使用してストアすることは、これらの貴重なリソースの非効率的な使用になる可能性がある。本発明は、レジスタ・オーバーレイを使用して、この問題に対処する。この手法では、ＰＶＡ実行モードに入る時に、アーキテクチャ的に可視のレジスタ・ファイルの一部が、レジスタの新しい組によって「オーバーレイ」される。レジスタは、オーバーレイされる時に、（ｉ）オーバーレイされたインスタンスおよび（ｉｉ）一時的インスタンスという２つのインスタンスを有する。ＰＶＡモードでは、その一時的インスタンスだけが、プログラマに可視であり、オーバーレイされたインスタンスは不可視である。逆に、実行が、ＰＶＡモードを出てスカラ・モードに入る時に、オーバーレイされたインスタンスが、再び可視になり、一時的インスタンスは、存在しなくなる。

【０１４１】

図１５に、３つの実行モードすなわちスカラ、ＰＶＡ、およびスカラのシーケンスにまたがる、オーバーレイされたレジスタの可視性を示す。１実施形態で、レジスタＲ０およびＲ１が、ＰＶＡ実行中にオーバーレイされ得るレジスタの組として指定される。これを、下の表２に示す。これらのレジスタのオーバーレイされたインスタンスは、対応するｃｓ－ｌｏａｄロード・アドレスをストアするのに使用される。これらのレジスタの一時的インスタンスは、ｃｓ－ｌｏａｄ　L_0 および L_1 を介してメモリからプリフェッチされるデータをストアするのに使用される。

【０１４２】

【表２】

表２．レジスタの割振りとアクセス可能性

レジスタ	オーバーレイされるインスタンス (スカラ・モードでのみ可視)	一時的インスタンス (PVA モードでのみ可視)	
	内容	内容	動作
R0	L_0 のロード・アドレス	L_0 のプリフェッチされたデータ	読取専用
R1	L_1 のロード・アドレス	L_1 のプリフェッチされたデータ	読取専用

Ｒ０およびＲ１の一時的インスタンスを、それぞれＴ＿Ｒ０およびＴ＿Ｒ１と表す。Ｒ０およびＲ１のオーバーレイされるインスタンスを、それぞれＯ＿Ｒ０およびＯ＿Ｒ１と表す。

【０１４３】

Ｔ＿Ｒ０およびＴ＿Ｒ１は、読取専用レジスタである。ループ本体内でのこの一時レジ

スタへの書込は、ハードウェアによって無視される。これらの一時レジスタは、ＰＶＡ実行中に限って定義される。ＰＶＡループが、その実行を終了する時に、これらの一時レジスタに含まれるデータは、失われる。その時点でのそのようなレジスタへのアクセスによって、レジスタのオーバーレイされたインスタンスが検索され、これは、最後に実行されたｃｓ－ｌｏａｄ動作のロード・アドレスである。

【０１４４】

また、ＰＶＡ実行が割り込まれる時に、Ｔ＿Ｒ０およびＴ＿Ｒ１は、コンテキストの一部として保存されない。割込みからリターンする時に、データをＴ＿Ｒ０およびＴ＿Ｒ１にプリフェッチしたｃｓ－ｌｏａｄ動作が、Ｏ＿Ｒ０およびＯ＿Ｒ１にストアされたロード・アドレスを使用して再初期化される。レジスタＴ＿Ｒ０およびＴ＿Ｒ１は、普通のＰＶＡ実行を再開し得るようになる前に再初期化される。

【０１４５】

図１６に、図４のレジスタ・ファイルＲＦの１実施形態を示す。ＲＦ２６には、他のレジスタの中でも、Ｏ＿Ｒ０　１４２、Ｔ＿Ｒ０　１４３、Ｏ＿Ｒ１　１５２、およびＴ＿Ｒ１　１５３が含まれる。ＲＦ２６内では、レジスタＯ＿Ｒ０　１４２、Ｏ＿Ｒ１　１５２、およびＲ２　１６２が、それぞれ加算器１４０、１５０、および１６０を使用して更新される。

【０１４６】

レジスタＯ＿Ｒ０　１４２の値が、マルチプレクサ１４４に供給される。この値は、加算器１４０の入力にもフィードバックされる。加算器１４０のもう１つの入力は、ＳＳＲ［STR_0］に結合される。加算器１４０の出力は、入力としてマルチプレクサ１４１に供給される。マルチプレクサ１４１の出力は、Ｏ＿Ｒ０　１４２の入力に結合される。マルチプレクサ１６４が、もう１つの入力としてマルチプレクサ１４１に供給される。Ｔ＿Ｒ０　１４３は、その入力をＬ０＿ｄｂｕｓから取り、その出力をマルチプレクサ１４４のもう１つの入力に供給する。マルチプレクサ１４４の出力は、ＲＦ２６の出力Ｒ０として供給される。加算器１４０、マルチプレクサ１４１、およびＯ＿Ｒ０　１４２を使用することによって、Ｏ＿Ｒ０　１４２を、各サイクルにＳＳＲ［STR_0］によって指定される量だけ増分し得る。

【０１４７】

同一の機構を使用して、各サイクルに、Ｏ＿Ｒ１　１５２およびＲ２　１６２を、それぞれストライド値ＳＳＲ［STR_1］およびＳＳＲ［STR_S］を加算することによって、類似するハードウェア構成によって更新し得る。

【０１４８】

ＰＶＡ実行中に、Ｏ＿Ｒ０　１４２およびＴ＿Ｒ０　１４３の更新が、対応するｃｓ－ｌｏａｄ動作がＲＦ２６にコミットされるのと同時に行われる。同様に、Ｏ＿Ｒ１　１５２およびＴ＿Ｒ１　１５３の更新は、対応するｃｓ－ｌｏａｄ動作がＲＦ２６にコミットされるのと同時に行われる。Ｒ２　１６２の更新は、対応するｃｓ－ｓｔｏｒｅがメモリにコミットされるのと同時に行われる。任意の所与の時点で、Ｔ＿Ｒ０　１４３（またはＴ＿Ｒ１　１５３）に、メモリからプリフェッチされたデータが含まれ、アドレスは、Ｏ＿Ｒ０　１４２（またはＯ＿Ｒ１　１５２）にストアされる。この一貫性は、ＰＶＡ実行中の命令境界にまたがって維持される。一時レジスタＴ＿Ｒ０　１４４およびＴ＿Ｒ１　１５４が、ＣＶＡ実行またはスカラ実行に不要であることに留意されたい。

【０１４９】

ＰＶＡ実行中の一時インスタンスおよびオーバーレイされるインスタンスの一貫性を、図１７に示す。５命令の実行シーケンスが示されている。水平軸は、時間を表し、レジスタＲ０の一時的インスタンスおよびオーバーレイされるインスタンスが、それぞれ垂直軸に表されている。命令境界は、垂直の破線によって示されている。この例では、ストリームL_0を介してストリーミングされるデータが、Ｄ０、Ｄ１、Ｄ２、Ｄ３、．．．であり、Ａ０、Ａ１、Ａ２、Ａ３、．．．が、対応するロード・アドレスである。この５つの命令の中で、Ｉ０、Ｉ２、およびＩ３が、レジスタＲ０を読み取る命令である。Ｒ０からの

読み取りは、図１７に示されているように、次の命令境界の先頭での新しい「アドレス－データ」対によるＯ＿Ｒ０　１４２およびＴ＿Ｒ０　１４３の同時更新を引き起こす。命令Ｉ１は、Ｒ０を読み取らない命令であり、したがって、Ｏ＿Ｒ０　１４２およびＴ＿Ｒ０　１４３は、次の命令境界で更新されない。レジスタＯ＿Ｒ０　１４２およびＴ＿Ｒ０　１４３は、レジスタＲ０の読み取りに続く次の命令境界で更新される。それまでは、このレジスタは最後に更新された値を維持する。レジスタＯ＿Ｒ１　１５２およびＴ＿Ｒ１　１５３は、類似する形でハードウェアによって更新される。

【０１５０】

図１６に戻って、ＲＦ２６からのＲ０の読取は、Ｔ＿Ｒ０　１４３とＯ＿Ｒ０　１４２の間で選択される。この選択は、マルチプレクサ１４４によって、ＰＶＡ＿ｅｘｅ０と称する制御信号を使用して制御される。この信号は、計算機がＰＶＡモードで実行しており、ストリームL_0がイネーブルされる時にアサートされる。ＲＦ２６からのＲ０の読取によって、この場合には、Ｔ＿Ｒ０　１４３の内容が出力される。それ以外で、ＰＶＡ＿ｅｘｅ０がアサートされない時には、Ｒ０からの読取によって、Ｏ＿Ｒ０　１４２の内容が出力される。類似する制御機構が、ＰＶＡ実行中のＲＦ２６からのＲ１読取に関して存在する。

【０１５１】

レジスタＲ２　１６２は、オーバーレイされない。ＲＦ２６のレジスタＲ３からＲ１５と同様に、Ｒ２は、単一のインスタンスを有する。しかし、Ｒ２は、Ｓストリームがイネーブルされたｐｖａ実行中に、レジスタＯ＿Ｒ０　１４２およびＯ＿Ｒ１　１５２に類似する形で更新される。

【０１５２】

スカラ実行に関して、メモリ・ロードは、メモリ・ブロックＭ０　１４から$L_0$６を介してＬ０＿ｄｂｕｓに、またはＭ１　１６から$L_1$８を介してＬ１＿ｄｂｕｓに、データをフェッチすることと、その後、ＲＦ２６のレジスタのいずれかに書き込むことによって実行される。したがって、Ｌ０＿ｄｂｕｓまたはｌ１＿ｄｂｕｓからＲＦ２６のレジスタのいずれかにデータを書き込むパスが必要である。このパスは、図１６に示されているように、ＲＦ２６内のマルチプレクサ１６４によって提供される。マルチプレクサ１６４は、その入力をＬ０＿ｄｂｕｓ、Ｌ１＿ｄｂｕｓ、およびｒｅｓｕｌｔ＿ｂｕｓから取り、その出力をマルチプレクサ１４１、マルチプレクサ１５１、マルチプレクサ１６１、およびレジスタ・アレイ１６３に供給する。レジスタ・アレイ１６３によって、アーキテクチャ的レジスタＲ３からＲ１５が実施される。マルチプレクサ１６４を介して、Ｌ０＿ｄｂｕｓおよびＬ１＿ｄｂｕｓが、Ｔ＿Ｒ０　１４３およびＴ＿Ｒ１　１５３を除く、ＲＦ２６内のレジスタのいずれかに書き込み得る。この実施形態では、一時レジスタＴ＿Ｒ０　１４３およびＴ＿Ｒ１　１５３が、スカラ実行中に使用可能でない。

【０１５３】

ベクトル実行（ＣＶＡ実行またはＰＶＡ実行）の前に、あるレジスタを、正しく初期化する必要がある。これらの特殊なレジスタには、ハードウェアが正しいベクトル実行を実行するのに必要なすべての情報が含まれる。表３に、これらの特殊なレジスタを詳細に示す。

【０１５４】

【表３】

表３．ベクトル実行に関する特殊なレジスタ

レジスタ	表記	レジスタ内容
ストライドおよびサイズ・レジスタ	SSR	L_0、L_1、および S^1 に関するストライド値およびオペランド・サイズ
カウント・インデックス・レジスタ	CIR	CR: 実行される反復の回数 IXR: PVA ループの「ローカル PC」
汎用レジスタ R0	O_R0	L_0 のローカル・アドレス [1, 2]
汎用レジスタ R1	O_R1	L_1 のローカル・アドレス [1, 2]
汎用レジスタ R2	R2	S のストア・アドレス [1]
汎用レジスタ R3	R3	リダクション CVA またはハイブリッド CVA の部分的結果および最終結果 [1]
汎用レジスタ R4	R4	オペランド X のソース [1]
汎用レジスタ R5	R5	オペランド Y のソース [1]
汎用レジスタ R6	R6	オペランド Z のソース [1]

１．適用可能な場合に。
２．これらはオーバーレイされるインスタンスのみである。

この実施形態では、ＳＳＲ４２およびＣＩＲ５０だけが、特殊な制御レジスタである。他のすべてのレジスタが、汎用レジスタである。ベクトル演算に応じて、これらのレジスタが、ベクトル動作中に特殊な意味を伝える場合とそうでない場合がある。たとえば、Ｒ２は、ベクトル実行中にＳストリームがイネーブルされる時に特殊な意味を有するが、この特殊な意味は、Ｓストリームに関する最新のｃｓ－ｓｔｏｒｅアドレスがストアされることを指す。

【０１５５】

表３には、Ｒ０およびＲ１のオーバーレイされるインスタンスだけが含まれる。割込みまたは例外の時に、ＳＳＲ４２およびＣＩＲ５０が、ハードウェアによって保存される。これらの特殊な制御レジスタを保存するために、追加のハードウェア・サポートが必要である。レジスタＲ０からＲ６は、汎用レジスタなので、ハードウェアによって自動的に保存される。一時レジスタＴ＿Ｒ０　１４３およびＴ＿Ｒ１　１５３（表３に示さず）だけが、ハードウェアによって保存されない。

【０１５６】

割込みからリターンする時に、通常の実行を再開する前に、下記の情報を計算機に復元する必要がある。Ｏ＿Ｒ０　１４２、Ｏ＿Ｒ１　１５２、およびＲ２　１６２の内容を、ＲＦ２６から、それぞれＬ＿Ｒ０　１１４（L_0 ６内）、Ｌ＿Ｒ１　１２４（L_1 ８内）、およびＬ＿Ｒ２　１３６（Ｓ３６内）コピーする必要がある。後者の３つのローカル・レジスタは、それぞれ図１２から１４に示されている。

【０１５７】

さらに、リダクションＣＶＡおよびハイブリッドＣＶＡに関して、Ｒ３にストアされた中間の部分的結果を、Ｓ３６内のラッチ１３０に復元し、さらに、マルチプレクサ１３３を介してｓ＿ｄｂｕｓに復元する必要がある（図１４参照）。この部分的結果の復元には、ＲＦ２６からＲ３を読み取り、そのデータをＰ３４を介し、ｐ＿ｂｕｓに、さらにＳ３６を介してｓ＿ｄｂｕｓに転送することが含まれる（図４参照）。

【０１５８】

図４からわかるように、３つの独立のオンチップ・メモリ・ブロック、Ｍ０　１４、Ｍ１　１６、およびＴＭ２０がある。ＰＶＡ実行中にプログラム・ループ命令をストアするループ・キャッシュ２２もある。Ｍ０　１４およびＭ１　１６は、メインのオンチップ・

メモリである。Ｍ０　１４は、命令およびデータのストアに使用される。Ｍ１　１６は、データのみのストアに使用される。ＴＭ２０も、データのみのストアに使用され、具体的には、ベクトル実行中に一時ベクトルをストアするのに使用される。

【０１５９】

図４に示された１実施形態のメモリ・システムでは、ロード・ユニットL_0　６が、Ｍ０　１４およびＴＭ２０への読取アクセスを有し、ロード・ユニットL_1　８が、Ｍ１　１６およびＴＭ２０への読取アクセスを有し、Ｓ３６が、Ｍ０　１４、Ｍ１　１６、およびＴＭ２０のすべてへの書込アクセスを有する。Ｍ０　１４およびＭ１　１６は、単一ポート式メモリであるが、ＴＭ２０は、１つの読取ポートおよび１つの書込ポートを有する。これらのメモリ・ブロックの内容、アクセシビリティ、ならびに読取ポートおよび書込ポートの数を、表４に示す。代替実施形態で、スカラ動作およびベクトル動作を許容する他のメモリ構成を実施し得る。

【０１６０】

【表４】

表４．メモリの詳細

メモリ・ブロック	内容	データ・ストリーム			読取/書込ポートの数	ストリーム間のアービトレーション
		L0	L1	S		
M0	命令およびデータ	読取	-	書込	1 (読取または書込)	L_0 対 S
M1	データ	-	読取	書込	1 (読取または書込)	L_1 対 S
TM	データ (一時ベクトル)	読取	読取	書込	2 (1 つの読取と 1 つの書込)	L_0 対 L_1

ストリームL_0、L_1、およびＳをイネーブルされたコンパウンドＣＶＡまたはハイブリッドＣＶＡを実行するために、メモリ・システムは、１サイクルあたり１結果のピーク・スループット・レートを維持するために、１サイクルあたり２つのデータ読取および１つのデータ書込をサポートする必要がある。ＣＶＡ実行中には命令要求がない。これは、ＣＶＡ命令がデコードされ、実行されたならば、ＣＶＡ実行の残りに関してさらなる命令が不要であるからである。２つのデータ読取および１つのデータ書込に加えて、ＰＶＡ実行では、各サイクルに１つの命令フェッチも必要になる可能性がある。

【０１６１】

各サイクルに、図４に示されたメモリ・システムは、Ｍ０　１４、Ｍ１　１６、およびＴＭ２０を介する２つのデータ読取および１つのデータ書込までをサポートすることが可能であり、同一のサイクルに、ループ・キャッシュ２２の使用を介する１つの命令フェッチもサポートすることが可能である。３つのメモリ・ブロックＭ０　１４、Ｍ１　１６、およびＴＭ２０は、ある事前定義のメモリ空間を参照することによってアクセスし得る、すなわち、これらは、メモリ・マッピングされたブロックである。

【０１６２】

１実施形態によれば、図４に示されたＴＭ２０は、ベクトル実行中に一時ベクトルをストアするのに使用される小さいＲＡＭメモリである。ＴＭ２０は、ディジタル・フィルタリングの係数ベクトルなど、頻繁に使用される定数ベクトルに使用することも可能である。ＴＭ２０は、一時ベクトルを保持する、従来のベクトル計算機のベクトル・レジスタの拡張である。従来のベクトル計算機の最適化コンパイラと同様に、最適化コンパイラは、メモリに書き戻す前に、可能な限りこれらの一時ベクトルを操作することを試みる。ＴＭ２０は、Ｍ０　１４およびＭ１　１６に対するメモリ帯域幅圧力を減らすのに役立つ。ＴＭ２０は、これらのより大きいメモリ・ブロックの電力消費を減らすのにも役立つ。

【０１６３】

ＴＭ２０へのアクセスは、ベクトル・レジスタ番号など、ベクトル命令で指定される明

示的な参照ではなく、適当なメモリ空間を参照することによって行われる。具体的に言うと、これらのアクセスは、ストリームL$_0$、L$_1$、およびSをセットアップすることによって行われる。

【０１６４】

これらの一時ベクトルを構成し、割り振り、使用する時に、コンパイラは、一時記憶空間の編成においてより高い柔軟性を有する。たとえば、ＴＭ２０または他の類似する一時メモリが、ｎ要素のベクトルをストアし得る場合に、そのメモリを、それぞれがｎ／ｍ要素の長さを有するｍ個のベクトルの記憶空間として編成することも可能である。ＴＭ２０を、異なる長さの複数のベクトルの記憶空間として編成することも可能である。この場合に、コンパイラは、ＴＭ２０内のフラグメンテーションを最小にするようにベクトル割振りを管理することを試みる。

【０１６５】

次の例では、ＴＭ２０が、１つの読取ポートおよび１つの書込ポートを有する直接マッピングされる５１２バイトであると仮定する。ＴＭ２０を使用して、この例では、ベクトル実行を高速化し得る。

【０１６６】

やはり、例２に示されたループを検討されたい。このループが、Ｃ［ｉ］＝（ｌｓｒ（Ａ［ｉ］，Ｒ９）｜ｌｓｌ（Ａ［ｉ］、Ｒ８）を実行していることを想起されたい。ベクトル化されたコードを、下に示す。

【０１６７】

【数１６】

```
Some initialization code
// assign L0 to vector A; assign S to a temporary vector in TM
mov R5, R9
CVA   lsr @L0, R5, @S
Some initialization code
//assign L0 to vector A; assign L1 to the temporary vector in TM
//assign S to vector C.
mov R5, R8
CVA   lsl @L0, R5, @P,     or @L1, @P, @S
```

この例では、一時ベクトルが、ＴＭ２０内で作成され、割り振られる。第１のＣＶＡ命令のデスティネーションおよび第２のＣＶＡ命令のソース・オペランドの１つによって、ＴＭ２０を介して一時ベクトルがアクセスされる。第１のＣＶＡ命令は、Ｍ０　１４からL$_0$ストリームを介してベクトルＡをソーシングし、Ｓストリームを介して一時ベクトルをＴＭ２０に書き込む。第２のＣＶＡ命令は、Ｍ０　１４からL$_0$ストリームを介してベクトルＡをソーシングし、L$_1$ストリームを介してＴＭ２０から一時ベクトルをソーシングする。また、この命令は、Ｓストリームを介してＭ１　１６に結果ベクトルを書き込む。

【０１６８】

この例では、第２のＣＶＡ命令で、３つのデータ・ストリームすなわち、２つの入力、および１つの出力が使用される。これらのベクトル実行では、メモリ衝突は生じない。Ｍ０　１４およびＭ１　１６だけを使用すると、メモリ衝突が引き起こされる。ＴＭ２０がなければ、第２のＣＶＡ命令は、各結果要素を作るのに２サイクルを要する。

【０１６９】

処理されるベクトルのサイズが、ＴＭ２０のサイズより大きい時には、ベクトル動作が、ソフトウェア制御の下で複数のベクトル動作に分割され、各ベクトル動作が、ＴＭ２０のストレージで使用可能な長さのベクトルを操作する。この意味で、ＴＭ２０の使用は、ベクトル・レジスタの「ストリップマイニング（ｓｔｒｉｐ－ｍｉｎｉｎｇ）」に類似する。しかし、固定長ベクトル・レジスタと異なって、コンパイラは、この場合に、割振りと使用が可能な一時ベクトルの数とストリップマイニング反復の間のトレードオフに対する柔軟性を有する。

【０１７０】

もう１つの例として、ベクトル長がコンパイル時に未知であると仮定して、上のループのストリップマイニングおよびベクトル化を行う。要素サイズは、ワード（４バイト）であることが既知である。各ベクトル要素が、４バイト長であり、ＴＭ２０が５１２バイトなので、１２８要素を超える長さのベクトルは、ＴＭ２０のオーバーフローを避けるために、多少のストリップマイニングを必要とする。

【０１７１】

この例では、２つのＣＶＡ命令および１つの一時ベクトルだけが使用され、ＴＭ２０全体を、１２８以下の長さの一時ベクトルのストア専用にし得る。下に、Ｃ風の言語での、未知のベクトル長ｎを有するストリップマイニングされたコードを示す。

【０１７２】

【数１７】

```
low = 1;
VL = (n mod 128);                    // find the odd size piece first
for (j=0; j<n/128; j++)        {
        for (i=low; i<low+VL-1; i++) {        // runs for length VL
        C[i] = (lsr(A[i],R9)  |  lsl(A[i],R8)); //main CVA operations
        }
        low = low + VL;
        VL = 128;                    // reset VL to 128 after the first
                                     // odd size piece
}
```

ある独立のベクトルＡ、Ｂ、Ｃ、およびＤに関して、ベクトル・リダクション演算

Σ_i（（Ａ［ｉ］＊Ｂ［ｉ］＋Ｃ［ｉ］）＊　Ａ［ｉ］＊Ｂ［ｉ］＊Ｄ［ｉ］）

の実行の例のように、最高の可能な性能レベルを維持しながら電力消費を減らすのにＴＭ２０を使用することも可能である。ベクトルＡおよびＣが、Ｍ０　１４内にあり、ベクトルＢおよびＤが、Ｍ１　１６内にあると仮定する。

【０１７３】

３つの一時ベクトルおよび４つのＣＶＡ命令（３つのコンパウンドＣＶＡおよび１つのリダクションＣＶＡ）を使用する、実行時間に関する最適解を下に示す。

（１）Ｔ１［ｉ］＝Ａ［ｉ］＊Ｂ［ｉ］；

（２）Ｔ２［ｉ］＝Ｔ１［ｉ］＋Ｃ［ｉ］；

（３）Ｔ３［ｉ］＝Ｔ１［ｉ］＊Ｄ［ｉ］；

（４）リダクション結果＝Σ_i（Ｔ２［ｉ］＊Ｔ３［ｉ］）

ベクトルＡおよびＢは、Ｍ０　１４およびＭ１　１６にあるので、ベクトルＴ１は、ＴＭ２０内で割り振られなければならない。したがって、ベクトルＴ３は、ベクトルＤがＭ１　１６内にあるものとして、Ｍ０　１４内になければならない。ベクトルＣはＭ０　１４

にあるので、ベクトルＴ２は、Ｍ１　１６またはＴＭ２０内に置き得る。表５に、一時ベクトルＴ１およびＴ２の割振りの２つの可能な解決を示す。

【０１７４】

【表５】

表５．一時メモリ割振り

一時ベクトル	解決 (I)			解決 (II)		
	M0	M1	TM	M0	M1	TM
T1			X			X
T2		X				X
T3	X			X		

この解決の両方が、メモリ衝突をこうむらない。両方の解決が、この計算機での最適性能レベルを達成する。しかし、表５の解決（ＩＩ）は、ベクトルＴ２がＴＭ２０に割り振られるので、より低電力の解決を与える。解決（ＩＩ）の短所は、一時ベクトルＴ１およびＴ２の両方が同時にＴＭ２０に存在することを必要とすることである。ＴＭ２０が、両方のベクトルを保持するのに十分に大きくはない場合に、解決（Ｉ）が、唯一の実行可能な解決である。ＴＭ２０が、単一のベクトルにも小さすぎる場合には、解決（Ｉ）をストリップマイニングする必要がある。

【０１７５】

本発明の１実施形態のベクトル実行のすべてが、割込み可能である。割込みは、ベクトル計算の完了の前のベクトル実行の一時的中断を引き起こし得る。スカラ・コンテキストに加えて、追加のベクトル・コンテキストを保存し、その結果、割込みからのリターンの際にベクトル実行を正しく再開可能にする必要がある。割り込み時に、これらのベクトル・コンテキストの一部は保存され、一部は破棄される。保存されるベクトル・コンテキストに、表３に示されたすべてのレジスタが含まれる。このレジスタには、ＳＳＲ４２、ＣＩＲ５０、オーバーレイされるインスタンスＯ＿Ｒ０　１４２およびＯ＿Ｒ１　１５２、Ｒ２　１６２、およびＲＦ２６内の他のレジスタすなわち、Ｔ＿Ｒ０　１４３およびＴ＿Ｒ１　１５３を除くＲＦ２６内のすべてのレジスタが含まれる。

【０１７６】

ＣＶＡ実行に関して、各ベクトル要素に対する動作は、あるベクトル要素に対する動作の結果が割込み発生時に書き戻されない場合に、すべての中間結果が破棄されるという意味で、「アトミック」である。この１つまたは複数の要素に対して実行されるすべての動作が、割込みからのリターンの際に繰り返されなければならない。リダクションＣＶＡ実行またはハイブリッドＣＶＡ実行に関して、各サイクルに作られる部分的結果が、ＲＦ２６内のレジスタＲ３に継続的に書き戻される。割込み時に、部分的結果が既にＲＦ２６にあるので、したがって、部分的結果は自動的に保存される。これによって、部分的結果を保存するための追加の浪費される時間が除去される。しかし、割込みからのリターンの時に、ＲＦ２６内のレジスタＲ３の内容が、通常のＣＶＡ実行を再会する前にｓ＿ｄｂｕｓに復元される。

【０１７７】

ＰＶＡ実行に関して、ループ本体で作られるすべての中間結果が、レジスタ・ファイル、ＲＦ２６にストアされる。したがって、中間結果を保存するために追加の時間は浪費されない。一時レジスタＴ＿Ｒ０　１４３およびＴ＿Ｒ１　１５３は、ベクトル・コンテキストの一部として保存されない。割込みからリターンする時に、これらの一時物をフェッチしたｃｓ－ｌｏａｄ命令は、入力ストリームがイネーブルされる場合に再初期化される。その後、一時レジスタＴ＿Ｒ０　１４３およびＴ＿Ｒ１　１５３が、通常のＰＶＡ実行を再会する前にそれ相応に更新される。この場合に、ハードウェアは、割込みをサービスする過程の間にメモリ位置が変更されなかったと仮定する。

【０１７８】
図１８に、図４のプログラム・シーケンサ２４の一部を示す。ループ実行および分岐のターゲットを監視するために、カウンタ・ベースの方式を使用し得る。ＰＶＡ命令に出会う時に、その命令で指定されるＬｏｏｐ＿ｓｉｚｅが、図４のプログラム・シーケンサ２４などのハードウェアによって取り込まれる。さらに、ＩＸＲ７０（図１１参照）を使用して、ループ本体のどの命令が現在実行されているかを追跡する。ＩＸＲの実施の１実施形態を、図１８に示す。ＩＸＲ７０は、ループ本体の中でローカル・プログラム・カウンタ（ＰＣ）のように振る舞う。ループの最初の命令が実行されている時に、ＩＸＲ７０に１がセットされる。順次実行される命令ごとに、ＩＸＲ７０が１つ増分される。ループの最後の命令が実行されている時に、ＩＸＲ７０はＬｏｏｐ＿ｓｉｚｅと等しい。ＩＸＲ７０が（Ｌｏｏｐ＿ｓｉｚｅ－１）と等しい時に、命令フェッチが、ループの最初の命令に向けられる。最後の命令が、シーケンシャル命令であるか、命令が制御フローの変更を引き起こさない（ターゲットがループ本体の外部にある）場合に、実行が、ループの先頭に転送され、ＩＸＲ７０が、１にリセットされる。

【０１７９】
ＩＸＲ７０は、加算器７２の出力であるＩＸＲ＿ｃｏｕｎｔ値を受け取る。ＩＸＲ７０にストアされた値は、一方の入力として加算器７２に供給され、他方の入力は、マルチプレクサ７４から来る。マルチプレクサ７４への一方の入力は、１であり、他方の入力は、分岐命令の変位フィールドである。この形で、加算器７２は、次の順次値または分岐変位値のいずれかを出力する。ＩＸＲ７０は、ＩＸＲ７０を１にリセットするｒｅｓｅｔ信号と、次のＩＸＲ＿ｃｏｕｎｔを受け入れるｌｏａｄ信号も受け取る。

【０１８０】
分岐がＰＶＡ実行中に採用される（順方向または逆方向のいずれか）場合に、分岐命令の分岐変位フィールドが、ＩＸＲ７０に加算される。この加算演算は、マルチプレクサ７４および加算器７２によって実行される。マルチプレクサ７４は、この場合に、加算器７２への分岐命令の変位フィールドを選択する。このレジスタ値と分岐変位の合計が負またはループ・サイズを超える場合には、分岐ターゲットがループ本体の外部にあることが示され、ＰＶＡループ実行が終了する。

【０１８１】
図１８を続けると、ＩＸＲ＿ｃｏｕｎｔ値も、入力としてブロック７６、７８、および８０に供給される。ＩＸＲ＿ｃｏｕｎｔは、継続的に０、Ｌｏｏｐ＿ｓｉｚｅ、およびＬｏｏｐ＿ｓｉｚｅ－１と比較される。（ＩＸＲ＿ｃｏｕｎｔ＜０）または（ＩＸＲ＿ｃｏｕｎｔ＞Ｌｏｏｐ＿ｓｉｚｅ）である場合に、命令がループ本体の外で実行されようとしていることが示される。ブロック７８で（ＩＸＲ＿ｃｏｕｎｔ＜０）と判定される場合に、ブロック７８の出力が、「ｏｒ」ゲート８２への入力としてアサートされる。同様に、ブロック８０で（ＩＸＲ＿ｃｏｕｎｔ＞Ｌｏｏｐ＿ｓｉｚｅ）と判定される場合に、ブロック８０の出力が、「ｏｒ」ゲート８２にアサートされる。どちらの場合でも、「ｏｒ」ゲート８２の出力によって、ループ実行が終了される。また、ブロック７６で（ＩＸＲ＿ｃｏｕｎｔ＝＝Ｌｏｏｐ＿ｓｉｚｅ－１）と判定される場合に、ループの先頭からの命令フェッチが開始される。

【０１８２】
ＰＶＡ命令の最後の反復の終りに向かって、ループ内の最後の命令が、ループ本体の外のターゲット・アドレスを有する採用される条件分岐である場合に、ループの最初の命令を誤ってフェッチし、実行することに関する１サイクルのペナルティをこうむる。この場合に、この命令の実行の結果が、却下される、すなわち、結果が、破棄され、書き戻されない。分岐ターゲットを使用する新しい命令フェッチが開始される。

【０１８３】
割込みからＰＶＡ実行を再開する時に、まず、必要なループ制御情報を回復しなければならない。これは、複数の形で行い得る。１実施形態で、ＰＶＡ命令に最初に出会う時に、ＰＣ（ＰＶＡ命令のアドレス）が、一時ハードウェア位置に保存される。割込み時に、

このハードウェア・コピーがＰＣとして保存される。割込みからリターンする時に、まずＰＣを使用してＰＶＡ命令をフェッチして、Ｌｏｏｐ＿ｓｉｚｅ、ｃｓ－ｓｔｏｒｅ－ｉｎｄｅｘなどを含むループ制御情報のすべてを回復する。この情報が回復された後に、ＩＸＲ７０の内容がＰＣに加算される。ＰＶＡループ実行が、新しいＰＣによってポイントされる命令から継続される。

【０１８４】

もう１つの実施形態で、ＰＶＡ実行が割り込まれる時に、ＰＣは、ループ本体の中の、実行が割り込まれた命令をポイントする。このＰＣを、実行コンテキストの一部として保存する。割込みからリターンする時に、ＩＸＲ７２の内容をＰＣから減算して、ＰＶＡ命令アドレスを得る。次に、ＰＶＡ命令をフェッチして、ｃｓ－ｓｔｏｒｅ－ｉｎｄｅｘ、Ｌｏｏｐ＿ｓｉｚｅなどを含むループ制御情報のすべてを回復する。

【０１８５】

ＰＶＡ実行では、「ｃｓ－ｓｔｏｒｅ」ラベルに配置された命令およびそれに関連するｃｓ－ｓｔｏｒｅ動作の実行が、「アトミック」動作である。例４からのベクトル化されたＰＶＡループをもう一度検討されたい。

【０１８６】

【数１８】

```
<Some initialization code>
// assign L0 to vector A; assign S to C[i].
mov         R4, 4
mov         R8, 8
mov    R6, 16

PVA         @L0, @S, #4;        // PVA instruction
mov         R2, R8
cmplt       R6, R0        // is A[i] > 16?
cs-store:
movt        R2, R4        // R2 = (A[i]>16)? R4:R8;
                          // cs-store performed here
```

「ｍｏｖｔ」命令および関連するｃｓ－ｓｔｏｒｅ動作は、「アトミック」である。ｃｓ－ｓｔｏｒｅが、割込みに起因して完了しない場合に、「ｍｏｖｔ」命令も、「実行されない」と考えられる。割込みからリターンする時に、「ｍｏｖｔ」命令から実行が再開される。

【０１８７】

上で示したベクトル化された例のそれぞれについて、コンパイラは、元のコードを評価するが、このコードは、高水準言語プログラミング言語またはアセンブリ・コードで記述され得る。コンパイラは、コードを分析して、ＣＶＡ命令および／またはＰＶＡ命令に適用可能なループおよび他の構成を探す。本発明の１実施形態によるコンパイルの処理フローを、図１９に示す。

【０１８８】

図１９からわかるように、この処理では、まず、判断菱形１７０で、ループまたはコードのセクションが、ＣＶＡ実行を使用してベクトル化可能であるかどうかを判断し、そうである場合に、ブロック１７１で、少なくとも１つのＣＶＡ命令を使用してコードをベク

トル化する。次に、ブロック１７２で、ＣＶＡベクトル化されたループの実行時間を推定する。次に、判断菱形１７３に継続する。

【０１８９】

その代わりに、コードがＣＶＡベクトル化可能でない場合にも、判断菱形１７３に継続されて、ＣＶＡ実行およびＰＶＡ実行の組合せを使用してコードをベクトル化可能であるかどうかを判定する。そうである場合には、ブロック１７４でそのような組合せを使用してコードをベクトル化し、ブロック１７５で、その実行速度を推定する。その後、処理フローは判断菱形１７６で継続される。

【０１９０】

コードが、ＣＶＡ単独またはＣＶＡ／ＰＶＡ組合せのいずれによってもベクトル化可能でない場合にも、処理は、判断菱形１７６に継続されて、コードがＰＶＡベクトル化可能であるかどうかを判定する。そうである場合には、ブロック１７７で、少なくとも１つのＰＶＡ命令を使用してコードをベクトル化する。ブロック１７８で、その実行時間を推定する。

【０１９１】

判断菱形１７９では、ベクトル化されたコードの実行速度が、上で述べたベクトル化方法のいずれかを使用して、元のループより改善されるかどうかを判定する。改善がない場合には、ブロック１８１で、元のコードを変更しない。実行速度が改善された場合には、ステップ１８０で、最短の実行時間を有する最良のベクトル化方法を使用するコードを実施する。この実施形態では、コンパイラが、可能なすべてのベクトル化方法を試行し、その中から最良の方法を選択する。この実施形態では、実行時間が、唯一の選択判断基準として使用される。他の実施形態では、電力消費などの他の判断基準の使用を組み込んで、最良のベクトル化方法を選択することも可能である。

【０１９２】

代替実施形態に、まずＣＶＡ構成だけを使用することによるコードのベクトル化を組み込むことも可能であることに留意されたい。この実施形態では、次に、コードがＣＶＡベクトル化可能でない場合に限ってＣＶＡ／ＰＶＡの組合せを試行する。ＣＶＡ／ＰＶＡの組合せを使用してもコードがベクトル化可能でない場合には、ＰＶＡ構成を使用するベクトル化を試行する。この代替実施形態では、最も頻繁に、ＣＶＡベクトル化が実行速度の最大の改善を作り、次がＣＶＡ／ＰＶＡの組合せであり、最後がＰＶＡベクトル化であることが認識されている。この処理は、コードをベクトル化し得る最初の方法を見つけた時に終了する。この代替実施形態は、実行時間を推定するのに使用されるプロファイル情報が入手可能でない時に有利である。他の代替実施形態で、所与のアプリケーションに適する他の順序付けを見つけ得る。

【０１９３】

この実施形態によれば、ＰＶＡ実行は、各サイクルに１つの命令を発行することに制限される。代替実施形態では、ＰＶＡ実行を、図２に示されたものに類似する、各サイクルに複数の独立の命令を発行する機能を組み込むように拡張し得る。

【０１９４】

そのような拡張の１つである二重発行擬似ベクトル計算機を、図２１に示すが、この図では、各サイクルに、２つの独立の命令を、２つの独立の計算ユニット、Ｐ１　２３５およびＰ２　２３４に発行し得る。スカラ・モードまたはＰＶＡモードで実行される時に、２つの結果を、レジスタ・ファイルＲＦ２３６またはメモリに書き戻し得る。

【０１９５】

図２１で、レジスタ・ファイルＲＦ２２６ならびにデータ・ストリームL_0およびL_1は、独立に、マルチプレクサ２２８、２３０、２３１、２３２、および２３３にデータを供給する。マルチプレクサ２２８および２３０は、機能ユニットＰ２　２３４にデータを供給する。マルチプレクサ２３１および２３２は、機能ユニットＰ１　２３５にデータを供給する。Ｐ２　２３４は、ｒｅｓｕｌｔｌ＿ｂｕｓを介してＳ２３６およびＲＦ２２６にデータを供給する。Ｐ１　２３５は、マルチプレクサ２３７およびＳ２３６にデータを

供給する。マルチプレクサ２４１は、Ｓ２３６にデータを供給する。Ｓ２３６は、データ・ストリームＳを介してメモリ・システムに、ならびにマルチプレクサ２３７および２４１にデータを供給する。マルチプレクサ２３７は、ｒｅｓｕｌｔ２＿ｂｕｓを介してＲＦ２２６にデータを供給する。ＲＦ　２２６も、マルチプレクサ２２８、２３０、２３１、および２３２にデータを供給する。

【０１９６】

この二重発行擬似ベクトル計算機では、３タイプのＣＶＡすなわち、コンパウンドＣＶＡ、リダクションＣＶＡ、およびハイブリッドＣＶＡがある。この３タイプのＣＶＡの依存性グラフを、図２２に示す。図５の依存性グラフと対照的に、この計算機では、リダクションＣＶＡおよびハイブリッドＣＶＡに、下記の一般形式を有する３入力算術関数ｓ＿ｏｐが含まれる。

【０１９７】

スカラＴ、Ｕ、およびＶについて、Ｔ　ｓ＿ｏｐ　Ｕ　ｓ＿ｏｐ　Ｖ

この依存性グラフでは、オペランドＷ、Ｘ、Ｙ、およびＺが、独立に、入力ストリーム$Ｌ_0$、入力ストリーム$Ｌ_1$、または指定されたレジスタからソーシングし得る。このオペランド・ソーシング・モードを、本明細書では独立ソーシング・モードと称する。その代わりに、これらのオペランドのそれぞれが、入力ストリーム$Ｌ_0$または$Ｌ_1$の０拡張された上位ハーフワードまたは下位ハーフワードからソーシングすることも可能である。このオペランド・ソーシング・モードを、本明細書ではクロス・ソーシング・モードと称する。表６に、独立ソーシング・モードおよび２つの異なるクロス・ソーシング・モード（クロス・ソーシング・モードＩおよびＩＩ）を示す。

【０１９８】

【表６】

表６．オペランドＷ、Ｘ、Ｙ、およびＺの可能なソース

オペランド	オペランド・ソーシング・モード		
	独立ソーシング・モード	クロス・ソーシング・モードI	クロス・ソーシング・モードII
W	L_0、L_1、R0	{0, L_0[31:16]}	{0, L_0[31:16])
X	L_0、L_1、R4	{0, L_1[15:0]}	{0, L_1[31:16]}
Y	L_0、L_1、R8	{0, L_1[31:16]}	{0, L_1[15:0]}
Z	L_0、L_1、R12	{0, L_0[15;0]}	{0, L_0[15:0]}

この表で、｛０、$Ｌ_0$［１５：０］｝は、入力データ・ストリーム$Ｌ_0$からの０拡張された下位ハーフワードを表す。｛０、$Ｌ_0$［３１：１６］｝は、入力データ・ストリーム$Ｌ_1$からの０拡張された上位ハーフワードを表す。類似する表記が、$Ｌ_1$ストリームに使用される。これらの０拡張動作の結果は、３２ビット・オペランドである。コンパウンドＣＶＡの一般形式は、次のように表し得る。

【０１９９】

$Ｓ_i$＝（$Ｗ_i$　ｐｌ＿ｏｐ　$Ｘ_i$）ｓ＿ｏｐ（$Ｙ_i$　ｐ２＿ｏｐ　$Ｚ_i$）　ｉ＝０、．．．、ｎ－１

リダクションＣＶＡの一般形式は、次のように表し得る。

【０２００】

$Ｓ_0$＝（$Ｗ_0$　ｐｌ＿ｏｐ　$Ｘ_0$）ｓ＿ｏｐ（$Ｙ_0$　ｐ２＿ｏｐ　$Ｚ_0$）；

$Ｓ_i$＝（$Ｗ_i$　ｐｌ＿ｏｐ　$Ｘ_i$）ｓ＿ｏｐ（$Ｙ_i$　ｐ２＿ｏｐ　$Ｚ_i$）ｓ＿ｏｐ $Ｓ_{i-1}$、　ｉ＝１、．．．、ｎ－１；

Ｒ＝$Ｓ_{n-1}$

ここで、$Ｓ_i$は、ｉ番目の部分的結果を表し、Ｒは、ベクトル・リダクション演算のスカ

ラ結果を表す。

【０２０１】

ハイブリッドＣＶＡの一般形式は、S_i、ｉ＝０、．．．、ｎ－１が、Ｓストリームを介してメモリに書き込まれることを除いて、リダクションＣＶＡと同一である。

クロス・ソーシング・モードの多数の使用の１つが、ベクトルＡ、Ｂ、およびＣの要素のすべてが複素数である複素ベクトル乗算Ｃ［ｉ］＝Ａ［ｉ］＊Ｂ［ｉ］、ｉ＝０、．．．、ｎ－１である。複素数は、メモリ・システム内で２つの１６ビット・ハーフワードの連結として表すことが可能であり、上位ハーフワードは、複素数の実数部を表し、下位ハーフワードは、複数数の虚数部を表す。この表現を使用すると、複素数が、３２ビット・データによって完全に表される。

【０２０２】

２つの複素数ＸおよびＹの乗算の結果は、（Ｒｅ　Ｘ＊Ｒｅ　Ｙ－Ｉｍ　Ｘ＊Ｉｍ　Ｙ）＋ｊ（Ｒｅ　Ｘ＊Ｉｍ　Ｙ＋Ｉｍ　Ｘ＊Ｒｅ　Ｙ）によって与えられ、ここで、「Ｒｅ　Ｗ」は、複素数Ｗの実数部を表し、「Ｉｍ　Ｗ」は、複素数Ｗの虚数部を表す。

【０２０３】

図２１に戻ると、２つの複素ベクトルＡおよびＢの間の乗算を、２つのコンパウンドＣＶＡを使用することによって実行し得る。これらのＣＶＡの両方が、L_0ストリームを介するベクトルＡおよびL_1ストリームを介するベクトルＢでストリーム化される。このＣＶＡの両方で、ｐ１＿ｏｐおよびｐ２＿ｏｐが、乗算関数であり、ｓ＿ｏｐが、加算関数である。第１ＣＶＡ命令では、クロス・ソーシング・モードＩを使用し、結果ベクトルＣの虚数部を含む結果ベクトルが作られる。第２ＣＶＡ命令では、クロス・ソーシング・モードＩＩを使用し、結果ベクトルＣの実数部を含む結果ベクトルが作られる。ストリームＳについて２のオペランド・サイズ（ハーフワード）および４のストライド値（１ワード離れる）を使用することによって、この２つのＣＶＡ命令を、同一のベクトル位置Ｃに書き込むことが可能であり、具体的には、２つのＣＶＡ命令を、互いの結果を上書きせずにベクトルＣに書き込み得る。２つのＣＶＡ命令の実行の後に、ベクトルＣの各要素に、要求された実数値および虚数値が含まれる。

【０２０４】

上で説明したクロス・ソーシング・モードを、ＰＶＡ実行にも使用し得る。この場合に、ｐ１＿ｏｐを実行するＰ１　２３５と、ｐ２＿ｏｐを実行するＰ２　２３４によって作られる２つの結果が、レジスタ・ファイルＲＦ　２２６内の２つの別個のレジスタおよび／またはメモリ・システムに独立に書き戻される。

【０２０５】

図２７から３６に、あるプログラム・ループ命令に関連するサイズまたは長さを拡張することによって、ＰＶＡ実行に関連する能力を高める方法を示す。本発明の１実施形態では、ＰＶＡ実行中に処理される命令などのループ実行に関連するある命令に、標準命令サイズを使用する標準部分と、第２の命令サイズを使用する増補命令部分が含まれる。増補命令部分の追加によって、ループ実行中に処理される増補命令の総合的なサイズが、強化された実行能力を提供するように拡張される。本発明のいくつかの実施形態は、あるプログラム・ループの命令に伴う追加の機能性を含めることによって、クリティカル・ループ実行の性能を高めるという長所を有する。

【０２０６】

図２７を参照すると、本発明の１実施形態による、標準命令および増補命令を処理するシステムを示すブロック図が示され、システム４００として全体的に参照される。デコーダ４３０および４４０は、プログラム４０５の命令を処理するのに使用される。プログラム４０５は、メモリ４５０にストアされる。プログラム４０５には標準命令４６０および４６５ならびに増補命令４７０が含まれる。増補命令４７０には、標準命令部分すなわち標準命令部分４７５と、増補命令部分である増補命令部分４７７が含まれる。標準デコーダ４３０は、標準命令４６０および４６５ならびに増補命令４７０の標準命令部分４７５の処理に使用される。副デコーダ４４０は、増補命令４７０の増補命令部分４７７の処理

に使用される。

【０２０７】

１実施形態で、標準デコーダ４３０が、標準命令４６０および４６５など、命令の特定のグループに関連する標準コマンドの処理に使用される。命令フェッチ・コンポーネント４２０が、メモリ４５０から次の命令にアクセスするのに使用される。たとえば、標準デコーダ４３０は、プログラム４０５の次に処理される命令にアクセスするように命令フェッチ・コンポーネント４２０に指示し得る。次の命令のアドレスに関連するメモリ・アドレスを命令フェッチ・コンポーネント４２０に供給して、メモリ４５０内の次の命令の位置を示し得る。代替案では、命令フェッチ・コンポーネント４２０が、プログラム・カウンタ（図示せず）を監視して、次に処理される命令のアドレスを判定し得る。

【０２０８】

１実施形態で、命令フェッチ・コンポーネント４２０が、標準命令４６０の命令にアクセスする。したがって、命令フェッチ・コンポーネント４２０は、処理のために標準デコーダ４３０に命令を供給する。１実施形態で、標準命令４６０および４６５の命令に、スカラ命令または非ループ命令が含まれる。増補命令４７０の命令に、プログラム・ループ命令が含まれる。したがって、システム４００は、増補命令４７０を処理するためにループ実行モードまたはＰＶＡ実行モードに切り替え得る。１実施形態で、ループ実行モードまたはＰＶＡ実行モードが、ＰＶＡ命令またはループ初期化命令によってトリガされる。たとえば、増補命令４７０の最初の命令または標準命令４６０の最後の命令を、ＰＶＡ実行を開始するのに使用されるＰＶＡ命令とすることが可能である。その代わりに、ループ実行またはＰＶＡ実行を、「ＷＨＩＬＥ」コマンドまたは「ＤＯ　ＵＮＴＩＬ」コマンドなどのプログラム・ループ命令によってトリガし得る。ループ実行モードまたはＰＶＡ実行モードをトリガする他の方法を、本発明の範囲から逸脱せずに使用可能であることに留意されたい。さらに、システムは、コンパイル処理中に、標準命令の代わりに増補命令を使用可能にし得る。コンパイラは、後で説明するように、ループまたはループ開始コマンドに出会う時に、増補命令の使用を可能にし得る。

【０２０９】

システム４００が、ＰＶＡ実行モードまたはループ実行モードに入ったならば、副デコーダ４４０がイネーブルされる。１実施形態で、副デコーダ４４０は、ＩＮＴＥＲ－ＤＥＣＯＤＥＲ　ＣＯＮＴＲＯＬ信号４１０を介するなど、標準デコーダ４３０によってイネーブルまたはディスエーブルされる。したがって、１実施形態で、副デコーダは、命令フェッチ・コンポーネント４２０に類似する命令フェッチ・コンポーネント４２１を使用して、標準デコーダ４３０によってアクセスされる標準命令部分４７５の標準命令部分に関連する増補命令部分４７７の増補命令部分にアクセスする。代替実施形態では、単一の命令フェッチ・コンポーネント４２０だけが、標準命令部分４７５ならびに増補命令部分４７７の両方をメモリ４５０からフェッチするのに使用し得る。そのような実施形態では、副デコーダ４４０が、命令フェッチ・コンポーネント４２０および標準デコーダ４３０によって増補命令４７０の増補命令部分４７７を受け取り得る。そのような代替実施形態では、標準デコーダ４３０が、修正またはデコードを実行せずに、副デコーダ４４０に増補命令部分４７７を供給し得る。代替案では、命令フェッチ・コンポーネント４２０が、副デコーダ４４０に増補命令部分４７７を直接に供給し得る。

【０２１０】

１実施形態で、増補命令４７０が、プログラム・ループ４８０の一部である。プログラム・ループ４８０には、ループ初期化コマンドと、それに続くＫ個の増補命令の組が含まれ、ここで、Ｋは、１を超える、増補命令の数である。１実施形態で、ループ初期化コマンドが、ループ実行を初期化するために含まれる。たとえば、ループ初期化コマンドは、Ｋ個の増補命令が処理される反復の回数を示し得る。他のプログラム・ループに関連する命令を、標準命令４６０および４６５内に含め得ることに留意されたい。しかし、標準命令４６０および４６５のプログラム・ループ命令は、増補命令４７０のプログラム・ループ命令と比較して、増補命令部分を有しない。Ｋ個の増補命令は、標準コードおよび増補

コードのＫ個の組に分解される。

【０２１１】

標準コードには、フォーマットにおいて標準命令４６０および４６５の少なくとも一部の命令に似るコードが含まれる。たとえば、標準コードに、オペコードおよび１つまたは複数のオペランドを含めることが可能であり、オペコードは、オペランドに対して実行される動作を指定し得る。標準コードは、標準デコーダ４３０を使用して処理される。１実施形態で、標準命令部分４７５の標準コードに、増補命令４７０の命令に関連する、ビット０からＮなどのビットの第１の組が含まれる。増補命令部分４７７の増補コードに、増補命令４７０の命令の、ビットＮ＋１からＭなどのビットの第２の組が含まれる。本発明の範囲から逸脱せずに、標準コードで、他のフォーマットを使用し得ることを諒解されたい。ＮおよびＭは、特定のビット位置を指定するのに使用され、ＭはＮより大きい。１実施形態で、標準命令４６０および４６５ならびに標準命令部分４７５が、同一の固定サイズを有し、類似するか同一の命令エンコーディングを使用し得る。

【０２１２】

増補命令部分４７７のコードによって、標準命令部分４７５の処理コードに関連する特定の拡張機能性を指定することが可能である。１実施形態で、標準命令部分４７５の処理が、増補命令部分４７７の処理に基づく。１実施形態で、標準デコーダ４３０が、ｓｔａｎｄａｒｄ＿ｃｏｄｅ＿１などの特定の標準命令部分４７５にアクセスするので、副デコーダ４４０は、ａｕｇｍｅｎｔｅｄ＿ｃｏｄｅ＿１などの関連する増補命令部分４７７にアクセスする。ｓｔａｎｄａｒｄ＿ｃｏｄｅ＿１の処理は、ａｕｇｍｅｎｔｅｄ＿ｃｏｄｅ＿１の処理に依存し得る。たとえば、１実施形態で、ａｕｇｍｅｎｔｅｄ＿ｃｏｄｅ＿１を使用して、ｓｔａｎｄａｒｄ＿ｃｏｄｅ＿１の条件実行が指定される。システム４００の条件コードとａｕｇｍｅｎｔｅｄ＿ｃｏｄｅ＿１によって指定される条件とのマッチングに応じて、副デコーダ４４０は、ＩＮＴＥＲ－ＤＥＣＯＤＥＲ　ＣＯＮＴＲＯＬ信号４１０を使用して、標準デコーダ４３０でのｓｔａｎｄａｒｄ＿ｃｏｄｅ＿１の処理を取り消し得る。

【０２１３】

増補命令部分４７７は、標準命令部分４７５に関連する実行の他の形を指定するのに使用し得る。たとえば、増補命令部分４７７の個々のコードによって、標準命令部分４７５の個々のコードに関連する早期終了を指定し得る。したがって、増補命令部分４７７の命令によって、プログラム・ループ４８０のさらなる実行を終了しなければならないことを示し得る。したがって、システム４００の条件コードと増補命令部分４７７の命令によって指定される条件に応じて、副デコーダ４４０は、ＩＮＴＥＲ－ＤＥＣＯＤＥＲ　ＣＯＮＴＲＯＬ信号４１０を提供して、増補命令４７０のさらなる命令をスキップし得る。たとえば、標準デコーダ４３０に、増補命令４７０の代わりに標準命令４６５の最初の命令を処理するように指令し得る。

【０２１４】

増補命令部分４７７のコードによって、標準命令部分４７５の関連コードを処理する時の早期継続を指定することも可能である。たとえば、システム４００の条件コード値と、副デコーダ４４０によって処理されている増補命令部分４７７の現在のコードによって指定される条件とに応じて、副デコーダ４４０は、共通の標準デコーダ４３０にＩＮＴＥＲ－ＤＥＣＯＤＥＲ　ＣＯＮＴＲＯＬ信号４１０を供給して、プログラム・ループ４８０の次の反復をスキップし得る。

【０２１５】

増補命令部分４７７のコードを使用して、標準命令部分４７５のコードを処理するための追加のオペランド指定子を提供することも可能である。１実施形態で、増補命令の増補コードによって、第１オペランドおよび第２オペランドの処理からの値が返される第３レジスタ・オペランドを指定し得る。１実施形態で、増補コードにイネーブル・ビットを含めて、デスティネーション・オペランドとしての第３オペランドの使用をイネーブルすることも可能である。増補命令部分４７７に関連するコードによって、標準命令部分４７５

のコードの処理に関するオペランドとして使用される即値フィールドをエンコードすることも可能である。代替案では、増補命令部分４７７によって、標準命令部分４７５で既にエンコードされている即値フィールドの長さを拡張するのに使用される即値フィールドをエンコードすることも可能である。この拡張の結果として、即値フィールド・オペランドの結果の長さが増やされる。

【０２１６】

本発明の１実施形態では、増補命令部分４７７を、後で図２８で示すように、関連する標準命令部分４７５からメモリ４５０に別々にストアし得る。増補命令部分４７７を、後で図２９で示すように、標準命令４６０の命令などの別々のコマンドを介して指定し得る。代替案では、標準命令部分４７５を、後で図３０で示すように、標準命令部分４７５の各関連する部分に隣接して提供し得る。

【０２１７】

本発明の１実施形態で、増補命令部分４７７が、標準デコーダ４３０で処理される関連する標準命令部分４７５と時間的に並行して副デコーダ４４０によって処理される。

図２８を参照すると、本発明の１実施形態による、メモリに増補命令部分をストアする形を示すブロック図が示されている。標準命令４６０および４６５に、標準デコーダ４３０（図２７）などの標準命令デコーダを使用して処理される標準命令が含まれる。１実施形態で、標準命令４６０および４６５に、同一の固定命令長が含まれる。増補命令が、特定の命令に関連する機能性を拡張するために提供され、この増補命令には、標準命令４６０および４６５の固定命令長より長い命令長が含まれる。増補命令は、標準命令部分５１０および増補命令部分５１５からなる。図示の実施形態では、増補命令部分５１５が、メモリ５０５のうちで、関連する標準命令部分５１０と異なる部分にストアされる。

【０２１８】

増補命令は、一般に、ループ実行命令またはＰＶＡ実行命令である。標準命令部分５１０に、標準命令４６０および４６５に類似するコマンドが含まれ、標準命令４６０および４６５に関連するものと同一の固定命令長を含め得る。さらに、標準命令部分５１０を、標準命令４６０および４６５と同一の標準命令デコーダを使用して処理し得る。１実施形態で、標準命令部分５１０が、標準命令部分５１０が標準命令４６０および４６５に関連して処理される順序でメモリ５０５にストアされる。たとえば、標準命令部分５１０は、標準命令４６０が処理された後、標準命令４６５が処理される前に処理され得る。１実施形態で、標準命令部分５１０および増補命令部分５１５は、プログラム・ループに関連する。ループ初期化命令５１１を使用して、ループ実行および／または増補命令処理を初期化し得る。

【０２１９】

増補命令部分５１５は、標準命令部分５１０に関連し、関連する標準命令部分５１０と時間的に並行して処理される。しかし、増補命令部分５１５は、標準命令部分５１０と別々のメモリのブロックにストアされる。たとえば、図示の実施形態で、増補命令部分５１５が、メモリ５０５内で、標準命令４６５の後にストアされる。増補命令レジスタ５２０が、増補命令部分５１５へのポインタを提供する。１実施形態で、増補命令レジスタ５２０が、レジスタ・ファイルにストアされる複数のレジスタの一部である。増補命令レジスタ５２０へのポインタを、命令を介して提供し得る。たとえば、ループ初期化命令５１１に、増補命令レジスタ５２０をポイントする命令フィールドを含め得る。その代わりに、ループ初期化命令５１１などの命令を使用して、増補命令レジスタ５２０のポインタを初期化して、増補命令部分５１５など、次に処理される増補命令部分の組の位置を提供し得る。

【０２２０】

増補命令レジスタ５２０を使用して、標準命令部分５１０の関連する標準コードのとの並列処理のために、増補命令部分５１５の増補コードにアクセスし得る。たとえば、ｓｔａｎｄａｒｄ＿ｃｏｄｅ＿１が標準命令デコーダによってアクセスされる時に、副デコーダ４４０（図２７）などの副命令デコーダによってａｕｇｍｅｎｔｅｄ＿ｃｏｄｅ＿１に

アクセスし得る。増補命令レジスタ５２０によって提供されるポインタを使用して、次に処理される増補命令部分のメモリ・アドレスを更新し得る。したがって、Ｋ個の増補命令部分のそれぞれを、増補命令レジスタ５２０のポインタを介して、関連する標準命令部分と並列に処理し得る。

【０２２１】

１実施形態で、増補命令レジスタ５２０に、各増補命令部分のポインタが含まれる。代替案では、増補命令レジスタ５２０を、増補命令部分５１５の最初の増補命令部分ａｕｇｍｅｎｔｅｄ＿ｃｏｄｅ＿１だけに初期化し得る。ａｕｇｍｅｎｔｅｄ＿ｃｏｄｅ＿１にアクセスした後に、増補命令レジスタ５２０に関連するメモリ・ポインタを、ａｕｇｍｅｎｔｅｄ＿ｃｏｄｅ＿２をポイントするように増分し得る。同様に、増補命令レジスタ５２０の単一ポインタによって、Ｋ個のすべての増補命令部分へのアクセスを提供し得る。

【０２２２】

図２９を参照すると、本発明の１実施形態による、メモリ内で増補命令を提供する形を示すブロック図が示されている。命令のプログラムが、メモリ６０５にストアされる。命令のプログラムに、標準命令４６０、ループ初期化命令６１０、標準ループ命令６２０、および標準命令４６５が含まれる。１実施形態で、命令４６０、６２０、および４６５の各部分に、同一の固定命令長が含まれる。ループ初期化命令６１０は、命令４６０、６２０、および４６５と同一の命令長を有する標準初期化命令部分６１５および増補命令部分６１７の両方を含むように拡張される。

【０２２３】

ループ初期化命令６１０は、標準命令４６０および４６５と比較して、拡張された命令長を有する増補命令として提供される。標準初期化部分６１５を使用して、標準ループ命令６２０の実行の初期化を提供し得る。たとえば、標準初期化命令部分６１５を使用して、増補命令部分６１７の処理をイネーブルすることが可能であり、あるいは標準ループ命令６２０のループ実行またはＰＶＡ実行をイネーブルすることが可能である。他の形のプログラム・ループ命令を、本発明の範囲から逸脱せずに標準命令４６０または４６５に含め得ることに留意されたい。

【０２２４】

増補命令部分６１７を使用して、追加の処理機能を指定し得る。たとえば、増補命令部分６１７によって、標準ループ命令６２０のコード部分に関して実行される追加処理を指定し得る。たとえば、増補命令部分６１７のビットの第１の組によって、標準ループ命令６２０のｓｔａｎｄａｒｄ＿ｃｏｄｅ＿１の処理の早期終条件を指定し得る。さらに、増補命令部分６１７のビットの第２の組を使用して、標準ループ命令６２０のｓｔａｎｄａｒｄ＿ｃｏｄｅ＿２の処理を早期終了するように指定し得る。１実施形態で、増補命令部分６１７でエンコードされたビット・マスクのＫ個の組を使用して、標準ループ命令６２０の個々の命令の早期終了機能をイネーブルし、かつ／または指定し得る。たとえば、増補命令部分６１７のビットの第１の組によって、標準ループ命令６２０のｓｔａｎｄａｒｄ＿ｃｏｄｅ＿１での早期終了を指定し得る。ビットの第１の組に、早期終了をイネーブルするイネーブル・ビットと、早期終了をトリガする条件を指定する条件ビットを含め得る。同様に、増補命令部分６１７のビットの第２の組によって、ｓｔａｎｄａｒｄ＿ｃｏｄｅ２など、標準ループ命令６２０の別の命令の早期終了を指定し得る。

【０２２５】

代替案では、増補命令部分６１７を使用して、標準ループ命令６２０のそれぞれに早期継続機能を増補し得る。１実施形態で、増補命令部分６１７でエンコードされるビット・マスクのＫ個の組を使用して、標準ループ命令６２０の個々の命令の早期継続機能をイネーブルし、かつ／または指定する。

【０２２６】

他の機能性を、増補命令部分６１７と共に提供することも可能であることに留意されたい。たとえば、増補命令部分６１７を使用して、標準ループ命令６２０のデスティネーション・オペランドを指定し得る。

【０２２７】
もう１つの例で、増補命令部分６１７を使用して、標準ループ命令６２０によって使用される様々な入力データ・ストリームおよび出力データ・ストリームをイネーブルするためなど、マスク・ベクトルを提供し得る。ループ初期化命令６１０以外の命令を使用して、増補命令部分６１７を提供し得ることを諒解されたい。
【０２２８】
図３０を参照すると、本発明の１実施形態による、増補命令部分７２７をストアする形を示すブロック図が示されている。命令のプログラムが、メモリ７０５に保管され、この命令のプログラムには、標準命令４６０、ループ初期化命令７１１、増補命令７２０、および標準命令４６５が含まれる。標準命令４６０および４６５ならびにループ初期化命令７１１に、標準エンコーディングの命令と、同一の固定命令長が含まれる。増補命令７２０に、標準命令部分７２５および増補命令部分７２７が含まれる。１実施形態で、標準命令部分７２５に、同一の標準エンコーディングの標準コードと、標準命令４６０および４６５と同一の固定命令長が含まれ、標準命令部分７２５は、標準命令デコーダによって処理される。しかし、増補命令部分７２７は、副命令デコーダを使用して、標準命令部分７２５と時間的に並行して処理される。
【０２２９】
１実施形態で、ループ初期化命令７１１が、増補命令７２０に関連するループ実行モードをトリガするのに使用される。１実施形態で、ループ初期化命令７１１が、さらに、増補命令７２０の増補命令部分７２７を処理するのに使用される副命令デコーダをトリガするのに使用される。標準命令部分７２５は、標準命令４６０に関してそれらが処理される形でメモリにストアされ、標準命令４６０は、増補命令７２０および標準命令４６５の前に処理され、標準命令４６５は、増補命令７２０の後で処理される。
【０２３０】
増補命令部分７２７に、標準命令部分７２５のコードの機能を拡張するコードが含まれる。増補命令部分７２７のａｕｇｍｅｎｔｅｄ＿ｃｏｄｅ＿１は、標準命令部分７２５のｓｔａｎｄａｒｄ＿ｃｏｄｅ＿１に対応する。増補命令部分７２７のアクセスを単純にするために、増補命令部分を、メモリ７０５内の関連する標準命令部分７２５に隣接してストアし得る。たとえば、ａｕｇｍｅｎｔｅｄ＿ｃｏｄｅ＿１が、メモリ７０５内でｓｔａｎｄａｒｄ＿ｃｏｄｅ＿１の次にストアされ、ａｕｇｅｍｔｎｅｄ＿ｃｏｄｅ＿２が、ｓｔａｎｄａｒｄ＿ｃｏｄｅ＿２の次にストアされる。同様に、Ｋ個の増補命令部分のそれぞれが、めいめいの標準命令部分の次にストアされる。増補命令部分７２７を、標準命令部分７２５の次にストアされるものとして説明したが、増補命令部分７２７を、本発明の範囲から逸脱せずに各関連する標準命令部分の前に保管し得ることに留意されたい。
【０２３１】
前に述べたように、増補命令部分７２７は、標準命令部分７２５に関連する処理機能性を拡張するか指定し得る。たとえば、増補命令部分は、特定の標準命令部分に関連する条件実行条件をイネーブルし、指定することが可能であり、この条件実行では、特定の標準命令部分の処理が、増補命令部分に関連する条件に基づいて条件的に実行される。増補命令部分は、特定の標準命令部分に関連する早期継続条件または早期終了条件を指定することも可能である。
【０２３２】
図３１を参照すると、本発明の１実施形態による、増補命令に関連するフィールドを示すブロック図が示されている。増補命令８００などの増補命令が、あるプログラム・ループの命令に追加機能性を提供するのに使用される。増補命令８００は、標準部分８１０および増補部分８２０に分割される。増補部分８２０には、プログラム・ループ内の標準命令部分８１０を処理する早期継続機能および早期終了機能を指定するビットE_T、C_T、E_C、およびC_Cが含まれる。
【０２３３】
システム４００（図２７）など、増補命令８００の処理に使用されるシステムに関連す

る条件コードは、システムによって処理されるコマンドに基づいて変更される。たとえば、「ｃｏｍｐｌｔ」命令をシステムによって処理して、２つの別々のレジスタ値を比較し得る。その比較に基づいて、条件コードを変更して、真の状態または偽の状態に影響を及ぼす。１実施形態で、処理システムが、条件コードに、現在の状態を示すために「１」または「０」のいずれかの値をセットする。代替実施形態では、複数の条件コード、あるいは複数のビットまたは値を有する条件コードが、存在することが可能であり、増補命令部分８２０によって指定され得る。

【０２３４】

条件コードの現在値に基づいて、増補命令８００は、増補命令８００に関連するプログラム・ループの処理に影響を及ぼし得る。前に述べたように、標準命令デコーダを使用して、標準命令部分８１０を処理することが可能であり、副命令デコーダを使用して、増補命令部分８２０を処理することが可能である。増補命令部分８２０は、早期終了条件を確立し得る。早期終了条件を用いると、処理システム４００が、増補命令８００に関連するプログラム・ループのさらなる処理を取り消し得るようになる。早期終了イネーブル・ビットE_Tを使用して、増補命令８００の早期終了機能をイネーブルし得る。早期終了条件ビットC_Tをセットして、「０」または「１」のいずれかの値にセットされることによって特定の論理条件を示し得る。早期終了ビットE_Tがイネーブル状態にセットされ、早期終了条件ビットC_Tが、処理システムの条件コードの値と等しい場合に、プログラム・ループのさらなる実行が停止される。標準命令デコーダが、標準命令部分８１０を処理したならば、次に処理される命令は、プログラム・ループの外の次の命令である。

【０２３５】

早期終了に似て、早期継続ビットE_CおよびC_Cの組によって、早期継続条件を示し得る。早期継続イネーブル・ビットE_Cを使用して、増補命令８００に関連する早期継続をイネーブルし得る。早期継続イネーブル・ビットE_Cに、イネーブル状態がセットされ、早期継続条件ビットC_Cの値がシステムの条件コードの値と同等である場合に、システムは、プログラム・ループの現在の反復でのさらなる命令の処理を停止し、プログラム・ループに関連する次の反復に継続し得る。

【０２３６】

他の機能も、増補命令部分８２０に似た増補命令部分の使用を介して提供し得る。たとえば、増補命令部分によって、増補命令の条件実行機能を指定し得る。図３２を参照すると、本発明の１実施形態による、条件実行機能を提供する増補命令に関連するフィールドを示すブロック図が示されている。増補命令７５０などの増補命令を使用して、一部のプログラム・ループの命令に追加機能を提供する。増補命令７５０は、標準命令部分７５２および増補命令部分７５１に分割される。

【０２３７】

増補命令部分７５１に、条件実行制御フィールド７５５が含まれる。条件実行制御フィールド７５５に、条件実行のイネーブル・ビットE_Xおよび条件実行の条件コードC_Xが含まれる。条件実行のイネーブル・ビットE_Xによってイネーブルされる時に、増補命令７５０の実行が、条件実行の条件コードC_Xの値に基づいて条件的に行われる。条件実行の条件コードC_Xによって指定される条件が満たされる時に、標準命令部分７５２の実行が実行される。しかし、条件実行の条件コードC_Xによって指定される条件が満たされない時には、標準命令部分７５２の実行が実行されず、増補命令７５０は、効果的に「ノーオペレーション」命令として扱われる。条件実行制御フィールド７５５を提供することによって、標準命令部分７５２の通常の実行を、有利に条件的にし得る。代替実施形態で、条件実行制御フィールド７５５を拡張して、複数の条件コードおよび／または複数のビットを有する条件コード・フィールドを含め得ることに留意されたい。

【０２３８】

増補命令の増補命令部分を使用して、標準命令部分の処理の追加オペランドを提供することも可能である。１実施形態で、標準命令部分が、２オペランド命令であり、算術演算などの動作が、２つのソース・オペランドからの値を使用して実行される。普通のシステ

ムでは、２オペランド命令の処理の結果は、ソース・オペランドの１つにストアされ、そのソース・オペランドの元の値が破壊される。その代わりに、増補命令部分を使用して、第１オペランドおよび第２オペランドの処理からの値を返し得る第３オペランドを提供し得る。

【０２３９】

図３３を参照すると、本発明の１実施形態による、追加オペランドを含む増補命令に関連するフィールドを示すブロック図が示されている。増補命令８３０などの増補命令を使用して、あるプログラム・ループの命令に追加機能性を提供する。増補命令８３０は、標準命令部分８５０および増補命令部分８４０に分割される。増補命令部分８４０には、追加オペランド指定子Ｒｃが含まれ、このＲｃを使用して、前に述べたように第３オペランドを指定し得る。この第３オペランド値は、標準命令部分８５０の結果を受け取るデスティネーション・レジスタ指定子として使用し得る。追加オペランド指定子Ｒｃを提供することによって、標準命令部分８５０の通常の破壊的動作を有利に防ぎ得る。

【０２４０】

増補命令部部分を使用して、特定の標準命令部分の即値フィールドを指定することも可能である。したがって、標準命令部分によって、増補命令部分によって指定される即値フィールドをソース・オペランドとして使用し得る。その代わりに、増補命令部分を使用して、標準命令部分で指定される即値フィールドの長さを拡張し得る。

【０２４１】

図３４を参照すると、本発明の１実施形態による、即値フィールドを提供する増補命令に関連するフィールドを示すブロック図が示されている。増補命令８６０などの増補命令を使用して、あるプログラム・ループの命令に追加機能性を提供する。増補命令８６０は、標準命令部分８７０および増補命令部分８８０に分割される。増補命令部分８８０に、即値フィールド８８１が含まれ、即値フィールド８８１は、ソース・オペランドとして使用するか、これを使用して標準命令部分８７０で既に指定されている即値フィールドの長さを拡張し得る。拡張として即値フィールド８８１を提供することによって、標準命令部分８７０の即値の制限された範囲を有利に回避し得る。

【０２４２】

本発明の１実施形態では、図３２、３３、および３４に関して説明した機能を、増補命令部分をエンコードすることによって組み合わせ、選択し得る。図３５を参照すると、ブロック図に、増補命令９００に関連するエンコーディングおよび機能の例が示されている。増補命令９００は、標準命令部分９１０および増補命令部分９２０からなる。増補命令９３０の増補命令部分は、即値フィールド９３２を備える。増補命令９４０の増補命令部分は、追加オペランド指定子Ｒｃを備える。増補命令９５０の増補命令部分は、早期継続／早期終了制御フィールド９５２を備える。増補命令９３０および９４０は、条件実行制御フィールド９３４および９４４によって増補命令の条件実行を指定する機能も備える。

【０２４３】

図３５では、３つの追加機能が、増補命令部分９２０の上位ビット９３１、９４１、および９５１のエンコーディングによって区別される。図２７の副デコーダ４４０は、これらの上位ビットを検査して、これらの増補命令部分によって指定される追加機能を判定する。

【０２４４】

増補命令を使用して、標準命令幅を使用して可能ではないさらなる機能を定義する拡張命令幅を提供し得る。たとえば、前に図４で述べた実施形態は、データ・ストリーミング・サポートを提供した。図４で述べた実施形態では、２つの入力ストリーム、L_0 ６およびL_1 ８ならびに単一の出力ストリームＳ３６だけがサポートされた。しかし、増補命令を使用すると、データ・ストリーミング・マスク・ベクトルを使用し得る。データ・ストリーミング・マスク・ベクトルには、異なるデータ・ストリームをイネーブルするのに使用し得る複数のビットが含まれる。たとえば、データ・ストリーミング・マスク・ベクトルの第１データ・ビットを使用して、第１データ・ストリームをイネーブルすることが可

能であり、データ・ストリーミング・マスク・ベクトルの第２データ・ビットを使用して、第２データ・ストリームをイネーブルすることが可能である。１実施形態で、データ・ストリーミング・マスク・ベクトルが、ロード部分およびストア部分に分割され、ロード部分のビットは、入力データ・ストリームをイネーブルし、ストア部分のビットは、出力データ・ストリームをイネーブルする。データ・ストリーミング・マスク・ベクトルによって、複数の入力データ・ストリームおよび出力データ・ストリームを指定し得る。たとえば、１実施形態で、３２ビット・データ・ストリーミング・マスク・ベクトルによって、１６個までの入力データ・ストリームおよび１６個までの出力データ・ストリームをイネーブルし得る。

【０２４５】

１実施形態で、データ・ストリーム・マスク・ベクトルが、増補命令部分を介して指定される。たとえば、ループ初期化命令６１０（図２９）などのループ初期化命令に関連する増補命令部分によって、データ・ストリーミング・マスク・ベクトル値を提供して、複数のデータ・ストリームを同時に初期化し得る。各データ・ストリームを、アーキテクチャ的レジスタR_iに関連付け得る。アーキテクチャ的レジスタR_iからの読取または書込によって、関連する入力データ・ストリームからデータ要素が検索されるか、関連する出力データ・ストリームにデータ要素がストアされる。図１５に関して前に説明したように、アーキテクチャ的レジスタR_iに、データ・ストリームの次のデータ要素に関連し、ＰＶＡ実行中または非ループ実行中にのみ可視になる一時インスタンスと、スカラ実行中または非ループ実行中にのみ可視になるオーバーレイされるインスタンスが含まれる。

【０２４６】

本明細書で述べない他の機能を、本発明の範囲から逸脱せずに、増補命令の使用を介して追加または拡張することも可能であることを諒解されたい。

図３６を参照すると、本発明の１実施形態による、コマンドをコンパイルする方法を示す流れ図が示されている。コンパイラは、一般に、特定のプログラミング言語に従って記述されたコマンドを、コマンドを処理するのに使用される特定のプロセッサによって理解されるシステム・コードに変換するのに使用される。一般に、コンパイラは、特定のコマンドについて類似するコードを生成する。図示の実施形態では、コンパイラが、プログラム・ループに関連するコマンドを増補命令に変換する。したがって、コード密度は、大きくは影響されず、増補命令について前に述べた特徴を、プログラム・ループに関連するコマンドと共に使用し得る。

【０２４７】

ステップ９６０で、コンパイラが、次に処理されるコマンドを受け取る。コマンドは、Ｃ、Ｃ＋＋、Ｊａｖａ（登録商標）、ＢＡＳＩＣ、および類似物など、特定のプログラミング言語のコマンドに関連するものとし得る。ステップ９６２で、受け取ったコマンドがプログラム・ループに関連するかどうかを判定する。コンパイラは、分岐コマンド、ＤＯＵＮＴＩＬコマンド、またはＷＨＩＬＥコマンドなどのプログラム・ループを初期化するか処理する既知のコマンドにコマンドを関連付けることによって、コマンドを認識し得る。同様に、コマンドを、ＰＶＡ初期化命令に関連付け、コマンドがプログラム・ループに関連することを示し得る。

【０２４８】

ステップ９６４で、コマンドがプログラム・ループに関連しない場合に、コンパイラは、コマンドを標準命令に変換する。前に述べたように、標準命令は、一般にコンパイラによってシステム・コードを提供するのに使用される命令を表し、固定サイズまたは標準サイズであるものとし得る。標準命令が生成されたならば、コンパイラは、ステップ９６０に戻って新しいコマンドを検索し得る。

【０２４９】

ステップ９６６で、受け取ったコマンドがプログラム・ループに関連するものとして識別される場合に、コンパイラは、コマンドを増補命令に変換する。増補命令には、前に述べた標準命令に類似する標準命令部分と、増補命令部分が含まれる。増補命令を使用して

、前に述べたように、標準命令部分の処理に関する拡張プロパティを指定し得る。たとえば、増補命令部分によって、早期終了、早期継続、または条件実行の制御情報などの条件制御情報を指定し得る。その代わりに、増補命令部分によって、標準命令部分に関連するデータの処理に関するデータ・ストリーミングを指定し得る。増補命令部分を使用して、標準命令部分の処理に関する即値フィールドまたは即値フィールド拡張を提供することも可能である。本発明の範囲から逸脱せずに、他の機能を使用することも可能である。増補命令が、コマンドについて生成されたならば、コンパイラは、ステップ９６０に戻って、処理される新しいコマンドを検索し得る。１実施形態で、標準命令および標準命令部分は、標準デコーダ４３０（図２７）などの標準デコーダによって処理されるが、増補命令部分は、副デコーダ４４０（図２７）などの副デコーダを使用して処理される。

【０２５０】

前述の明細書で、本発明を、特定の実施形態に関して説明した。しかし、当業者は、請求項に示された本発明の範囲から逸脱せずに、様々な修正および変更を行い得ることを諒解するであろう。たとえば、本明細書で教示されたソフトウェアを、コンピュータ・ハード・ディスク、フロッピ・ディスク、８８．９ｍｍ（３．５インチ）ディスク、コンピュータ記憶テープ、磁気ドラム、スタティック・ランダム・アクセス・メモリ（ＳＲＡＭ）セル、ダイナミック・ランダム・アクセス・メモリ（ＤＲＡＭ）セル、電気的消去可能（ＥＥＰＲＯＭ、ＥＰＲＯＭ、フラッシュ）セル、不揮発性セル、強誘電体メモリまたは強磁性体メモリ、コンパクト・ディスク（ＣＤ）、レーザ・ディスク、光ディスク、および類似するコンピュータ可読媒体の１つまたは複数で実施し得る。したがって、本明細書および図面は、制限的な意味ではなく例示的な意味であると考えられ、そのような修正のすべてが、本発明の範囲に含まれることが意図されている。

【図面の簡単な説明】

【０２５１】

【図１】本発明の１実施形態による、データ処理システムの実行モードを概略図形式で示す図。

【図２】ＤＳＰプロセッサ内の機能ユニットの従来技術の動作をブロック図形式で示す図。

【図３】真のベクトル計算機の機能ユニットの従来技術の動作をブロック図形式で示す図。

【図４】本発明の１実施形態による、擬似ベクトル算術演算の処理に適合された処理システムをブロック図形式で示す図。

【図５】本発明の１実施形態による、ＣＶＡ（ｃａｎｏｎｉｃａｌ　ｖｅｃｔｏｒ　ａｒｉｔｈｍｅｔｉｃ）の３つのタイプのデータ依存性グラフをデータ流れ図形式で示す図。

【図６】本発明の１実施形態による、図４の処理システムのうちでＣＶＡ処理に関する部分の構成をブロック図形式で示す図。

【図７】本発明の１実施形態による、ＣＶＡ命令のフォーマットをブロック図形式で示す図。

【図８】本発明の１実施形態による、ＰＶＡ（ｐｓｅｕｄｏ－ｖｅｃｔｏｒ　ａｒｉｔｈｍｅｔｉｃ）命令のフォーマットをブロック図形式で示す図。

【図９】本発明の１実施形態による、ＰＶＡ命令を使用して構成されるプログラム・ループの構造をプログラム流れ図形式で示す図。

【図１０】本発明の１実施形態による、ストライド・サイズ・レジスタ（ＳＳＲ）のフォーマットをブロック図形式で示す図。

【図１１】本発明の１実施形態による、カウント・インデックス・レジスタ（ＣＩＲ）のフォーマットをブロック図形式で示す図。

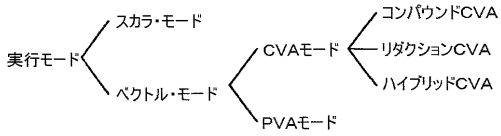
【図１２】本発明の１実施形態による、ロード・ユニットL_0の１実施形態をブロック図形式で示す図。

【図１３】本発明の１実施形態による、ロード・ユニットL_1の１実施形態をブロック図形式で示す図。

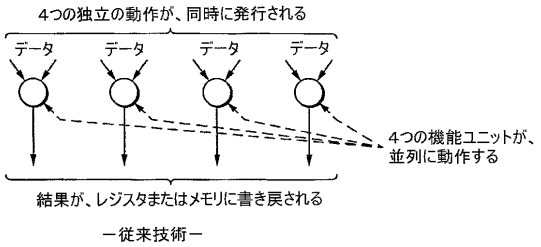
【図１４】本発明の１実施形態による、ストア・ユニットＳの１実施形態をブロック図形式で示す図。
【図１５】本発明の１実施形態による、動作のあるモード中にオーバーレイされるレジスタを有するレジスア・ファイルをブロック図形式で示す図。
【図１６】本発明の１実施形態による、一時レジスタおよびオーバーレイされるレジスタを有するレジスタ・ファイルをブロック図形式で示す図。
【図１７】本発明の１実施形態による、一時レジスタおよびオーバーレイされるレジスタの対応をタイミング図形式で示す図。
【図１８】本発明の１実施形態による、プログラム・ループ実行を管理するカウンタをブロック図形式で示す図。
【図１９】本発明の１実施形態による、コンピュータ・プログラム・コードを処理する判断フローを流れ図形式で示す図。
【図２０】本発明の１実施形態による、早期ループ継続機能を有するＰＶＡ命令の代替実施形態をブロック図形式で示す図。
【図２１】本発明の代替実施形態による、擬似ベクトル計算機をブロック図形式で示す図。
【図２２】本発明の１実施形態による、図２１の擬似ベクトル計算機の依存性グラフをデータ流れ図形式で示す図。
【図２３】本発明の代替実施形態による、ＰＶＡ命令のフォーマットをブロック図形式で示す図。
【図２４】本発明の代替実施形態による、ＰＶＡ命令を使用して構成されるプログラム・ループの構造をプログラム流れ図形式で示す図。
【図２５】本発明の代替実施形態による、ＰＶＡ命令のフォーマットをブロック図形式で示す図。
【図２６】本発明の代替実施形態による、ＰＶＡ命令を使用して構成されるプログラム・ループの構造をプログラム流れ図形式で示す図。
【図２７】本発明の１実施形態による、標準命令および増補命令の標準命令部分を処理する第１デコーディング部分と、増補命令の増補命令部分を処理する第２デコーディング部分とを有するシステムをブロック図形式で示す図。
【図２８】本発明の１実施形態による、レジスタと結合された増補命令が、増補命令に関連する増補命令部分の組へのポインタを提供するメモリの構造をブロック図形式で示す図。
【図２９】本発明の１実施形態による、ループ初期化命令に標準初期化命令部分および増補命令部分が含まれるメモリの構造をブロック図形式で示す図。
【図３０】本発明の１実施形態による、増補命令がメモリ内の標準ループ命令に結合される命令のプログラムの構造をブロック図形式で示す図。
【図３１】本発明の１実施形態による、標準命令部分と、増補命令に関連する早期終了機能および早期継続機能を提供するのに使用される増補命令部分とを有する増補命令の構造をブロック図形式で示す図。
【図３２】本発明の１実施形態による、標準命令部分と、増補命令に関連する条件実行機能を提供するのに使用される増補命令部分とを有する増補命令の構造をブロック図形式で示す図。
【図３３】本発明の１実施形態による、標準命令部分と、増補命令に関連する追加オペランド機能を提供するのに使用される増補命令部分とを有する増補命令の構造をブロック図形式で示す図。
【図３４】本発明の１実施形態による、標準命令部分と、増補命令に関連する拡張即値機能を提供するのに使用される増補命令部分とを有する増補命令の構造をブロック図形式で示す図。
【図３５】本発明の１実施形態による、増補命令に関連する命令エンコーディングの例をブロック図形式で示す図。

【図３６】本発明の１実施形態による、コンパイリング・コマンドの方法を流れ図形式で示す図。

【図１】

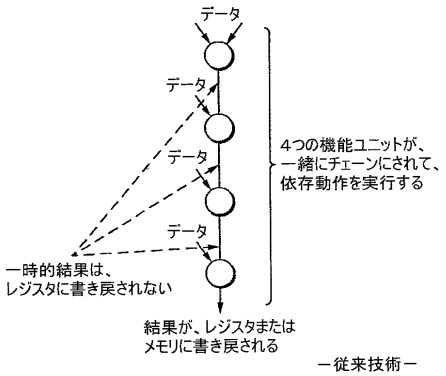


【図２】



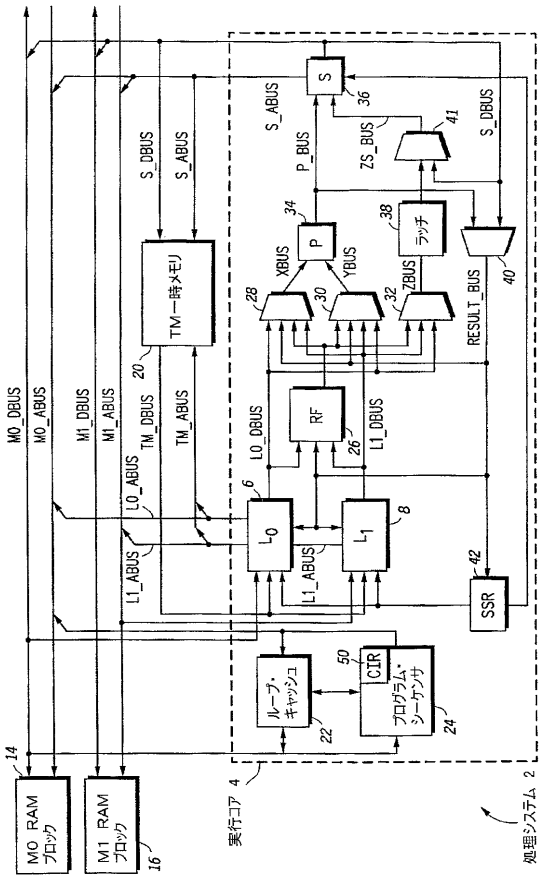
－従来技術－

【図３】

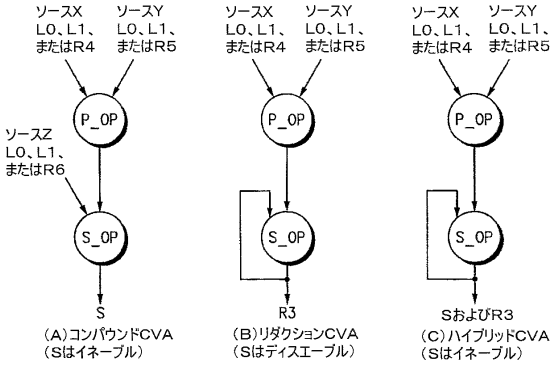


－従来技術－

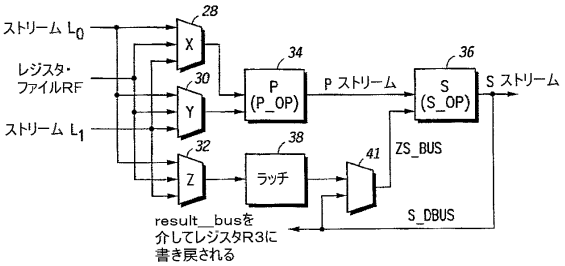
【図４】



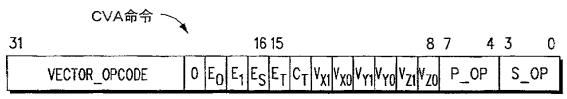
【図５】



【図６】



【図７】



VECTOR_OPCODE: ベクトル命令のオペコード

$E_0/E_1/E_S$: それぞれデータ・ストリームL0、L1、およびSのビットをイネーブルする

V_{X0}/V_{X1}: 次のように定義される。

- $V_{X1}V_{X0}$=00 オペランドXはR3からソーシングされる
- $V_{X1}V_{X0}$=01 オペランドXはL0からソーシングされる
- $V_{X1}V_{X0}$=10 オペランドXはL1からソーシングされる
- $V_{X1}V_{X0}$=11 オペランドXは0からソーシングされる

V_{Y0}/V_{Y1}: 次のように定義される。

- $V_{Y1}V_{Y0}$=00 オペランドYはR4からソーシングされる
- $V_{Y1}V_{Y0}$=01 オペランドYはL0からソーシングされる
- $V_{Y1}V_{Y0}$=10 オペランドYはL1からソーシングされる
- $V_{Y1}V_{Y0}$=11 オペランドYは0からソーシングされる

V_{Z0}/V_{Z1}: 次のように定義される。

- $V_{Z1}V_{Z0}$=00 オペランドZはR5からソーシングされる(コンパウンドCVA)
- $V_{Z1}V_{Z0}$=01 オペランドZはL0からソーシングされる(コンパウンドCVA)
- $V_{Z1}V_{Z0}$=10 オペランドZはL1からソーシングされる(コンパウンドCVA)
- $V_{Z1}V_{Z0}$=11 オペランドZは0からソーシングされる (リダクションおよびハイブリッドCVA)

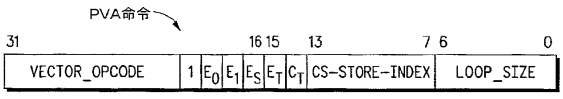
E_T: 早期終了のイネーブル・ビット

C_T: 早期終了の条件コード

P_OP: 主算術のオペコード

S_OP: 副算術のオペコード

【図８】



VECTOR_OPCODE: ベクトル命令のオペコード

$E_0/E_1/E_S$: それぞれデータ・ストリームL0、L1、およびSのビットをイネーブルする

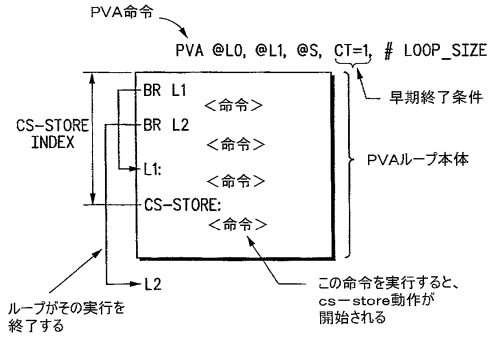
E_T: 早期終了のイネーブル・ビット

C_T: 早期終了の条件コード

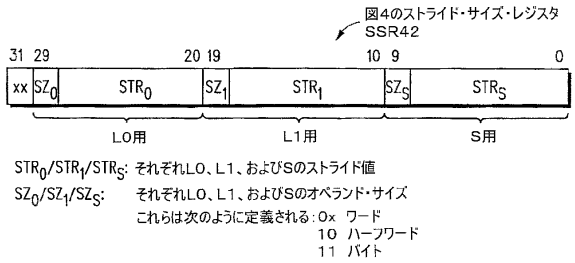
CS-STORE-INDEX: cs−store動作を開始する、ループ本体内の命令のインデックス

LOOP_SIZE: 命令数単位のループ本体のサイズ

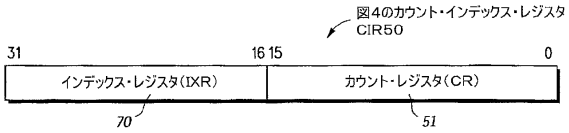
【図９】



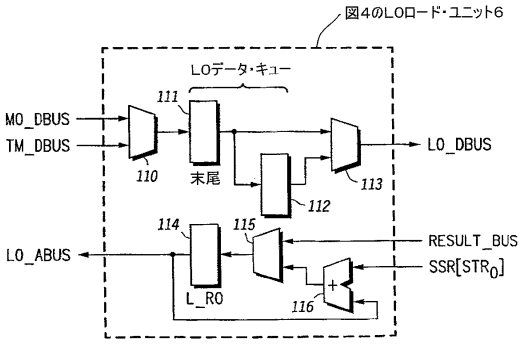
【図１０】



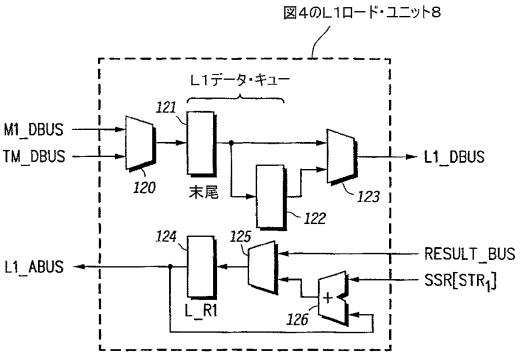
【図１１】



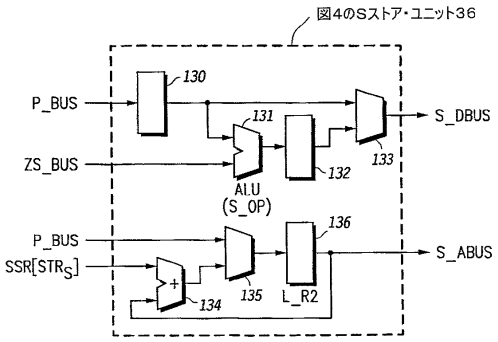
【図１２】



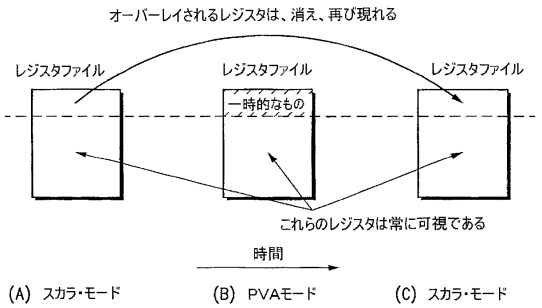
【図１３】



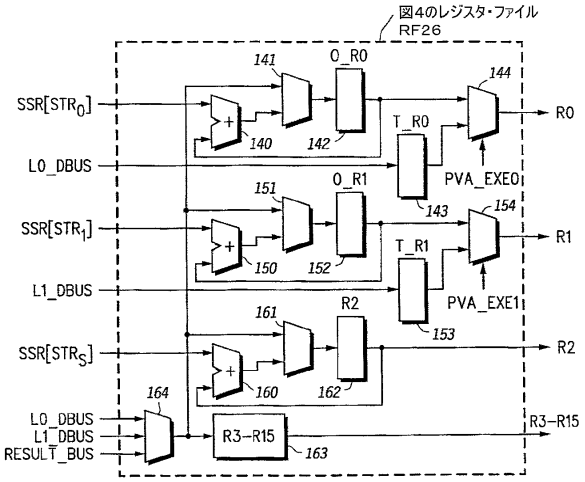
【図１４】



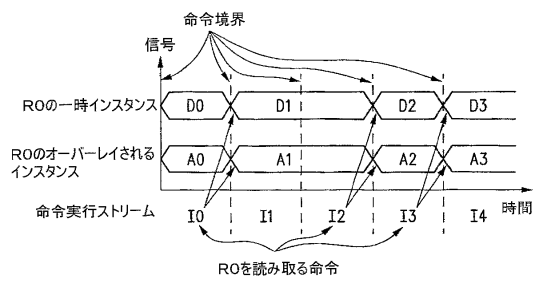
【図１５】



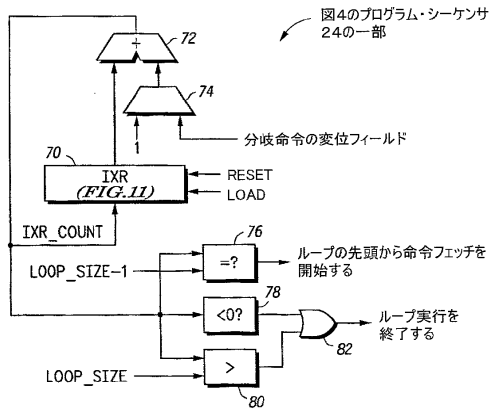
【図１６】



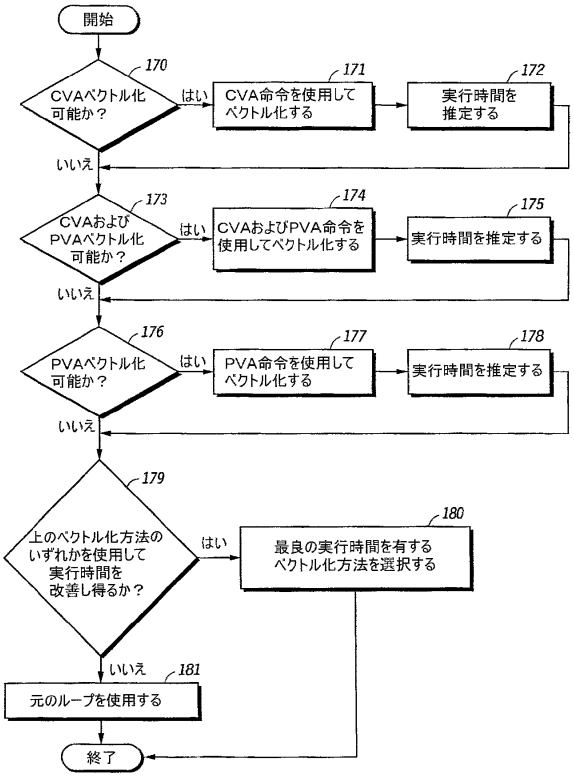
【図１７】



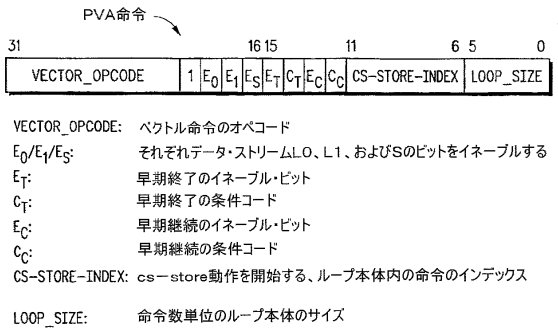
【図１８】



【図１９】



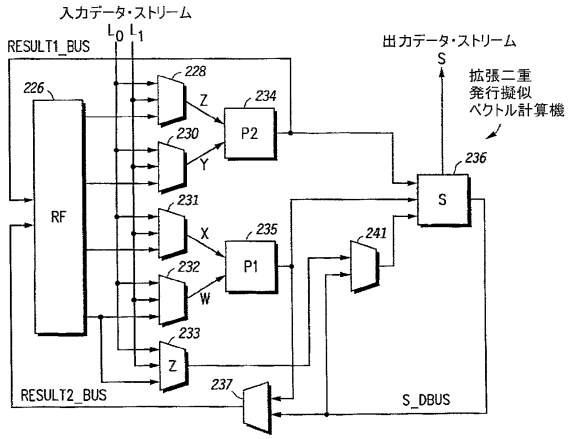
【図２０】



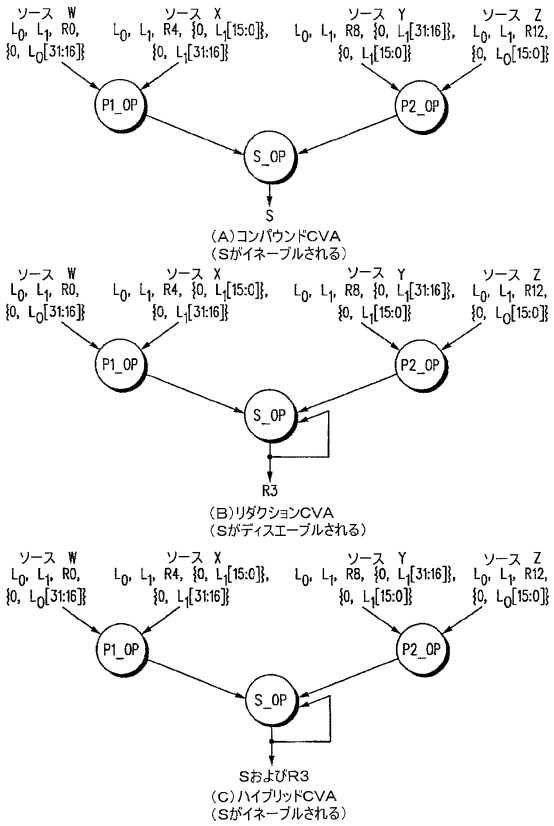
VECTOR_OPCODE: ベクトル命令のオペコード
$E_0/E_1/E_S$: それぞれデータ・ストリームL0、L1、およびSのビットをイネーブルする
E_T: 早期終了のイネーブル・ビット
C_T: 早期終了の条件コード
E_C: 早期継続のイネーブル・ビット
C_C: 早期継続の条件コード
CS-STORE-INDEX: cs－store動作を開始する、ループ本体内の命令のインデックス

LOOP_SIZE: 命令数単位のループ本体のサイズ

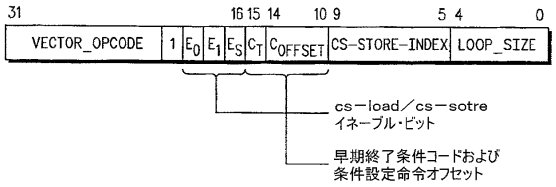
【図２１】



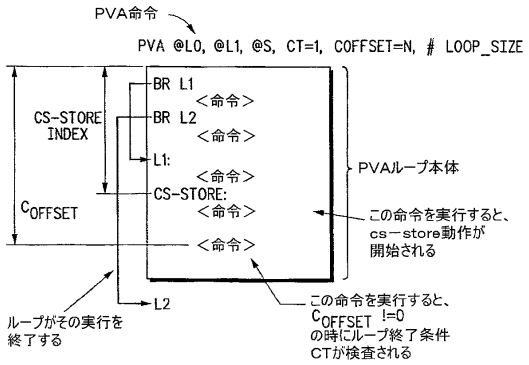
【図２２】



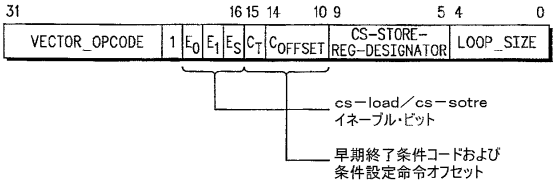
【図２３】



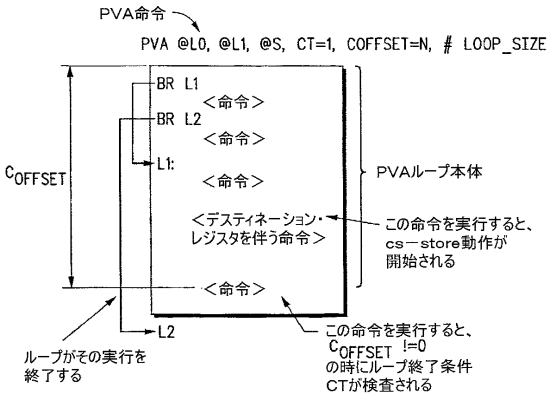
【図２４】



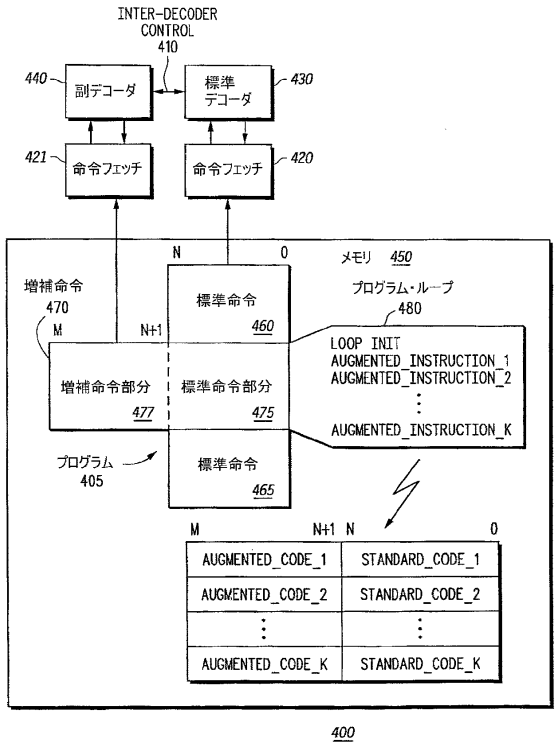
【図２５】



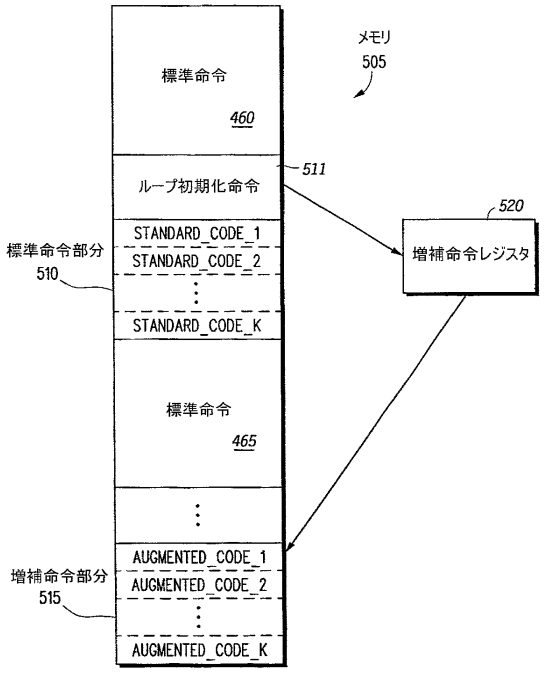
【図２６】



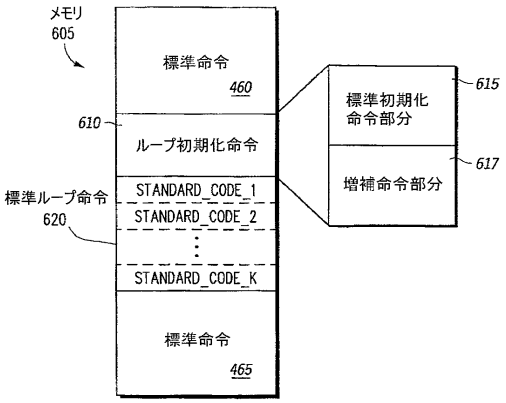
【図２７】



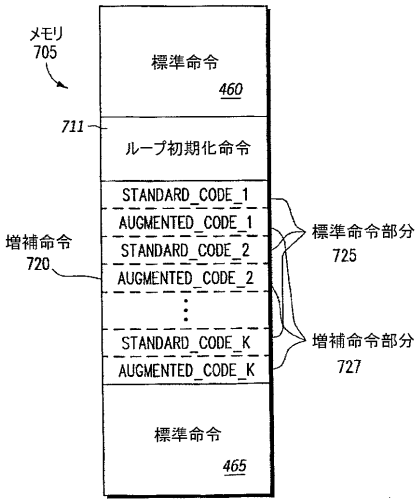
【図２８】



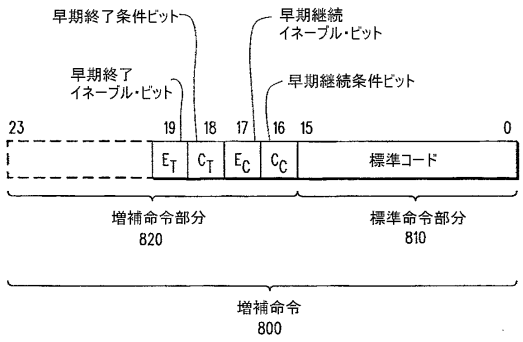
【図２９】



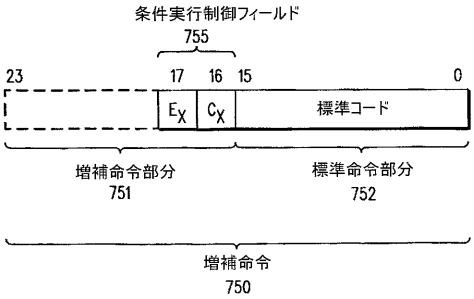
【図３０】



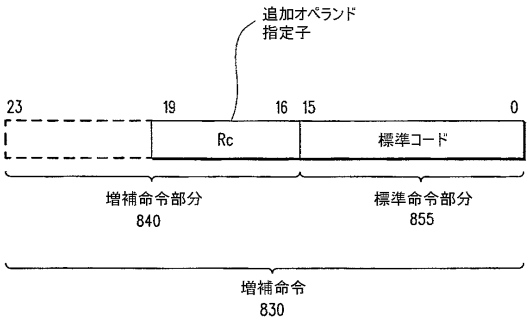
【図３１】



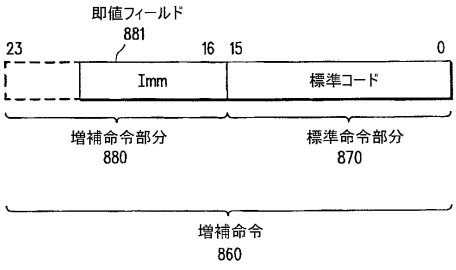
【図３２】



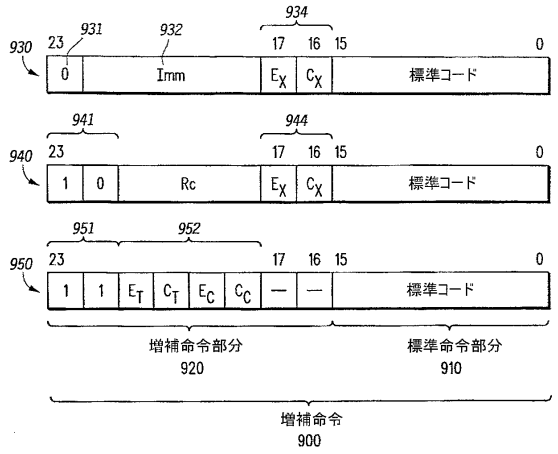
【図３３】



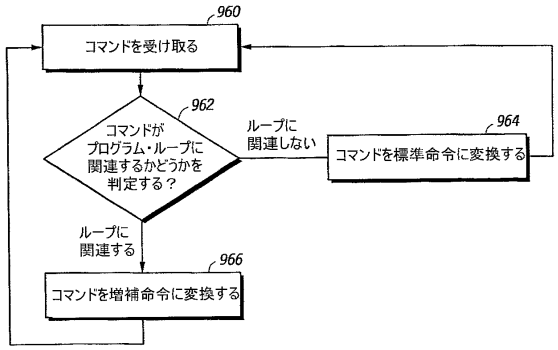
【図３４】



【図３５】


930
931
932
934
23
17
16
15
0
Imm
標準コード
940
941
944
Rc
950
951
952
増補命令部分
920
標準命令部分
910
増補命令
900

【図３６】


960
コマンドを受け取る
962
コマンドが
プログラム・ループに
関連するかどうかを
判定する？
ループに
関連しない
964
コマンドを標準命令に変換する
ループに
関連する
966
コマンドを増補命令に変換する

フロントページの続き

(72)発明者　モイヤー、ウィリアム　シー.
アメリカ合衆国　７８６２０　テキサス州　ドリッピング　スプリングス　ピア　ブランチ　ロード　１００５

審査官　石川　正二

(56)参考文献　特開平０５－０４０６３０（ＪＰ，Ａ）
特開平０７－１５２５５７（ＪＰ，Ａ）

(58)調査した分野(Int.Cl.，ＤＢ名)
G06F 17/16
G06F 9/38
G06F 9/45