(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号
特許第3645574号
(P3645574)
(45)発行日　平成17年5月11日(2005.5.11)　(24)登録日　平成17年2月10日(2005.2.10)

(51)Int.Cl.[7]　F I
G06F 9/305　G06F 9/30 340E
G06F 9/38　G06F 9/38 370X

請求項の数 10　(全 63 頁)

(21)出願番号　特願平10-514373
(86)(22)出願日　平成9年8月22日(1997.8.22)
(65)公表番号　特表2001-501001(P2001-501001A)
(43)公表日　平成13年1月23日(2001.1.23)
(86)国際出願番号　PCT/GB1997/002260
(87)国際公開番号　WO1998/012627
(87)国際公開日　平成10年3月26日(1998.3.26)
審査請求日　平成15年9月2日(2003.9.2)
(31)優先権主張番号　9619826.2
(32)優先日　平成8年9月23日(1996.9.23)
(33)優先権主張国　英国(GB)

(73)特許権者
エイアールエム　リミテッド
イギリス国シービー１　９エヌジェイ　ケンブリッジ、チェリー　ヒントン，フルバーン　ロード　１１０
(74)代理人
弁理士　浅村　皓
(74)代理人
弁理士　浅村　肇
(74)代理人
弁理士　林　鈺三
(74)代理人
弁理士　清水　邦明

最終頁に続く

(54)【発明の名称】データ処理システムにおける入力オペランド制御

(57)【特許請求の範囲】
【請求項１】
操作されるべきデータワードを記憶するための、それぞれが少なくともＮビット容量持つ複数のレジスタ（10）と、
プログラム命令ワードに応答して前記プログラム命令ワードが指定する演算を行う演算ユニット（20,22,24）と
を備えるデータ処理装置であって、前記演算ユニットは、
（ｉ）前記プログラム命令ワードに対して入力オペランド・データワードを記憶する前記複数のレジスタのソースレジスタを指定するソースレジスタビット領域（Register Number）、
を含む、少なくとも１つのプログラム命令ワードに応答し、更にプログラム命令ワードの少なくとも１つは
（ii）前記入力オペランドデータワードがＮビットサイズであるか（N/2）ビットサイズであるかを指定する入力オペランドサイズ・フラグ（size）と、
（iii）上位／下位フラグ（Hi/Lo）であって、
前記入力サイズ・フラグが（N/2）ビットサイズを指定する場合に、前記ソースレジスタの上位ビット位置と前記ソースレジスタの下位ビット位置のどちらに前記入力オペランドデータワードがあるかを指示し、
前記入力サイズフラグがＮビットサイズを指定する場合に、Ｎビット入力オペランドデータワードとして使用する前に、前記上位ビット位置に記憶されたビットを前記下位ビット

位置に移し、前記下位ビット位置に記憶されたビットを前記上位ビット位置に移すべきであるかどうかを指示する前記上位／下位フラグ
を含むことを特徴とするデータ処理装置。
【請求項２】
データ記憶デバイスと前記複数のレジスタとの間でデータワードをやり取りするためのＮビットデータバスを備えることを特徴とする請求項１に記載の装置。
【請求項３】
前記Ｎビットデータバスからデータワードを受け取り、前記Ｎビットデータワードを前記複数のレジスタに供給するための入力バッファを備えることを特徴とする請求項１に記載の装置。
【請求項４】
前記演算ユニットが、１つのソースレジスタの上位ビット位置と下位ビット位置にそれぞれ記憶されている第１（N/2）ビット入力オペランドデータワードと第２（N/2）ビット入力オペランドデータワードに対して別々の演算を行う少なくとも１つの並列操作プログラム命令ワードに応答することを特徴とする請求項１乃至３のいずれかに記載の装置。
【請求項５】
前記演算ユニットが、演算操作におけるビット位置の間のキャリー・チェインとして機能する信号パスを持ち、並列操作プログラム命令ワードを実行する時に、前記信号パスが前記第１（N/2）ビット入力オペランド・データワードと前記第２（N/2）ビット入力オペランド・データワードとの間で割れることを特徴とする請求項４に記載の装置。
【請求項６】
前記並列操作プログラム命令ワードは、
（ｉ）２つの並列（N/2）ビット加算が行われる並列加算、
（ii）２つの並列（N/2）ビット減算が行われる並列減算、
（iii）２つの並列（N/2）ビットシフトが行われる並列シフト、
（iv）（N/2）ビット加算と（N/2）ビット減算とが並列に行われる並列加算／減算
の１つの演算を行うことを特徴とする請求項４及び５のいずれかに記載の装置。
【請求項７】
前記演算ユニットがＮビットデータパスを備えることを特徴とする請求項１乃至６のいずれかに記載の装置。
【請求項８】
前記データパスの下位（M/2）ビットに供給するものとして、前記ソースレジスタの上位ビット位置または下位ビット位置のいずれかに記憶された（N/2）ビット入力オペランドデータワードを選択するための前記上位／下位フラグに応答するマルチプレクサを少なくとも１つ備えることを特徴とする請求項８に記載の装置。
【請求項９】
前記Ｎビットデータパス入力の前に、（N/2）ビット入力オペランドデータワードを符号拡張するための回路を備えることを特徴とする請求項８及び９のいずれかに記載の装置。
【請求項１０】
操作されるべきデータワードを、それぞれが少なくともＮビット容量持つ複数のレジスタに記憶するステップと、
プログラム命令ワードに応答して前記プログラム命令ワードが指定する演算を行うステップと
から成るデータ処理方法であって、
少なくとも１つのプログラム命令ワードに応答して、
（ｉ）前記プログラム命令ワード内のソースレジスタビット領域を使用して、前記プログラム命令ワードに対して入力オペランド・データワードを記憶する前記複数のレジスタのソースレジスタを選択し、その際、
（ii）前記プログラム命令ワード内の入力オペランドサイズフラグを使用して、前記入力オペランドデータワードがＮビットサイズであるか、あるいは（N/2）ビットサイズであ

るかを選択し、
（iii）前記プログラム命令ワード内の上位／下位フラグを使用して、前記入力サイズ・フラグが（N/2）ビットサイズを指定する場合に、前記ソースレジスタの上位ビット位置と前記ソースレジスタの下位ビット位置のどちらに前記入力オペランドデータワードがあるかを選択し、
前記入力サイズフラグがＮビットサイズを指定する場合に、Ｎビット入力オペランドデータワードとして使用する前に、前記上位ビット位置に記憶されたビットを前記下位ビット位置に移動し、前記下位ビット位置に記憶されたビットを前記上位ビット位置に移動すべきかどうかを前記上位／下位フラグが指示することを特徴とするデータ処理方法。
【発明の詳細な説明】
本発明は、データ処理システムに関する。特に、本発明はプログラム命令ワードの制御下で演算ユニットによって操作されるデータワードを記憶するための複数のレジスタを備えるデータ処理システムに関する。
公知のプログラム命令として、１つのレジスタに含まれる入力オペランドか、あるいは２つのレジスタに含まれるが単一入力オペランドとして扱われるべき入力オペランドのどちらに操作を実行するかを指定するものが提供されている。
米国特許US－Ａ－5,442,769は、物理レジスタの異なる部分を操作用に選択することのできる領域を備えるレジスタ領域を持つコプロセッサを開示している。
また欧州公開特許出願EP－Ａ－0,654,733は、ＮビットデータパスＬで２つのN/2ビット操作を並行して実行するマルチ・ゲージ処理を可能とするプロセッサを開示している。
本発明は、一面から見ると、
操作されるべきデータワードを記憶するための、それぞれが少なくともＮビット容量持つ複数のレジスタと、
プログラム命令ワードに応答して前記プログラム命令ワードが指定する演算を行う演算ユニットと
を備えるデータ処理装置であって、前記演算ユニットは、
（ｉ）前記プログラム命令ワードに対して入力オペランド・データワードを記憶する前記複数のレジスタのソースレジスタを指定するソースレジスタビット領域と、
（ii）前記入力オペランドデータワードがＮビットサイズであるか（N/2）ビットサイズであるかを指定する入力オペランドサイズ・フラグと、
（iii）前記入力サイズ・フラグが（N/2）ビットサイズを指定する場合に、前記ソースレジスタの上位ビット位置と前記ソースレジスタの下位ビット位置のどちらに前記入力オペランドデータワードがあるかを指示する上位／下位フラグと
を備え、前記入力サイズフラグがＮビットサイズを指定する場合に、前記上位／下位フラグは、Ｎビット入力オペランドデータワードとして使用する前に、前記上位ビット位置に記憶されたそれらのビットを前記下位ビット位置に移動させ且つ前記下位ビット位置に記憶されたそれらのビットを前記上位ビット位置に移動させるべきかどうかを指定する
少なくとも１つのプログラム命令ワードに応答することを特徴とするデータ処理装置を提供する。
最近、データ処理において、システムのデータパス幅を増加させる傾向がある。初期のシステムは、８ビットデータパスであった。それが、16ビットデータパスになり、現在では、32ビットや64ビットのデータパスが一般的になっている。このようなデータパスの増大に伴って、データ処理システム内のレジスタも相応の幅を持つべく増大してきた。本発明では、操作されるべきデータワードがデータパス幅より小さい場合、これらのワードを記憶するのにレジスタをフルに使用することはデバイス・リソースの無駄になると考えた。このことは、特に、操作されるべきデータをすべてレジスタに入れなければならないロード／記憶アーキテクチャ・マシンの場合、キャッシュメインのメモリからデータを取り出す回数を減らしたい時に問題となる。本発明は、このような事情に鑑みて、入力オペランドサイズ・フラグと上位／下位フラグを使用して、入力オペランドのサイズを示すと共にレジスタのどちらの部分に記憶されているかを示すことによって上述の問題を解決する。

このように、１つのレジスタが１つより多くの入力オペランドを保持できることによって、デバイスのレジスタリソースをより効率良く利用しながら、それらの入力オペランドを別々に操作することができる。さらに、前記入力サイズフラグがＮビットサイズを指定する場合、前記上位／下位フラグは、Ｎビット入力オペランドデータワードとして使用する前に、前記上位ビット位置に記憶されたビットを前記下位ビット位置に移し、前記下位ビット位置に記憶されたビットを前記上位ビット位置に移すべきかどうかを指定する。この特徴は、変換操作の時にとりわけ役に立つ。

本発明の利点は、Ｎビットのデータバスがデータ記憶デバイスをレジスタにリンクする場合、更に大きくなる。この場合、データバスを使用して２つのオペランドを一度に転送することによって、バスの帯域をより有効に利用することができ、性能のボトルネックを減らすことができる。

本発明の好ましい実施の形態において、前記演算ユニットは、１つのソースレジスタの上位ビット位置と下位ビット位置にそれぞれ記憶されている第１（N/2）ビット入力オペランドデータワードと第２（N/2）ビット入力オペランドデータワードに対して別々の演算を行う少なくとも１つの並列操作プログラム命令ワードに応答する。

並列操作プログラム命令ワードが提供されることによって、入力オペランドが最大データパス幅よりサイズが小さくても、Ｎビットのデータパス容量をフルに利用して２つの独立した演算を演算ユニットによって行うことができる。これにより、それほど余分なオーバーヘッドを招くことなしに、システムのデータ処理能力をかなり上げることができる。

１つの変形例として、前記演算ユニットは、演算操作におけるビット位置の間のキャリー・チェインとして機能する信号パスを持ち、並列操作プログラム命令ワードを実行する時に、前記信号パスは、前記第１（N/2）ビット入力オペランド・データワードと前記第２（N/2）ビット入力オペランド・データワードの間で割れる。

並列操作プログラム命令は、様々の形態を取ることができるが、前記並列操作プログラム命令ワードは、以下の１つの演算を行うことが好ましい。

（ｉ）２つの並列（N/2）ビット加算が行われる並列加算、

（ii）２つの並列（N/2）ビット減算が行われる並列減算、

（iii）２つの並列（N/2）ビットシフトが行われる並列シフト、

（iv）（N/2）ビット加算と（N/2）ビット減算とが並列に行われる並列加算／減算。

この機能のハードウェア導入で特に効果的なのは、前記ソースレジスタの上位ビット位置と下位ビット位置のいずれかに記憶された（N/2）ビット入力オペランドデータワードを選択して前記データパスの下位（N/2）ビットに供給するための前記上位／下位フラグに応答する少なくとも１つのマルチプレクサを備えた場合である。

符号付き演算を余分な複雑化なしに行うためには、前記Ｎビットデータパスへ入力する前に、（N/2）ビット入力オペランドデータワードを符号拡張するための回路を備えることが好ましい。

また、本発明は、他の面から見ると、

操作されるべきデータワードを、それぞれが少なくともＮビット容量持つ複数のレジスタに記憶するステップと、

プログラム命令ワードに応答して前記プログラム命令ワードが指定する演算を行うステップと

から成るデータ処理方法であって、

少なくとも１つのプログラム命令ワードに応答して、

（ｉ）前記プログラム命令ワード内のソースレジスタビット領域を使用して、前記プログラム命令ワードに対して入力オペランド・データワードを記憶する前記複数のレジスタのソースレジスタを選択し、その際、

（ii）前記プログラム命令ワード内の入力オペランドサイズフラグを使用して、前記入力オペランドデータワードがＮビットサイズであるか、あるいは（N/2）ビットサイズであるかを選択し、

（iii）前記プログラム命令ワード内の上位／下位フラグを使用して、前記入力サイズ・

フラグが（N/2）ビットサイズを指定する場合に、前記ソースレジスタの上位ビット位置と前記ソースレジスタの下位ビット位置のどちらに前記入力オペランドデータワードがあるかを選択し、
前記入力サイズフラグがＮビットサイズを指定する場合に、Ｎビット入力オペランドデータワードとして使用する前に、前記上位ビット位置に記憶されたビットを前記下位ビット位置に移動し、前記下位ビット位置に記憶されたビットを前記上位ビット位置に移すべきであるかどうかを前記上位／下位フラグが指示することを特徴とするデータ処理方法を提供する。
以下、本発明の実施の形態を例として、添付図面を参照して説明する。
図１は、デジタル信号処理装置のハイレベルの構成を示し、
図２は、コプロセッサ（coprocessor）のレジスタ構成の入力バッファを示し、
図３は、前記コプロセッサ内のデータパスを示し、
図４は、レジスタから高次または低次のビットを読みだすためのマルチプレキシング回路を示し、
図５は、好ましい実施の形態におけるコプロセッサが使用するレジスタ・リマッピング（remapping）論理を示すブロック図であり、
図６は、図５に示されたレジスタ・リマッピング論理の詳細を示し、
図７は、ブロック・フィルタ・アルゴリズム（Block Filter Algorithm）を示す表である。
以下に説明するシステムは、デジタル信号処理（DSP）に関する。DSPは、いろいろな形態を取ることができるが、典型的には、大量のデータの高速（実時間）処理を必要とする処理である。このデータは、典型的には、アナログの物理的信号である。DSPの好例として、デジタル移動電話に使用されるものがある。そこでは、無線信号が送受信され、アナログ音声信号から、及びアナログ音声信号へのデコーディング及びエンコーディング（典型的には、畳み込み（convolution）、変換、相関の操作を使用）が必要となる。また、他の例として、ディスクヘッドからの信号が処理されてヘッド・トラッキング制御が行われるディスク・ドライバ・コントローラが挙げられる。
上記のような文脈において、マイクロプロセッサ・コア（ここでは、英国、ケンブリッジのアドヴァンスト・RISC・マシン・リミテッドにより設計されたマイクロプロセッサの範囲からのARMコア）上でのデジタル信号処理システムの説明をする。マイクロプロセッサとコプロセッサ・アーキテクチャとの間のインターフェースは、それ自体が、DSP機能を提供すべく具体的構成を持つ。以下の説明において、マイクロプロセッサ・コアはARM、コプロセッサはピッコロ（Piccolo）とする。ARMとピッコロは、典型的には、他の構成要素（たとえば、チップ上のDRAM、ROM,D/Aコンバータ、A/Dコンバータ）をASICの部分として含む単一の集積回路として製造される。
ピッコロは、ARMのコプロセッサであるから、ARM命令の集合の一部を実行する。ARMコプロセッサ命令により、（Load Coprocessor,LDC and Store Coprocessor,STC命令を使用して）ARMがピッコロとメモリーの間でデータをやり取りさせ、また、（move to coprocessor,MCR,及び、move from coprocessor,MCR命令を使用して）ARMがARMレジスタをピッコロとやり取りすることができる。ある見方をすれば、ARMとピッコロの相互作用は、ARMGAピッコロのデータに対して強力なアドレス生成器として作用し、ピッコロの方は、大量のデータを実時間で扱う必要のあるDSP操作を自由に行うことによって、対応の実時間結果を生み出すことである。
図１は、ARM2がピッコロ４に制御信号を発行して、データワードをピッコロ４に対して送信させ、またデータワードをピッコロ４から転送させる様子を示す。命令キャッシュ６は、ピッコロ４にとって必要なピッコロプログラム命令ワードを記憶する。単一のDRAMメモリ８は、ARM2とピッコロ４の両方にとって必要なすべてのデータ及び命令ワードを記憶する。ARM2は、メモリ８へのアドレッシング（addressing）及びすべてのデータ転送の制御に責任がある。単一のメモリ８、及び１セットのデータバスとアドレスバスから成る構成は、複数のメモリと高い帯域幅のバスを必要とする典型的DSPアプローチに比較して、構

成が簡単であり、費用も易い。
ピッコロは、命令キャッシュ６からの第２の命令ストリーム（デジタル信号処理プログラム命令ワード）を実行し、これにより、ピッコロのデータパスが制御される。これらの命令は、デジタル信号処理方式操作、例えば、Multiply－Accumulate（演算－累算）、及び制御フロー命令、例えば、ゼロ・オーバーヘッド・ループ命令を含む。これらの命令は、ピッコロのレジスタ10（図２を参照）に保持されているデータを操作する。このデータは、前もって、ARM2によってメモリ８から転送されたものである。複数の命令が命令キャッシュ６からストリームとして出され、命令キャッシュ６が、データバスを、完全な支配下に置く。小型ピッコロ命令キャッシュ６は、１行当たり16ワードの４行で、直接マップされたキャッシュ（64個の命令）となる。導入の方法によっては、命令キャッシュをもっと大きくしてもよい。
このように、２つのタスクが独立的に走る。ARMがデータをロードして、ピッコロがそれを処理する。これにより、16ビット・データ上で単一サイクル・データ処理が維持される。ピッコロの持つデータ入力メカニズム（図２に示される）により、ARMは、シーケンシャル・データを先に取り込み、そのデータがピッコロに必要になるより先にロードする。ピッコロは、ロードされたデータにどのような順序でもアクセスすることができ、古いデータが最後に使用されると、自動的にそのレジスタを再び満たす（すべての命令はソースオペランド１つにつき、ソースレジスタを再充填すべきであることを示す１ビットを持つ）。この入力メカニズムは、リオーダ（reorder）バッファと呼ばれ、入力バッファ12を備える。ピッコロにロードされる（以下に示すLDCまたはMCRを介して）すべての値には、その値の目的地がどのレジスタであるかを示すタグRnが付いている。タグRnは、入力バッファ内のデータワードの側に記憶される。あるレジスタがレジスタ選択回路14を介してアクセスされ、命令がデータレジスタの再充填を指定すると、そのレジスタは、信号Ｅによって「空き」の印がつく。すると、レジスタは、自動的に、再充填制御回路16によって、その入力バッファ12内でそのレジスタに向けられた最も早くロードされた最古の値を充填される。リオーダ・バッファは８つのタグ付き値を保持する。入力バッファ12の形式は、FIFOと似ているが、キーの中央からデータワードを抽出することができ、その後で、遅くに記憶されたワードが渡され、その空き場所を埋める。従って、入力から最も遠いデータワードが最古であり、入力バッファ12が正しいタグRnを持つ２つのデータワードを保持する時は、その最古のデータワードを使用して、どちらのデータワードでレジスタを再充填すべきかを決定することができる。
ピッコロは、図３に示されたように、データを出力バッファ18（FIFO）に記憶させて出力する。データはFIFOにシーケンシャルに書き込まれ、ARMによって同じ順序でメモリ８に読み出される。出力バッファ18は、８つの32ビットの値を保持する。
ピッコロは、コプロセッサ・インターフェース（図１のCP制御信号）を介してARMと接続する。ARMコプロセッサ命令の実行に際して、ピッコロは、それを実行するか、あるいは、ピッコロがその命令を実行できるようになるまでARMを待たせるか、あるいは命令実行を拒否することができる。最後の場合、ARMは、未定義命令例外とする。
ピッコロが実行する最も普通のコプロセッサ命令はLDCとSTCであり、これらは、それぞれデータワードをデータバスを介してメモリ８へ、及びメモリ８からロードし（LDC）、記憶させ（STC）、ARMがすべてのアドレスを生成する。リオーダ・バッファにデータをロードし、出力バッファ18からのデータを記憶するのもこれらの命令である。ピッコロは、入力リオーダ・バッファにデータをロードするのに充分な場所がなければARMをLDCのままにし、また出力バッファに記憶すべき充分なデータがなければARMをSTCのままにする。ピッコロは、また、ARM/コプロセッサ・レジスタ転送を行って、ARMがピッコロの特定の（special）レジスタにアクセスできるようにする。
ピッコロは、それ自身の命令はメモリから取り込み、図３に示されたピッコロのデータパスを制御し、リオーダ・バッファからレジスタへ、またレジスタから出力バッファ18へデータを転送する。これらの命令を行うピッコロの演算ユニットは、乗算／加算回路20を有し、これが乗算、加算、減算、乗算・累算、論理操作、シフト、及び回転を行う。また、

データパスには累算／退出（decumulate）回路22と、縮尺（scale）／飽和（saturate）回路24とが備わっている。
ピッコロ命令は、最初にメモリから命令キャッシュ６にロードされ、そこへピッコロがアクセスし、主記憶にアクセスバックする必要がない。
メモリがアボート（abort）した場合、ピッコロはそれを修復することができない。従って、ピッコロを仮想メモリシステムで使用する場合、すべてのピッコロのデータは、ピッコロのタスクの始めから終わりまで、物理的メモリになければならない。このことは、ピッコロのタスクの実時間性、例えば実時間DSPを考えれば、大した問題ではない。メモリ・アボートが起きると、ピッコロは停止して状態レジスタS2にフラグをセットする。
図３は、ピッコロの全体のデータパス機能を示す。レジスタ・バンク10は、３つの読み出しポートと２つの書き込みポートを使用する。１つの書き込みポート（Ｌポート）は、リオーダ・レジスタからレジスタを再充填するのに使用される。出力バッファ18は、ALU結果バス26から直接的に更新され、出力バッファ18からの出力は、ARMプログラム制御の支配下にある。ARMコプロセッサ・インターフェースは、LDC（Load coprocessor）命令をリオーダ・バッファに行い、出力バッファ18からSTC（Store Coprocessor）命令を行い、また、レジスタバンク10上にMCRとMRC（Move ARM register to/from CP register）を行う。
残りのレジスタ・ポートは、ALUに使用される。読み出しポート（Ａ及びＢ）は、入力を乗算／加算回路20に駆動し、Ｃ読み出しポートは、累算（accumulate）／退出（decumulate）回路22入力の駆動に使用される。残りの書き込みポートＷは、結果をレジスタバンク10に戻すのに使用される。
乗算器20は、符号付き又は符号無し16x16の乗算を行い、必要により48ビット累算を伴うこともできる。スケーラー（scaler）ユニット24は、０から31までの即値算術又は論理シフト右を提供することができ、その後、必要により飽和を行うことができる。シフタ（shifter）及び論理ユニット20は、各周期でシフト又は論理操作を行うことができる。
ピッコロは、D0－D15又はA0－A3,X0－X3,Y0－Y3,Z0－Z3という名のついた16個の汎用レジスタを持つ。最初の４つのレジスタ（A0－A3）は、累算用で、48ビットの幅があり、余分な16ビットが、多数の連続的計算の間にオーバーフローが生じないためのガードを提供する。残りのレジスタは32ビットの幅である。
ピッコロのレジスタは各々２つの独立した16ビットの値を含むものとして扱うことができる。ビット０からビット15までが下半分、ビット16からビット31までが上半分を含む。命令は、ソースオペランドとして各レジスタのどちらかの半分の16ビットを指定することができ、あるいは、全体の32ビットレジスタを指定することもできる。
また、ピッコロは、飽和演算に対する備えもある。乗算、加算、減算命令の変量は、結果が目的レジスタのサイズより大きい場合、飽和結果を提供する。目的レジスタが48ビットのアキュムレータであれば、値は32ビットで飽和される（つまり、48ビットの値を飽和させる方法はない）。48ビットのレジスタにはオーバーフローの検出がない。これは手頃な制限である。というのは、オーバーフローを起こすには、少なくとも65536乗算累積命令が必要であるから。
各ピッコロのレジスタは、「空き」（Ｅフラグ、図２参照）であるか、１つの値を含む（レジスタの半分だけが空きになることはない）。初期状態では、すべてのレジスタが空きの印がついている。各周期で、ピッコロは再充填制御回路16によって、空きレジスタの１つを、入力リオーダ・バッファからの値で埋める。あるいは、レジスタにALUからの値が書き込まれている場合は、「空き」ではない。もし、レジスタにALUからの書き込みがあり、これと同時に、リオーダ・バッファからのレジスタに入れられる値が控えている場合は、結果は未定義である。空きレジスタに読み出しが行われれば、ピッコロの実行ユニットはとまってしまう。
入力オーダ・バッファ（ROB）は、コプロセッサ・インタフェースとピッコロのレジスタ・バンクとの間にある。データがROBにロードされる時は、ARMコプロセッサが転送する。ROBは、多数の32ビットの値を含み、それぞれ値の目的地となるピッコロ・レジスタを示

すタグを持っている。タグは、また、そのデータが32ビットレジスタの全体に転送されるのか、あるいは32ビット中の下の16ビットだけに転送されるべきかも示す。データがレジスタ全体に転送される場合は、そのエントリーの下の16ビットは目的レジスタの下半分に転送され、上の16ビットはレジスタの上半分に転送される（目的レジスタが48ビット・アキュムレータの場合は、符号が拡張される）。データの目的地がレジスタの下半分だけ（いわゆるハーフ・レジスタ）の場合、下の16ビットが先に転送される。
レジスタのタグは常に物理的目的レジスタを示し、レジスタのリマッピングが行われることはない（レジスタのリマッピングについては、以下を参照）。
各周期で、ピッコロは、次のように、データ・エントリをROBからレジスタ・バンクへ転送しようとする。
－　ROBの各エントリが検査され、タグが空きレジスタと比較され、エントリの一部又は全部からレジスタへ転送が可能かどうかが決定される。
－　転送可能なエントリの集合から、最古のエントリが選択され、そのデータがレジスタバンクへ転送される。
－　このエントリのタグが更新されてエントリを空きにする。エントリの一部だけが転送された場合は、転送された部分だけが空きの印になる。
例えば、目的レジスタが完全に空きであり、選択されたROBエントリが含むデータが１つの全体レジスタ用であれば、32ビット全体が転送され、そのエントリは空きの印になる。目的レジスタの下半分が空きであり、ROBの含むデータがレジスタの下半分用であれば、ROBエントリの下の16ビットが目的レジスタの下半分へ転送され、ROBの下半分が空きの印になる。
どのエントリでも、上の16ビットと下の16ビットは、それぞれ独立に転送することができる。レジスタバンクに転送できるデータを含むエントリが皆無の場合、その周期では、転送は行われない。下の表は、目的ROBエントリと目的レジスタ状態のあらゆる可能な組み合わせを示す。

	目的，Rn，状態		
目的 ROB エントリ状態	空き	下半分空き	上半分空き
レジスタ全体 両半分有効	Rn.h <- entry.h Rn.l <- entry.l エントリ「空き」	Rn.l <- entry.l エントリ下半分 「空き」	Rn.h <- entry.h エントリ上半分 「空き」
レジスタ全体 上半分有効	Rn.h <- entry.h エントリ「空き」		Rn.h <- entry.h エントリ「空き」
レジスタ全体 下半分有効	Rn.l <- entry.l エントリ「空き」	Rn.l <- entry.l エントリ「空き」	
レジスタ半分 両半分有効	Rn.l <- entry.l エントリ下半分 「空き」	Rn.l <- entry.l エントリ下半分 「空き」	
レジスタ半分 上半分有効	Rn.l <- entry.h エントリ「空き」	Rn.l <- entry.h エントリ「空き」	

以上をまとめると、１つのレジスタの２つの半分は、互いに独立に、ROBから充填することができる。ROB内のデータは、レジスタ全体用に印が付けられるか、あるいはレジスタの下半分用の２つの16ビットの値としての印が付く。

データをROBにロードするにはARMコプロセッサ命令が使用される。ROBにおいてデータが印を付けられる方法は、転送に使用されたARMコプロセッサ命令がどれであったかによる。ROBにデータを転送するのに使用できるARM命令には以下のものがある。

LDP {<cond>} <16/32>	<dest>,[Rn] {!} ,#<size>
LDP {<cond>} <16/32> W	<dest>, <wrap>,[Rn] {!} ,#<size>
LDP {<cond>} 16U	<bank>,[Rn] {!}
MPR {<cond>}	<dest>, Rn
MRP {<cond>}	<dest>, Rn

ROBの構成には、以下のARM命令が提供される。

LDPA <bank list>

最初の３つは、LDC命令としてアセンブルされ、MPRとMRPは、MCR命令として、LDPAはCDP命令としてアセンブルされる。
上記＜dest＞は、ピッコロのレジスタ（A0－Z3）を示し、RnはARMレジスタを示し、＜size＞は４の乗数（ゼロを除く）となる定数としてのバイト数であり、＜wrap＞は、定数（１、２、４、８）を示す。｛　｝によって囲まれた領域は、オプションである。転送がリオーダ・バッファへ当てはまるようにするために、＜size＞は最大で32である。多くの場合、＜size＞は、この制限より小さくしてデッドロックを避ける。＜16/32＞領域は、ロードされるデータが16ビット・データとして扱われ、endianess特定動作（以下を参照）を行うべきか、あるいは32ビットデータであるかを示す。
注1:以下の説明において、LDPまたはLDPWに言及する場合、これらの命令の16ビット用変種と32ビット用変種の両方を含むものとする。
注2:1つのワード（word）は、メモリからの32ビットの固まりであり、それは、16ビットのデータ項目２つから成るか、あるいは32ビットのデータ項目１つからなる。
LDP命令は、多数のデータ項目をフル・レジスタ用として転送する。この命令は、メモリ内のアドレスRnから＜size＞/4ワードをロードし、それらをROBに挿入する。転送することのできるワード数は以下のように制限される。
－　＜size＞の量は、４の非ゼロ倍数でなければならず、
－　＜size＞は、特定の導入についてROBのサイズ以下でなければならない（第１版では８ワード、その後の版では、それ以下にならない保証があること）。転送される最初のデータ項目は目的地が＜dest＞のタグを付け、第２のデータ項目は、＜dest＞＋１というようになる（Z3からA0まではラッピング（wrapping））。もし“!"が指定された場合は、その後、レジスタRnが＜size＞によって１つずつ増加される。
LDP16の変種が使用された場合は、エンダイアン（endian）特定動作が２つの16ビットのハーフワードに行われて、それらがメモリシステムから戻される時には32ビットデータ項目とする。より詳しくは、以下のBig Endian及びLittle Endianサポートを参照せよ。
LDPW命令は、多数のデータ項目をレジスタのセットに転送する。最初に転送されるデータ項目には＜dest＞のタグが付き、次は＜dest＞＋１のタグが付き、以下同様。＜wrap＞転送が起きると、次に転送される項目は、＜dest＞用となり、以下同様。＜wrap＞の量は、ハーフワードの量で指定される。
LDPWには、次の制限がある。
－　＜size＞の量は、４の非ゼロ倍でなければならず、
－　＜size＞は、特定の導入についてROBのサイズ以下でなければならず（第１版では８ワード、その後の版では、それ以下にならない保証がある）、
－　＜dest＞は、｛A0,X0,Y0,Z0｝のいずれか１つでよく、
－　＜wrap＞は、LDP32Wについては｛2,4,8｝のいずれかの個数のハーフワードであり、LDP16Wについては｛1,2,4,8｝のいずれかの個数のハーフワードであり、
－　＜size＞の量は、２＊＜wrap＞より大きくなければならない。さもないと、ラッピングは起きず、代わりにLDP命令が使用される。

たとえば、次の命令

```
LDP32W      X0, 2, [R0] !, #8
```

は、２つのワードをROBにロードし、その目的地をフル・レジスタX0とする。R0は、８増加する。次の命令

```
LDP32w      X0, 4, [R0], #16
```

は、４つのワードをROBにロードし、それらの目的地をX0,X1,X0,X1（この順序で）とする。R0は影響されない。

LSP16Wに対しては、＜wrap＞は、１、２、４、又は８として指定できる。１のラップが指定されると、すべてのデータのタグの目的地が、目的レジスタの下半分＜dest＞.1.となる。これは、ハーフ・レジスタの場合である。

例えば、次の命令

```
LDP16W      X0, 1, [R0] !, #8
```

は、２つのワードをROBにロードし、それらを16ビットデータとして目的地をX0.1とする。R0は８増加される。次の命令

```
LDP16W      X0, 4, [R0], #16
```

は、LDP32Wの例と同様に挙動するが、ただし、エンダイアン特定動作は、メモリから戻されるデータ上に行われる。

LDP命令のすべての使用されないエンコーディングは、将来の拡張用に取っておくことができる。

LDP16U命令は、非ワードを揃え（non－word aligned）16ビットデータの効率良い転送をサポートする。LDP16Uサポートはレジスタ D4－D15（X,Y,Zバンク）になされる。LDP16Uサポートは、レジスタ32ビットワードのデータ１つ（２つの16ビットデータ項目を含む）をメモリからピッコロへ転送することになる。ピッコロは、このデータの下の16ビットを捨て、上の16ビットを保持レジスタに記憶する。X,Y,Zバンク用の保持レジスタがある。バンクの保持レジスタが通報されると（primed）と、データの目的地がそのバンク内のレジスタであれば、LDP {w} 命令の挙動が変化する。ROBにロードされたデータは、LDP命令によって転送されつつあるデータの下の16ビットと保持レジスタとの連結によって形成される。転送されつつある上の16ビットは、保持レジスタに入れられる。

```
entry <- data.1 | holding__register

holding__register <- data.h
```

このモードの動作は、LDPA命令によって打ち切られるまで続く。保持レジスタは、目的レジスタのタグもサイズも記録しない。これらの性質は、次のdata.1.の値を提供する命令から得られる。

メモリシステムから戻されたデータには、常にエンダイアン特定挙動が起きる可能性がある。LDP16Uと同等の非16ビットはない。というのは、32ビットデータ項目はすべてメモリにおいてワード揃えされるからである。

LDPA命令は、LDP16U命令によって開始された操作の非整列（unaligned）モードを取り止めるのに使用される。非整列モードは、バンクX,Y,Z上で独立に切ることができる。例えば、次の命令

```
LDPA        {X, Y}
```

は、バンクX,Y上で非整列モードを打ち切る。これらのバンクの保持レジスタ内のデータは、捨てられる。

非整列モードにないバンク上でLDPAを実行することは可能であり、そのバンクは整列モードのままである。

MPR命令は、ARMレジスタRnの内容をROBに入れ、ピッコロレジスタ＜dest＞に向けられる。目的レジスタ＜dest＞は、A0－Z3の範囲のフルレジスタならどれでもよい。例えば、次の命令

```
MPR         X0, R3
```

は、R3の内容をROBに移し、そのデータをフルレジスタX0用とする。

データがARMからピッコロに転送される時にエンダイアネス（endianess）特定挙動が生じることがない。というのは、ARMは、内部的に、あまりエンダイアンではないからである。

MPRW命令は、ARMレジスタRnの内容をROBに入れ、それを、16ビットピッコロレジスタ＜dest＞.l.向けの２つの16ビットデータ項目とする。＜dest＞についての制限は、LDPWの命令の場合と同じである（つまり、Z0,X0,Y0,Z0）。例えば、次の命令

MPRW　　X0，R3

は、R3の内容をROBに移し、X0.l.向けの２つの16ビット量のデータとする。尚、１でラップするLDP16Wの場合、32ビットレジスタの下半分だけが目的地となり得る。

MPRについては、データに対してエンダイアネス特定操作は何も行われない。

LDPは、次のようにエンコードされる。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

COND	110	P	U	N	W	l	Rn	DEST	PICCOLO1	SIZE/4

ここで、PICCOLO1は、ピッコロの最初のコプロセッサの番号（現在８）である。ＮビットがLDP32（１）とLDP16（０）との間の選択を行う。

LDPWは、次のようにエンコードされる。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

COND	110	P	U	N	W	l	Rn	DES	WRA	PICCOLO2	SIZE/4

ここで、DESTは、目的レジスタA0,X0,Y0,Z0に対する０－３であり、WRAPは、１、２、４、８の値のラップに対して０－３である。PICCOLO2は、ピッコロの第２のコプロセッサ番号（現在９）である。Ｎビットが、LDP32（１）とLDP16（０）との間の選択を行う。

LDP16Uは、次のようにエンコードされる。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

COND	110	P	U	0	W	l	Rn	DES	01	PICCOLO2	00000001

ここで、DESTは、目的バンクX,Y,Zに対する１－３である。

LDPAは、次のようにエンコードされる。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

COND	1110	0000	0000	0000	PICCOLO1	000	0	BANK

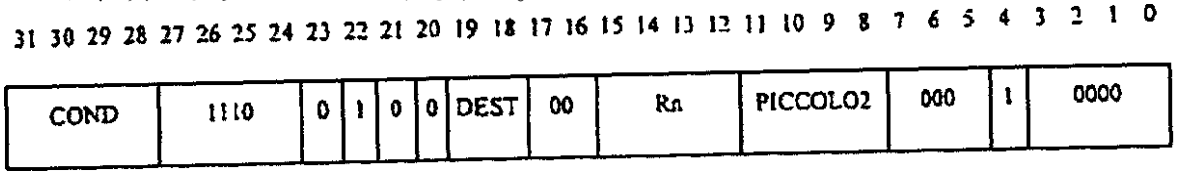
ここで、BANK〔3:0〕は、バンクごとの非整列モードを打ち切るのに使用される。BANK〔１〕がセットされると、バンクＸ上の非整列モードが打ち切られる。BANK〔２〕及びBANK〔３〕がセットされれば、それぞれバンクY,Z上の非整列モードが打ち切られる。尚、これはCDP操作である。

MPRは、次のようにエンコードされる。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

COND	1110	0	1	0	0	DEST	Rn	PICCOLO1	000	1	0000

MPRWは、次のようにエンコードされる。

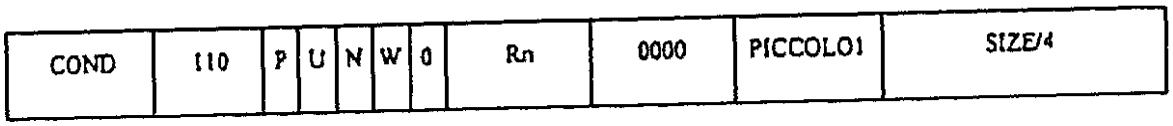


ここで、DESTは、目的レジスタX0,Y0,Z0に対する１－３である。
出力FIFOは、32ビットの値を８つまで保持することができる。これらは、次の（ARM）オペコード（opcodes）の１つを使用して、ピッコロから転送される。

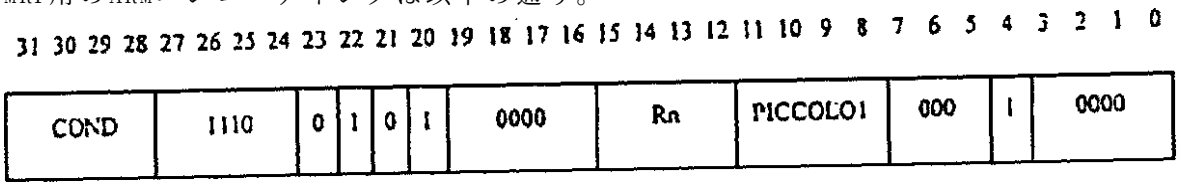
STP {<cond>} <16/32> [Rn] {!} ,#<size>

MRP Rn

最初のは、＜size＞/4ワードを出力FIFOから、ARMレジスタRnによって与えられるアドレスへ退避する。“!"があれば、Rnを指示する。デッドロックを避けるために、＜size＞は、出力FIFOのサイズ（この導入例では８エントリ）以下でなければならない。STP16の変種が使用された場合は、メモリシステムから戻されるデータにエンダイアン特定挙動が生じる可能性がある。
MRP命令は、出力FIFOから１つのワードを除去し、それをARMレジスタRnに入れる。MRPと同様に、このデータには、エンダイアン特定操作が適用されることはない。
STP用のARMエンコーディングは以下の通り。



ここで、Ｎは、STP32（１）とSTP16（０）との間の選択を行う。P,U,Wビットの定義については、ARMデータシートを参照せよ。
MRP用のARMエンコーディングは以下の通り。



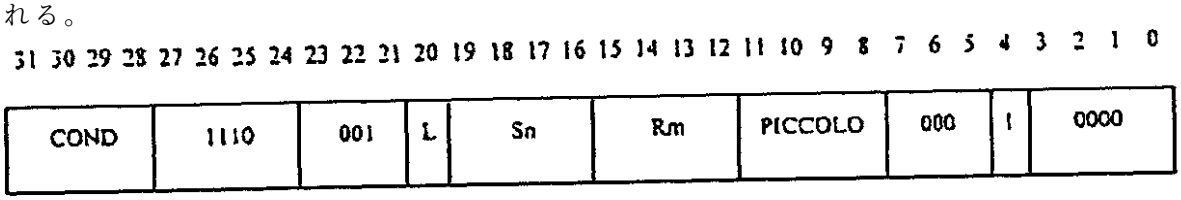
ピッコロ命令セットは、内部的にはエンダイアン操作がほとんどないと仮定している。例えば、32ビットレジスタに、複数の16ビット・ハーフとしてアクセスする場合、下半分がビット15から０を占めるとする。ピッコロは、大きなエンダイアン・メモリ又は周辺機器（peripherals）のあるシステムで動作することになるので、16ビットでパックされたデータを正しくロードできるようにしなければならない。
ピッコロ（つまり、DSPが採用されたコプロセッサ）は、ARM（例えば、英国、ケンブリッジのアドヴァンストRISCマシンズ・リミテッドによって製造されたARM7マイクロプロセッサ）のように、プログラマがプログラム可能周辺機器で制御できるであろう‘BIGEND'構成ピンを持っている。ピッコロは、このピンを使用して入力リオーダ・バッファ及び出力FIFOを構成する。
ARMがパック16ビットデータをリオーダ（reorder）・バッファにロードする時は、そのことを、LDP命令の16ビット形式を使用して示さなければならない。この情報は、‘BIGEND'構成入力の状態と組み合わされて、データを保持ラッチへ入れ且つリオーダ・バッファを適当な順序にする。特にbig endianモードの時は、保持レジスタはロードされたワードの下16ビットを記憶し、次のロードの上16ビットと対（ペア）にされる。保持レジスタの内容は、常に、リオーダ・バッファへ転送されたワードの下16ビットで終わる。
出力FIFOは、パックされた16ビット又は32ビットデータを含むことができる。プログラマ

は、STP命令の正確な形式を使用して、16ビットデータがデータバスの正しい半分に提供されていることをピッコロが確認できるようにしなければならない。big endianとして構成されている場合、STPの16ビット形式が使用されると、上16ビットハーフ及び下16ビットハーフが交換される。
ピッコロは、ARMからしかアクセスできないプライベート・レジスタを４つ持っている。これらは、S0－S2と呼ばれる。これらにアクセスできるのは、MRC命令とMCR命令だけである。オペコードは以下の通り。

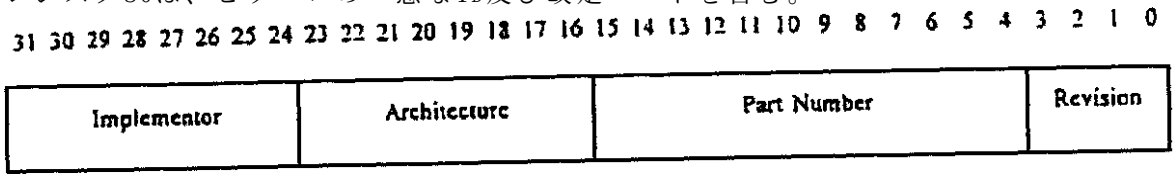
MPSR　　Sn，Rm

MRPS　　Rm，Sn

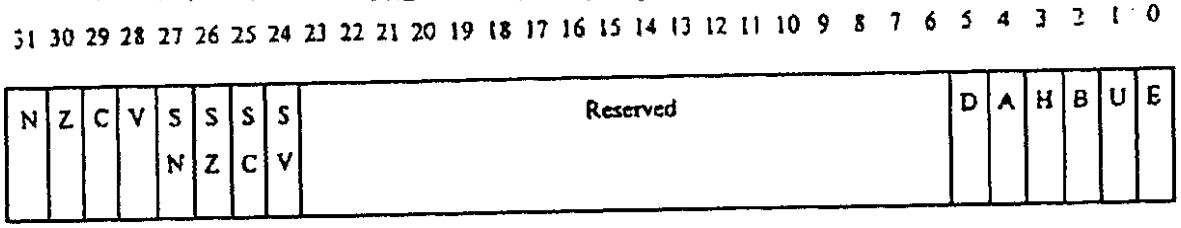
これらのオペコードは、ARMレジスタRmとプライベート・レジスタSnとの間で32ビット値を転送する。それらは、ARMにおいて、コプロセッサ・レジスタ転送としてエンコードされる。



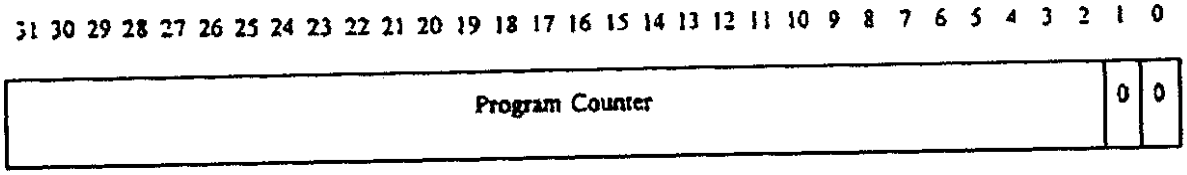
ここで、Ｌは、MPSRなら０、MRPSなら１である。
レジスタS0は、ピッコロの一意なID及び改定コードを含む。



〔3:0〕ビットは、プロセッサの改定番号を含む。
〔15:4〕ビットは、２進符号化された10進フォーマットの３桁部分の番号（ピッコロなら、0x500）を含む。
〔23:16〕ビットは、アーキテクチャ版数を含む。0x00＝第１版
〔31:24〕ビットは、導入者の商標のASCIIコードを含む。0x41＝Ａ＝ARM　Ltd.
レジスタS1は、ピッコロの状態レジスタである。



一次状件コードフラグ（N,Z,C,V）
二次状件コードフラグ（SN,SZ,SC,SV）
Ｅビット：ピッコロは、ARMによってディスエーブルされ、中止した。
Ｕビット：ピッコロは、UNDEFINED（未定義）命令に出会って、中止した。
Ｂビット：ピッコロは、BREAKPOINT（区切点）に出会って、中止した。
Ｈビット：ピッコロは、HALT（中止）命令に出会って、中止した。
Ａビット：ピッコロは、メモリ・アボート（ロード、ストア、又はピッコロ命令）によって、中止した。
Ｄビット：ピッコロは、デッドロック条件を検出し、中止した（以下を参照）。
レジスタS2はピッコロプログラム・カウンタである。



プログラム・カウンタに書き込みすると、ピッコロはそのアドレスで（中止状態であれば中止状態のまま）プログラムの実行を始める。プログラム・カウンタはリセットされた時、未定義である。というのは、プログラム・カウンタへの書き込みによって、ピッコロは常にスタートされるからである。

実行中、ピッコロは命令の実行及びコプロセッサ・インターフェースの状態を次のようにモニタする。

－　ピッコロは、レジスタ再充填されるのを、あるいは出力FIFOが使えるエントリを持つのを、待つ態勢に入った。

－　ROB内のスペースが不十分であるか、出力FIFO内の事項（items）が不十分であるかの理由で、コプロセッサ・インターフェースがビジー待ち状態（busy－waiting）にある。

これらの両方の条件が検出されると、ピッコロは、その状態レジスタにＤビットをセットし、中止し、ARMコプロセッサの命令を拒絶し、ARMは未定義命令トラップにはまる。

このデッドロック状態の検出により、少なくともプログラマにこのような条件が生じたことを知らせ、また失敗の正確な点（位置）を知らせることができるシステムが構成される。プログラマは、ARMとピッコロのプログラム・カウンタとレジスタを読めばよい。尚、強調しておくが、デッドロックが生じるのは、間違ったプログラムあるいはピッコロの状態を変造するシステム部分がある場合だけである。デッドロックは、データが少なすぎることや「オーバーロード」によって生じることはない。

ARMからピッコロを制御するのに使用できるいくつかの操作があり、それらはCDP命令によって提供される。これらのCDP命令は、ARMが優先状態にある時に受け付けられる。そうでないと、ピッコロはCDP命令を拒絶し、ARMは未定義命令トラップにはまる。以下の操作が使用可能である。

－　Reset（リセット）

－　Enter State Access Mode（状態アクセスモードに入る）

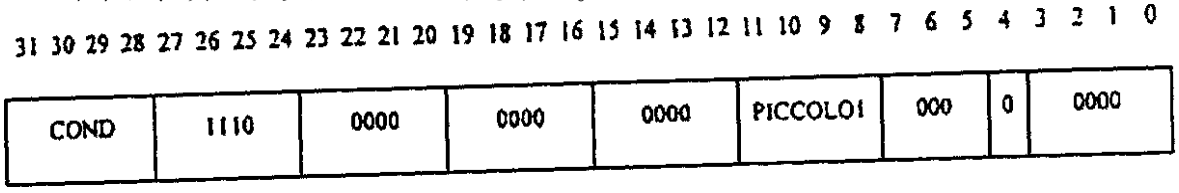
－　Enable（イネーブル）

－　Disable（ディスエーブル）

ピッコロは、PRESET命令によってソフトウェア内でプリセットされる。

PRESET;Clear Piccolo's state（ピッコロの状態をクリアする）

この命令は、次のようにエンコードされる。



この命令が実行されると、次のことが生じる。

－　すべてのレジスタが空き（再充填の態勢）の印になる。

－　入力ROBがクリアされる。

－　出力FIFOがクリアされる。

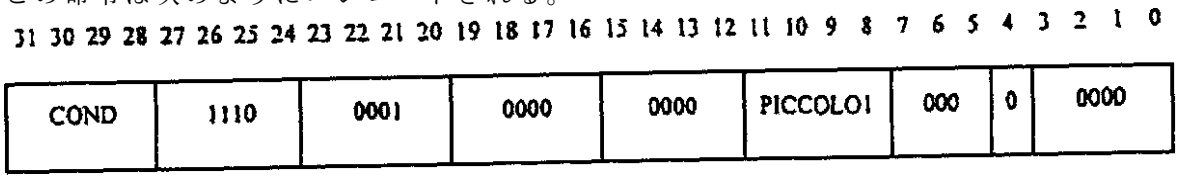
－　ループ・カウンタがリセットされる。

－　ピッコロは中止状態に入る（そしてS2のＨビットがセットされる）。

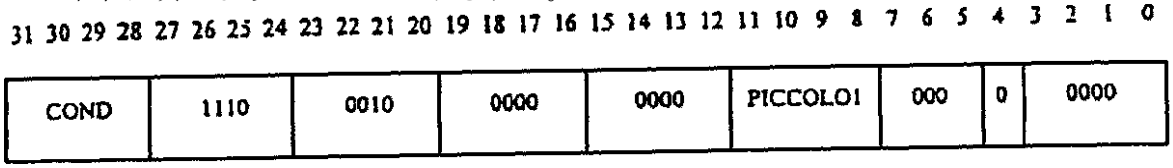
PRESET命令の実行には、いくつかのサイクル（この実施の形態では、２から３）が必要である。実行されている間に、以下のピッコロ上で実行されるべきARMコプロセッサ命令がビジー待ちになる。

状態アクセスモードにおいて、ピッコロの状態は、STC及びLDC命令（以下のARMからのピッコロ状態アクセスについての説明を参照）を使って退避され復元される。状態アクセス

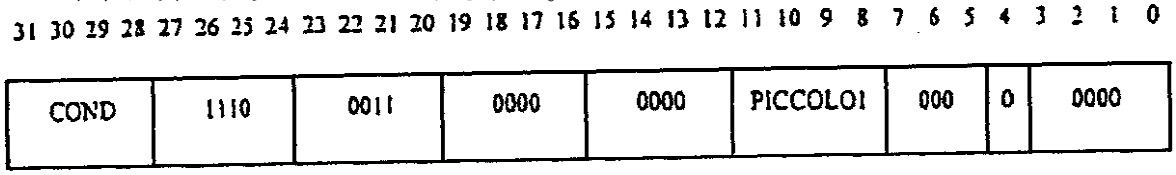
モードに入るには、PSTATE命令がまず実行されなければならない。
PSTATE;Enter State Access Mode（状態アクセスモードに入る）
この命令は次のようにエンコードされる。



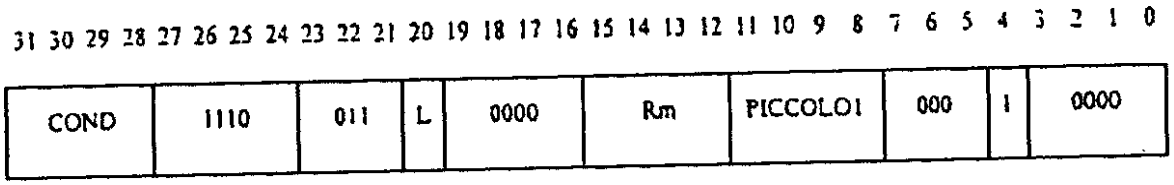
実行されると、PSTATE命令は、
－　ピッコロを中止し（すでに中止されているのでなければ）、Ｅビットをピッコロの状態レジスタにセットする。
－　ピッコロを状態アクセスモードに構成する。
PSTATE命令の実行が終わるまでにはいくつかのサイクルがある。というのは、ピッコロの命令パイプラインは中止する前に汲み出されなければならないからである。実行中、ピッコロ上で実行される次のARMコプロセッサ命令がビジー待ちになる。
PENABLE及びPDISABLE命令は、高速コンテキスト切替えに使用される。ピッコロがディスエーブルされると、専用レジスタ０と１だけが（IDレシスタ、状態レジスタ）アクセス可能となり、それも優先モードからだけである。これ以外の状態へアクセスすると、またユーザモードからアクセスすると、ARM未定義命令例外が生じる。ピッコロをディスエーブルすると、実行が中止される。ピッコロは、実行を中止すると、状態レジスタにＥビットをセットして応答する。
ピッコロをイネーブルするには、PENABLE命令を実行する。
PENABLE;Enable Piccolo
この命令は次のようにエンコードされる。



ピッコロをディスエーブルするには、PDISABLE命令を実行する。
PDISABLE;Disable Piccolo
この命令は次のようにエンコードされる。



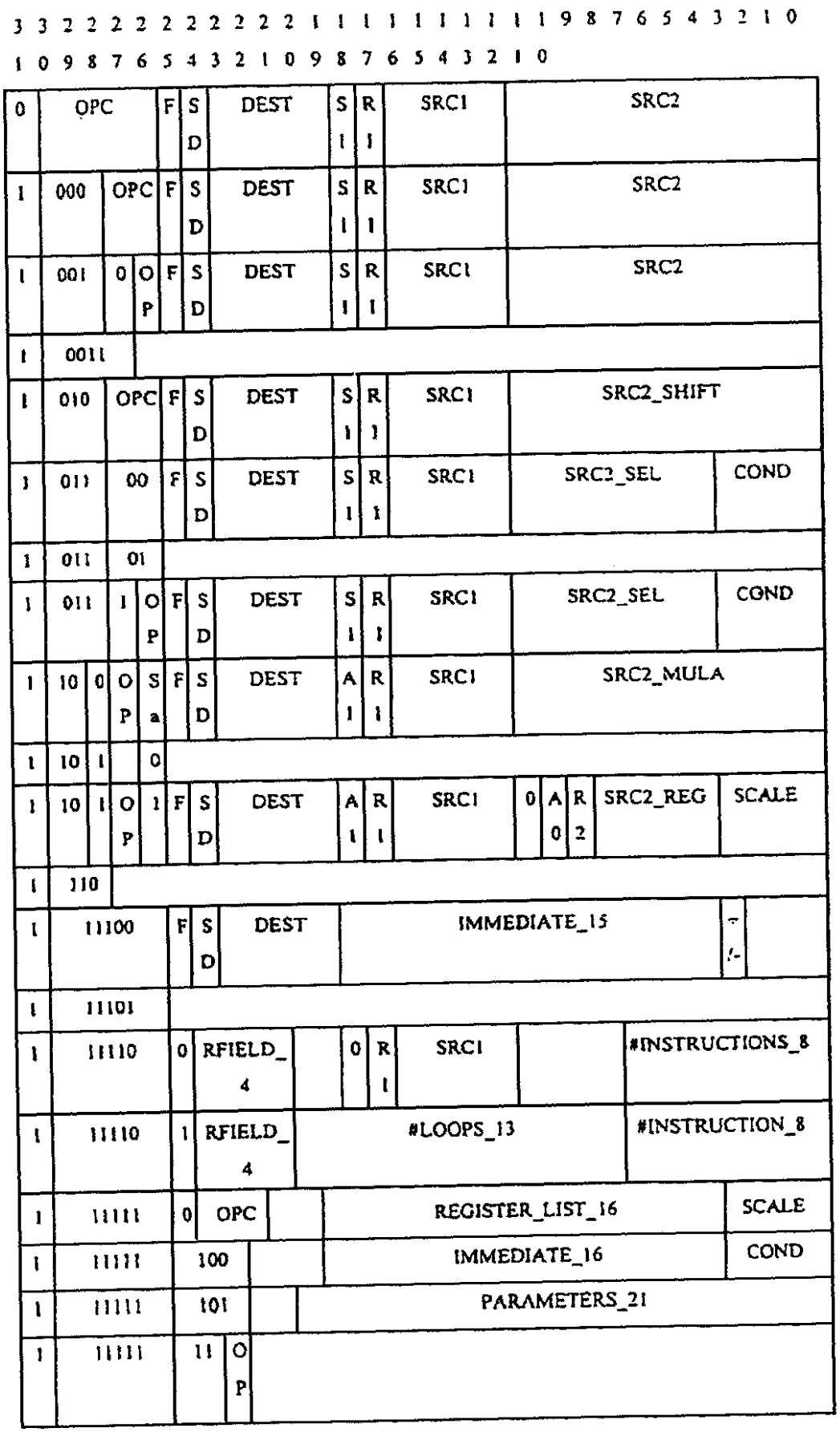
この命令が実行されると、次のことが生じる。
－　ピッコロの命令パイプラインが空になる（drain）。
－　ピッコロは中止して、状態レジスタにＨビットをセットする。
ピッコロ命令キャッシュは、ピッコロのデータパスを制御するピッコロの命令を保持する。もし存在すれば少なくとも64個の命令を保持し、それを16ワード境界から開始することが保証される。次のARMオペコードがMCRにアセンブルされる。その動作は、強制的にキャッシュに、（16ワード境界上にあるはずの）指定されたアドレスから始まる（16個の）命令のラインを取り込ませる（fetch）。この取り込みは、キャッシュがすでにこのアドレスに関係するデータを保持していても行われる。
PMIR　Rm
ピッコロは、PMIRが行われるより前に中止されなければならない。
このオペコードのMCRエンコーディングは以下の通り。



－　このセクションは、ピッコロのデータパスを制御するピッコロ命令セット（集合）に言及する。各命令は32ビットの長さである。これらの命令は、ピッコロ命令キャッシュから読み出される。

命令セットのデコードは、非常に直線的である。最初６ビット（26から31）が主要オペコードを与え、22から25までが、いくつかの特定の命令のためのマイナーなオペコードを提供する。灰色の影となっているコードは、現在未使用のものであり、拡張用として使える（それらは現時点で指示された値を含んでいなければならない）。

11の主要命令クラスがある。これは、いくつかのサブクラスのデコードを簡単にするため、命令にファイルされた主要オペコードに完全に対応するものではない。



上の表にある命令には、以下の名前がついている。
Standard Data Operation（標準データ操作）
Logical Operation（論理操作）
Conditional Add/Subtract（条件付加算／減算）

Undefined（未定義）
Shifts（シフト）
Select（選択）
Undefined（未定義）
Parallel Select（並列選択）
Multiply Accumulate（乗算累算）
Undefined（未定義）
Multiply Double（乗算ダブル）
Undefined（未定義）
Move Signed Immediate（符号付即値移動）
Undefined（未定義）
Repeat（反覆）
Repeat（反覆）
Register List Operation（レジスタ・シフト操作）
Branch（ブランチ）
Renaming Parameter Move（リネーム・パラメータ移動）
Halt/Break（中止／中断）
命令の各クラスのフォーマットは、次のセクションに詳しく述べてある。ソース及び目的オペランド領域は、ほとんどの命令において共通であり、レジスタ・リマッピングと同様、別のセクションに述べてある。
ほとんどの命令は２つのソースオペランドSource1,Source2を必要とする。
Source1（SRC1）オペランドは、次の７ビット・フォーマットを持つ。

18	17	16	15	14	13	12
Size	Refill	Register Number				Hi/Lo

この領域の要素は、次の意味を持つ。
－　Size－読み出すオペランドのサイズを示す（１＝32ビット、０＝16ビット）。
－　Refill－レジスタが読み出された後、空きの印になり、ROBから再充填できることを示す。
－　Register Number－32ビット、16ビットレジスタのどっちのレジスタを読み出すべきべきかエンコードする。
－　Hi/Lo－16ビット読み出しに対して、32ビットレジスタのどちらの半分を読み出すべきかを示す。32ビットオペランドに対してセットされた場合は、レジスタの２つの16ビット半分が入れ換えられなければならないことを示す。

Size	Hi/Lo	Portion of Register Accessed
0	0	Low 16 bits
0	1	High 16 bits
1	0	Full 32 bits
1	1	Full 32 bits, halves swapped

レジスタのサイズは、レジスタ番号に接尾辞を付けることによってアセンブラによって特

定される。下位16ビットなら、.l、上位16ビットなら、.h,32ビットの上下の16ビットを入れ換えるなら、.x。
一般のソース２（SCR2）は、次の３つの12ビット・フォーマットの１つを持つ。

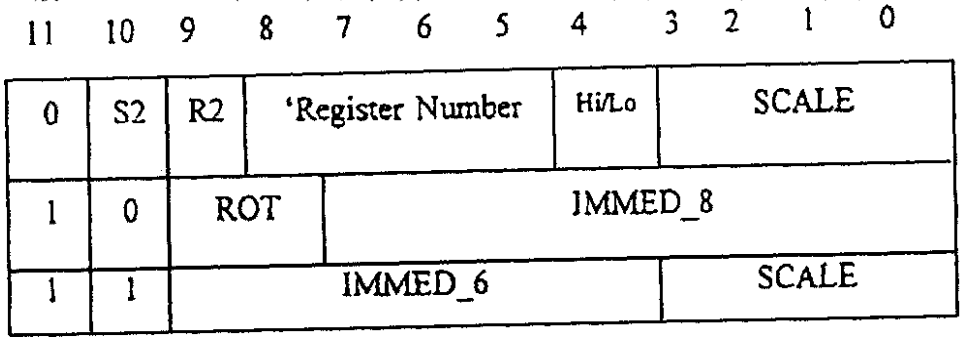


図４は、選択されたレジスタの適切な半分をピッコロのデータパスにスイッチするためのHi/Loビット及びSizeビットに応答するマルチプレクサ構成を示す。Sizeビットが16ビットであれば、符号拡張回路がデータパスの高次ビットに適切な０または１を入れる。
最初のエンコーディングは、ソースをレジスタとして指定し、その領域は、SCR1指定子（specifier）と同じエンコーディングを持つ。SCALE領域は、ALUの結果に適用されるべきスケールを指定する。

SCALE				Action
3	2	1	0	
0	0	0	0	ASR #0
0	0	0	1	ASR #1
0	0	1	0	ASR #2
0	0	1	1	ASR #3
0	1	0	0	ASR #4
0	1	0	1	RESERVED
0	1	1	0	ASR #6
0	1	1	1	ASL #1
1	0	0	0	ASR #8
1	0	0	1	ASR #16
1	0	1	0	ASR #10
1	0	1	1	RESERVED
1	1	0	0	ASR #12
1	1	0	1	ASR #13
1	1	1	0	ASR #14
1	1	1	1	ASR #15

８ビット即値（immediate）は、回転（rotate）エンコーディングによって、32ビット即値を生成し、それが、８ビット値及び２ビット回転（rotate）によって表現される。次の表は、８ビット値XYから生成される即値を示す。

ROT	IMMEDIATE
00	0x000000XY
01	0x0000XY00
10	0x00XY0000
11	0xXY000000

16ビット即値エンコーディングにより、６ビット符号無し即値（範囲０から63）を、ALUの出力に提供されるスケールと共に使用することができる。
一般のソース２エンコーディングは、ほとんどの命令変種に共通である。この規則には例外が少しあり、それがソース２エンコーディングの限定されたサブセットをサポートするか、あるいは、それを少し変形させる。
－ Select Instructions.（選択命令）
－ Shift Instructions.（シフト命令）
－ Parallel Operations.（並列操作）
－ Multiply Accumulate Instructions.（乗算累算命令）
－ Multiply Double Instructions.（乗算ダブル命令）
選択命令は、レジスタ又は16ビット符号無し即値であるオペランドをサポートするだけである。スケールは無効である。それは、これらのビットは命令の条件領域によって使用されるからである。

SRC2_SEL

11	10	9	8	7	6	5	4	3	2	1	0
0	S2	R2	Register number				Hi/Lo	COND			
1	1	IMMED_6						COND			

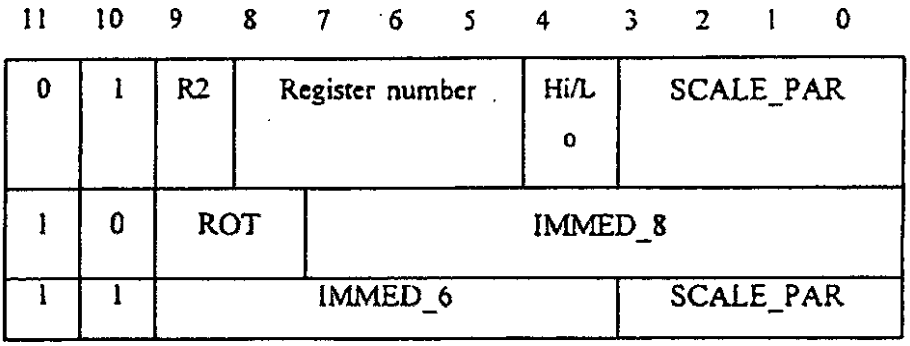
シフト命令は、16ビットレジスタ又は５ビット符号無し即値である１から31のオペランドをサポートするだけである。結果のスケールは無効である。

SRC2_SHIFT

11	10	9	8	7	6	5	4	3	2	1	0
0	0	R2	Register number				Hi/Lo	0	0	0	0
1	0	0	0	0	0	0	IMMED_5				

並列操作の場合、レジスタがオペランドのソースとして指定されていれば、32ビット読み出しが行われなければならない。即値エンコーディングは、並列操作については、少し違った意味を持つ。これにより、即値は、32ビットオペランドの16ビット半分の両方に複製できる。並列操作には少し制限のある範囲のスケールが使用できる。

SRC2_PARALLEL



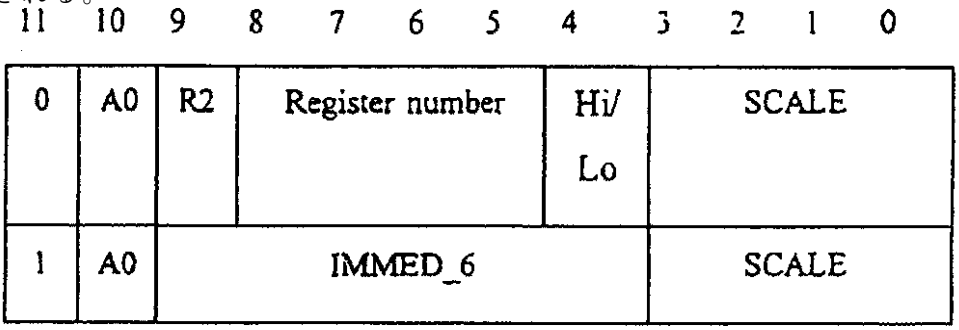
16ビット即値が使用された場合、常に、32ビット量の半分の両方に複製される。８ビット即値が使用された場合は、それが複製されるのは、それが32ビット量の上半分に回転されるべきであると回転（rotate）が示している時だけである。

ROT	IMMEDIATE
00	0x000000XY
01	0x0000XY00
10	0x00XY00XY
11	0xXY00XY00

並列選択操作にはスケールは無効である。スケール領域は、これらの命令では、０にセットされる。

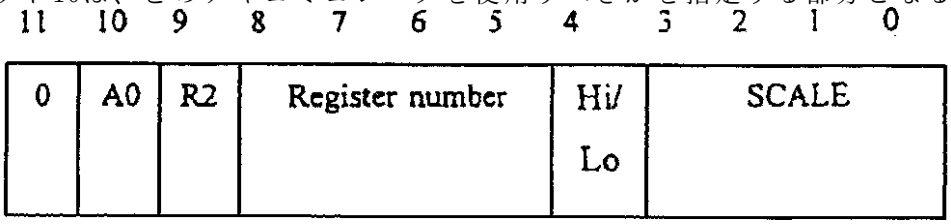
乗算累算命令では、８ビット回転即値を指定することはできない。領域のビット10は、どのアキュミュレータを使用すべきかを指定する部分となる。ソース２は、16ビットオペランドとして意味される。

SRC2_MULA



乗算ダブル命令は、定数を使用することができない。16ビットレジスタだけが指定できる。この領域のビット10は、どのアキュミュレータを使用すべきかを指定する部分となる。

SRC2_MULD



命令のうちいくつかは、常に32ビット操作（例えば、ADDADD）を含み、その場合、サイズ・ビットは、１にセットされ、Hi/Loビットは、場合によっては32ビットオペランドの２つの16ビット半分を交換するのに使用することができる。また、いくつかの命令は、常に

16ビット操作（例えば、MUL）を含み、サイズビットは０に設定されなければならない。Hi/Loビットは、レジスタのどちらの半分が使用されるかを選択する（見えないサイズビットは明らかなものと仮定する）。乗算・累算命令は、ソース・アキュミュレータと目的レジスタを独立に指定することができる。これらの命令においては、Sizeビットは、ソースアキュミュレータを指定するのに使用され、サイズビットは、命令タイプによって０と暗示される。

16ビット値が（Ａ又はＢバスを介して）使用される場合、それは、自動的に32ビット量に符号拡張される。48ビットレジスタが（Ａ又はＢバスを介して）読み出される場合、下の32ビットだけがバスに現れる。それは、どの場合でも、ソース１、ソース２は、32ビット値に変換されるからである。バスＣを使用する累算命令だけがアキュミュレータレジスタの48ビット全部にアクセスすることができる。

再充填ビットがセットされていれば、レジスタは使用後、空きの印になり、普通の再充填メカニズムによってROBから再充填される（ROBについてのセクションを参照）。ピッコロは、再充填が行われる以前にソースオペランドとしてレジスタが再び使用されないかぎり、止まらない。再充填されたデータが有効になる前のサイクルの最小数（最善の場合で、データはROBの先頭で待っている）は、１か２である。従って、再充填されたデータは、再充填要求の次の命令には使わない方が良い。もし、次の２つの命令上でオペランドの使用を避けることができるのであれば、その方が良い。というのは、これにより、より深いパイプライン導入上での性能損失を防ぐことになるから。

再充填ビットは、レジスタ番号に接尾辞“^"を付けることによってアセンブラで指定される。空きの印のついたレジスタのセクションは、レジスタのオペランドに依存する。各レジスタの２つの半分は、独立に、再充填の印をつけることもできる（例えば、X0.1^は、X0の下半分だけを再充填することになり、X0^は、X0全体を再充填することになる）。48ビットレジスタの上「半分」（ビット47:16）が再充填されると、16ビットのデータがビット31:16に書き込まれ、ビット47で符号拡張される。

同じレジスタを２回再充填しようと（例えば、ADD　X1,X0^,X0^）しても、再充填は１度しか行われない。アセンブラは、ADD　X1,X0,X0^という文法しか許可すべきではない。

レジスタ読み出しが、レジスタの再充填以前に試みられると、ピッコロは、レジスタが再充填されるまでまち状態で止まる。レジスタが再充填の印になると、レジスタは再充填の値が読まれる以前に更新され、その結果、UNPREDICATBLE（予想不可）となる（例えば、ADD　X0,X0^,X1は予想不可。なぜなら、X0については再充填の印であるから、X0とX1の合計で埋めることになる）。

４ビットスケール領域は14のスケールタイプをエンコードする。

－　ASR＃0,1,2,3,4,6,8,10
－　ASR＃12から16
－　LSL＃１

並列Max/Min命令は、スケールを提供しないので、ソース２の６ビット定数変種は使用されない（アセンブラにより０にセットされる）。

REPEAT命令内で、レジスタのリマッピングがサポートされ、REPEATが、ループを解かないままレジスタの移動「窓」にアクセスすることができる。これについて、以下、詳しく説明する。

目的オペランドは、次の７ビットフォーマットを持つ。

25	24	23	22	21	20	19
F	SD	HL	DEST			

この基本エンコーディングには10の変種がある。

アセンブラニューモニック		25	24	23	22	21	20	19
Dx	1	0	1	0	Dx			
Dx^	2	1	1	0	Dx			
Dx.l	3	0	0	0	Dx			
Dx.l^	4	1	0	0	Dx			
Dx.h	5	0	0	1	Dx _			
Dx.h^	6	1	0	1	Dx			
未定義		0	1	1	0000			
.l (レジスタ書き戻しなし, 16ビット)	7	1	1	1	0	0	00	
"" (レジスタ書き戻しなし, 32ビット)	8	1	1	1	0	1	00	
.l^ (16-ビット) 出力	9	1	1	1	1	0	00	
^ (32-ビット) 出力	10	1	1	1	1	1	00	

レジスタ番号（Dx）は16のレジスタのどれがアドレスされているかを示す。Hi/LoビットとSizeビットは、一緒になって、各32ビットレジスタへ16ビットレジスタのペアとしてアドレスする。Sizeビットは、どのようにしたら適切なフラグが、命令タイプに定義されているように、セットされるかを定義するもので、結果がレジスタバンク及び／又は出力FIFOに書き込まれるか否かには関係しない。これにより、比較及び同様の命令の構成ができる。命令の累算クラスのある加算は、結果をレジスタに書き戻さなければならない。

エンコード	レジスタ書込	FIFO書込	Vフラグ
1	レジスタ全体に書く	書込なし	32-ビット オーバーフロー
2	レジスタ全体に書く	32ビット書込	32-ビット オーバーフロー
3	ロー16ビットを Dx.1に書く	書込なし	16-ビット オーバーフロー
4	ロー16ビットを Dx.1に書く	ロー16ビットに 書込	16-ビット オーバーフロー
5	ロー16ビットを Dx.hに書く	書込なし	16-ビット オーバーフロー
6	ロー16ビットを Dx.hに書く	ロー16ビットに 書込	16-ビット オーバーフロー
7	書込なし	書込なし	16-ビット オーバーフロー
8	書込なし	書込なし	32-ビット オーバーフロー
9	書込なし	ロー16ビットに 書込	16-ビット オーバーフロー
10	書込なし	32ビット書込	32-ビット オーバーフロー

どの場合でも、レジスタへの書き戻し又は出力FIF0への挿入以前の操作の結果は、48ビット量である。２つの場合がある。

書き込みが16ビットならば、48ビット量は、以下の16ビット〔15:0〕を選択することによって、16ビットに減る。命令が飽和すれば、値は、－2^15から2^15－１の範囲に飽和される。次に16ビット値が指示されたレジスタに書き戻され、また、書き込みFIF0ビットがセットされれば、出力FIF0に書き戻される。出力FIF0に書き込まれた場合、それは、次の16ビット値が書き込まれるまで保持される。次の16ビットが書き込まれると、それらの値はペアとなって、単一32ビット値として出力FIF0に入れられる。

32ビットの書き込みならば、48ビットは、下の32ビット〔31:0〕を選んで32ビット量に減る。

32ビット書き込みでも、48ビット書き込みでも、命令が飽和すれば、48ビット値は－2^31－１から2^31の範囲の32ビット値に変換される。飽和すると、

－　アキュミュレーへの書き戻しが行われると、48ビット全部が書き込まれる。

－　32ビットレジスタへの書き戻しが行われると、ビット〔31:0〕が書き込まれる。

－　出力FIF0への書き戻しが行われるた場合も、やはりビット〔31:0〕が書き込まれる。

目的サイズは、レジスタ番号の後の.lまたは.hによって、アセンブラ内で指定される。レジスタへの書き戻しが全く行われない場合は、レジスタ番号は意味がなくなるので、目的

レジスタを省略して、レジスタへの書き込み無しとするか、あるいは、^を使って、出力FIFOだけへの書き込みを指示する。例えば、SUB,X0,Y0は、CMP X0,Y0と等価であり、ADD ^,X0,Y0は、X0＋Y0の値を出力FIFOに入れる。
出力FIFOに値を入れる空きがない場合は、ピッコロは、空きができるまで待機する。
16ビット値、例えば、ADD X0.h^,X1,X2が書き出されると、その値は、第２の16ビット値が書かれるまでラッチされる。次にこれら２つの値は結合されて、32ビット数として出力FIFOに入れられる。最初に書き込まれる16ビット値は、常に32ビットワードの下半分に現れる。出力FIFOに入れられたデータは、16又は32ビットデータとしての印がつき、endianessをbig endianシステム上で訂正することができる。
32ビット値が２つの16ビット書き込みの間に書き込まれると、その動作は未定義になる。
REPEAT命令内で、レジスタ・リマッピングがサポートされ、REPEATは、ループを解く（unroll）ことなしにレジスタの移動「窓」にアクセスすることができる。以下、これについて詳しく説明する。
本発明の好ましい実施の形態において、REPEAT命令は、レジスタ・オペランドがループ内で特定される方法を変更するメカニズムを提供する。このメカニズムの下で、アクセスするレジスタは命令内のレジスタ・オペランドとレジスタバンクのオフセットの機能によって決定される。オフセットは、プログラム可能な方法で変更でき、各命令ループの最後で変更されるのが好ましい。このメカニズムは、X,Y,Zバンク内にあるレジスタ上で独立に動作することができる。好ましい実施の形態では、この機能はＡバンク内のレジスタには使用できない。
論理レジスタ、物理レジスタという概念を使用することができる。命令オペランドは論理レジスタを参照し、これらは、特定のピッコロレジスタ10を同定する物理レジスタ・レファレンスにマップされる。すべての操作は、再充填も含み、物理レジスタ上で動作する。レジスタ・リマッピングが生じるのは、ピッコロ命令ストリームサイドだけであり、ピッコロにロードされるデータは常に物理レジスタを目的とし、リマッピングは行われない。
リマッピングのメカニズムについて、以下、図５を参照して説明する。図５は、ピッコロ・コプロセッサ４の多数の内部構成要素を示すブロック図である。メモリからARMコア２によって検索されるデータ項目は、リオーダ・バッファ12に入れられ、ピッコロレジスタ10は、先に図２を参照した方法で、リオーダ・バッファ12から再充填される。キャッシュ６に記憶されているピッコロの命令は、ピッコロ４内の命令デコーダ50に渡されることによって、ピコロ・プロセッサ・コア54に渡される前にデコードされる。ピッコロ・プロセッサ・コア54は、先に図３を参照して述べた乗算器／加算器回路20と、累算／退出回路22と、スケール／飽和（saturate）回路24とを備える。
命令デコーダ50がREPEAT命令によって同定された命令ループの一部を構成する命令を扱っていて、且つ、そのREPEAT命令が多数のレジスタのリマッピングを行うことが必要であると指示した場合は、レジスタ・リマッピング論理52が使用されて、必要なリマッピングが行われる。レジスタ・リマッピング論理52は、命令デコーダ50の一部であると考えて良い。ただし、当業者には明らかなように、レジスタ・リマッピング論理52は、命令デコーダ50に対して全く別のものとして提供されてもかまわない。
典型的な命令は、その命令にとって必要なデータ項目を含むレジスタを同定する１つまたは２つ以上のオペランドを備える。例えば、典型的な命令は、２つのソースオペランドと１つの目的ペランドを含むことができ、その命令が必要とするデータ項目を含む２つのレジスタと、その命令の結果を入れるべきレジスタを同定する。レジスタ・リマッピング論理52は、命令デコーダ50から命令のオペランドを受け取るが、それらは論理レジスタ・レファレンスを同定する。論理レジスタ・レファレンスに基づき、レジスタ・リマッピング論理は、物理レジスタのリマッピングをすべきかどうかを決定し、必要なら、物理レジスタ・レファレンスにリマッピングを適用する。また、リマッピングを適用すべきではないと決定された場合は、論理レジスタ・レファレンスが物理レジスタ・レファレンスとして提供される。リマッピングを行う好ましい方法については、後で、詳しく説明する。
レジスタ・リマッピング論理から各出力物理レジスタ・レファレンスは、ピッコロ・プロ

セッサ・コア54に渡されることによって、プロセッサ・コアが、物理レジスタ・レファレンスによって同定される特定のレジス10内のデータ項目に命令を適用できるようにする。
好ましい実施の形態によるリマッピングのメカニズムによれば、レジスタの各バンクは、２つのセクション、つまりその中でレジスタがリマップされるセクションと、レジスタがリマッピング無しで元のレジスタ・レファレンスを保持するセクションとの２つのセクションに割ることができる。好ましい実施の形態において、リマップされたセクションは、リマップされているレジスタ・バンクの下から開始される。
このリマッピングのメカニズムは多数のパラメータを使用し、これらのパラメータについては、図６を参照しながら、詳細に説明する。図６は、様々なパラメータがレジスタ・リマッピング論理52によっていかに使用されるかを示すブロック図である。尚、これらのパラメータは、リマップされているバンク内の点、例えば、バンクの下からの相対的値を与えられている。
レジスタ・リマッピング論理52は、２つの主要論理ブロック、つまりRemap（リマップ）ブロック56とBase Update（ベース更新）ブロック58とからなると考えることができる。
レジスタ・リマッピング論理52は、論理レジスタ・レファレンスに加えられるべきオフセット値を提供するベース・ポインタを使用する。このベース・ポインタの値は、ベース更新ブロック58によってリマップ・ブロックに提供される。
BASESTART信号を使用して、ベースポインタの初期値を定義することができる。例えば、典型的には、ゼロであるが、他の値を指定することもできる。このBASESTART信号は、ベース更新ブロック58内のマルチプレクサ60に渡される。命令ループの最初の繰り返しで、BASESTART信号は、マルチプレクサ60によって、記憶エレメント66に渡され、ループのその後の繰り返しでは、次のベース・ポインタ値がマルチプレクサ60によって記憶エレメント66に渡される。
記憶エレメント66の出力は、現在のベース・ポインタ値としてリマップ論理56に渡され、またベース更新論理58内の加算器62の入力の１つにも渡される。加算器62は、ベース・インクリメント値を提供するBASEINC信号を受け取る。加算器62は、記憶エレメント66によって供給される現在のベース・ポインタ値を、BASEINC値分だけインクリメントし、結果をモジュロ回路64へ渡すようになっている。
また、モジュロ回路は、BASEWRAP値を受け取り、この値を加算器62からの出力ベース・ポインタ信号と比較する。インクリメントされたベース・ポインタ値がBASEWRAP値以上であれば、その新しいベース・ポインタがラップラウンドされて、新しいオフセット値となる。モジュロ回路64の出力は、記憶エレメント66に記憶されるべき次のベース・ポインタ値となる。この出力はマルチプレクサ60に提供され、そこから、記憶エレメント66に提供される。
しかしながら、この次のベース・ポインタ値は、REPEAT命令を管理するループ・ハードウェアからBASEUPDATE信号を記憶エレメント66が受け取らないうちは、記憶エレメント66に記憶できない。BASEUPDATE信号は、ループ・ハードウェアによって周期的に生成され、例えば、命令ループが反復されるごとに、生成される。BASEUPDATE信号を記憶エレメント66が受け取ると、記憶エレメントは、以前のベース・ポインタ値にマルチプレクサ60から提供される次のベース・ポインタ値を上書きする。このように、リマップ論理58に供給されるベース・ポインタ値は、新しいベース・ポインタ値に変わる。
レジスタバンクのリマップされたセクション内でアクセスされるべき物理レジスタは、命令のオペランド内に含まれる論理レジスタ・レファレンスに、ベース更新論理58によって提供されるベース・ポインタ値を加えることによって決定される。この加算を行うのは加算器68であり、その出力は、モジュロ回路70に渡される。好ましい実施の形態において、モジュロ回路70は、レジスタ・ラップ値を受け取り、加算器68からの出力信号（論理レジスタ・レファレンスとベース・ポインタ値の和）がレジスタ・ラップ値を越えると、その結果へ、リマップされた領域の下でラップ（wrap）が行われる。モジュロ回路70の出力は、マルチプレクサ72に提供される。
REGCOUNT値がリマップ・ブロック56内の論理74に提供され、リマップされるべきバンク内

のレジスタの個数を同定する。論理74は、このREGCOUNT値を論理レジスタ・レファレンスと比較し、比較の結果により、制御信号をマルチプレクサ72に渡す。マルチプレクサ72は、その２つの入力で、論理レジスタ・レファレンスとモジュロ回路70からの出力（リマップされたレジスタ・レファレンス）を受け取る。本発明の好ましい実施の形態において、論理レジスタ・レファレンスがREGCOUNT値より小さければ、論理74は、マルチプレクサ72にリマップされたレジスタ・レファレンスを物理レジスタ・レファレンスとして出力させる。ただし、もし、論理レジスタ・レファレンスがREGCOUNT値以上であれば、論理74は、マルチプレクサ72に論理レジスタ・レファレンスを直接、物理レジスタ・レファレンスとして出力させる。

先に述べたように、好ましい実施の形態において、リマッピング・メカニズムを引き起こすのはREPEAT命令である。後で、より詳しく述べるように、REPEAT命令は、ハードウェアで４つのゼロサイクルループを提供する。これらのハードウェア・ループは、図５に命令デコーダ50の部分として図示されている。命令デコーダ50がキャッシュ６から命令を要求する度に、キャッシュはその命令を命令デコーダに戻し、それにより、命令デコーダは、戻された命令がREPEAT命令であるかどうかを判断する。もしそれであれば、ハードウェア・ループの１つが、そのREPEAT命令を扱うように構成される。

各繰り返し命令は、ループ内の命令の数と、ループを繰り返す回数（定数またはピッコロ・レジスタから読み出される）を指定する。２つのオペコードREPEATとNEXTがハードウェアループの定義用に提供され、NEXTオペコードは単に区切りとして使用されるだけで、命令としてアセンブルはされない。REPEATがループの頭に行き、NEXTがループの最後を区切ることによって、アセンブラはループ・本体内の命令の数を数えることができる。好ましい実施の形態において、REPEAT命令は、レジスタ・リマッピング論理52が使用すべきREGCOUNT,BASEINC,BASEWRAP,REGWRAPパラメータのようなリマッピング・パラメータを含むことができる。

レジスタ・リマッピング論理によって使用されるリマッピング・パラメータを記憶する多数のレジスタを提供することができる。これらのレジスタ内で、前もって定義されたリマッピング・パラメータの多数のセット（集合）を提供することができる一方、いくつかのレジスタはユーザ定義リマッピング・パラメータを記憶するために残される。REPEAT命令と共に指定されたリマッピング・パラメータが、前もって定義されたリマッピング・パラメータの１つと等しい場合、適当なREPEATエンコーディングが使用され、これにより、マルチプレクサ等が適当なリマッピング・パラメータをレジスタから直接にレジスタ・リマッピング論理へ提供する。一方、リマッピング・パラメータが前もって定義されたリマッピング・パラメータのどれとも等しくない場合は、アセンブラがRemapping Parameter Move（RMOV）命令を生成する。これにより、ユーザ定義レジスタ・リマッピング・パラメータの構成が可能となり、RMOV命令の後にREPEAT命令が続く。好ましくは、ユーザ定義リマッピング・パラメータは、RMOV命令によって、そのようなユーザ定義リマッピング・パラメータを記憶すべく残されていたレジスタに入れられ、マルチプレクサは、それらのレジスタの内用をレジスタ・リマッピング論理に渡すようプログラムされる。

好ましい実施の形態において、REGCOUNT,BASEIN,BASEWRAP,REGWRAPパラメータは、以下のチャートに示された値の１つを取る。

パラメータ	説　明
REGCOUNT	これは、0、2、4、又は8で、リマッピングを行う16ビットレジスタの数を同定する。REGCOUNT未満のレジスタはリマップされ、REGCOUNT以上のレジスタは直接アクセスされる。
BASEINC	これは、16ビットレジスタをいくつ使用して、各ループ繰り返しの最後でベース・ポインタがインクリメントされるかを定義する。好ましい実施の形態では、1、2、又は4の値を取れるが、実際には所望すれば、負の値を含む他の値も取れる。
BASEWRAP	これはベース計算のシーリングを決める。ベース・ラッピング・モジュラスは、2、4、8の値を取れる。
REGWRAP	これは、リマップ計算のシーリングを決める。レジスタ・ラッピング・モジュラスは2、4、8の値を取れる。REGWRAPは、REGCOUNTと等しく選択することができる。

図6に戻り、リマップ・ブロック56によって様々なパラメータが使用される例を次に示す（この例では、論理及び物理レジスタ値は、特定バンクに対する相対値である。）

```
if (Logical Register (論理レジスタ) <REGCOUNT)
    Physical Register(物理レジスタ) =(Logical Register(論理レジスタ) +
                                Base (ベース) ) MOD  REGCOUNT
else
    Physical Register(物理レジスタ) =Logical Register (論理レジスタ)
end if
```

ループの最後で、ループの次の繰り返しが始まる前に、次のベース・ポインタ更新がベース更新論理58によって行われる。

Base＝（Base＋BASEINC）MOD　BASEWRAP

リマッピング・ループの最後でレジスタ・リマッピングが打ち切られ、すべてのレジスタは物理レジスタとしてアクセスされる。好ましい実施の形態において、1つのリマッピングREPEATだけがどの時点においてもアクティブである。ループは、ネストされたままであるが、ある特定の時点で1つだけがリマッピング変数を更新してよい。ただし、所望するなら、リマッピング繰り返しはネストできるようにする。

本発明の好ましい実施の形態に基づくリマッピング・メカニズムを使用した結果としてのコード密度に関して達成される効果を示すために、以下、典型的なブロック・フィルタ・アルゴリズムについて説明する。まず、ブロック・フィルタ・アルゴリズムの原則について、図7を参照しながら説明する。図7に示されているように、アキュミュレータ・レジスタA0は、多数の乗算操作の結果を累算するように備えられている。この乗算操作というのは、係数c0とデータ項目d0との乗算、係数c1とデータ項目d1との乗算、係数c2とデータ項目d2との乗算などである。レジスタA1は、乗算操作の同様のセットの結果を累算していくが、今度は、係数がずれて、c0とd1、c1とd2、c2とd3と組み合わせの乗算になる。同様に、レジスタA2は、係数値を更にずらして、c0とd2、c1とd3、c2とd4といった組み合わせの乗算の結果を累算する。このシフト、乗算、累算のプロセスが、繰り返され、その結果がレジスタA3に入れられる。

本発明の好ましい実施の形態に基づくレジスタ・リマッピングを使用しないと、ブロック・フィルタ命令を実行するには、次のような命令ループが必要となる。

```
; start with 4 new data values
  （４つの新しいデータ値で始める）
ＺＥＲＯ｛Ａ０－Ａ３｝        ; Zero the accumulators
                              （アキュミュレータをゼロにする）
ＲＥＰＥＡＴ　Ｚ１            ; Ｚ１＝(number of coeffs/4)
                                        （係数の個数）
; do the next four coefficients, on the first time around:
（第１回目に、次の４つの係数で行う）
; a0 += d0* c0+ d1* c1+ d2* c2+ d3* c3
; a1 += d1* c0+ d2* c1+ d3* c2+ d4* c3
```

```
; a2 += d2* c0+ d3* c1+ d4* c2+ d5* c3
; a3 += d3* c0+ d4* c1+ d5* c2+ d6* c3
MULA  A0, X0.l^, Y0.l,   A0   ;a0 += d0* c0, and load d4
                                    (そしてｄ４をロードする)
MULA  A1, X0.h , Y0.l,   A1   ;a1 += d1* c0
MULA  A2, X1.l , Y0.l,   A2   ;a2 += d2* c0
MULA  A3, X1.h , Y0.l^,  A3   ;a3 += d3* c0, and load c4
                                    (そしてｃ４をロードする)
MULA  A0, X0.h^, Y0.h,   A0   ;a0 += d1* c1, and load d5
                                    (そしてｄ５をロードする)
MULA  A1, X1.l , Y0.h,   A1   ;a1 += d2* c1
MULA  A2, X1.h , Y0.h,   A2   ;a2 += d3* c1
MULA  A3, X0.l , Y0.h^,  A3   ;a3 += d4* c1, and load c5
                                    (そしてｃ５をロードする)
MULA  A0, X1.l^, Y1.l,   A0   ;a0 += d2* c2, and load d6
                                    (そしてｄ６をロードする)
MULA  A1, X1.h , Y1.l,   A1   ;a1 += d3* c2
MULA  A2, X0.l , Y1.l,   A2   ;a2 += d4* c2
MULA  A3, X0.h , Y1.l^,  A3   ;a3 += d5* c2, and load c6
                                    (そしてｃ６をロードする)
MULA  A0, X1.h^, Y1.h,   A0   ;a0 += d3* c3, and load d7
                                    (そしてｄ７をロードする)
MULA  A1, X0.l , Y1.h,   A1   ;a1 += d4* c3
MULA  A2, X0.h , Y1.h,   A2   ;a2 += d5* c3
MULA  A3, X1.l , Y1.h^,  A3   ;a3 += d6* c3, and load c7
                                    (そしてｃ７をロードする)
NEXT
```

この例において、データ値はレジスタのＸバンクに入れられ、係数値はレジスタのＹバンクに入れられる。第１ステップとして、４つのアキュミュレータ・レジスタA0,A1,A2,A3はゼロにセットされる。アキュミュレータ・レジスタがリセットされると、命令ループが開始され、このループはREPEAT命令及びNEXT命令によって区切られる。Z1の値は、この命令ループが繰り返される回数を示し、また後で述べる理由により、この回数は、実際には

、係数の個数（c0,c1,c2など）を４で割った数に等しい。
命令ループには16の乗算累算命令（MULA:multiply accumulate instructions）があり、１回目の繰り返しが終わると、その結果、レジスタA0,A1,A2,A3は、REPEAT命令と第１のMULA命令との間で上のコードで示される計算の結果を含む。乗算累算操作がどのように動作するかを示すために、最初の４つのMULA命令を考えることにする。最初の命令によって、Ｘバンク・レジスタ・ゼロの最初のすなわち下の16ビット内のデータ値と、Ｙバンク・レジスタ・ゼロ内の下の16ビットとが掛け合わされ、その結果がレジスタA0に加えられる。これと同時に、Ｘバンク・レジスタ・ゼロの下の16ビットが再充填の印になり、レジスタのこの部分に新しいデータ値が再充填できることを示す。このように印がつき、図７から明らかなように、データ項目d0が係数c0で乗算されると（これは最初のMULAによって表される）、d0は、ブロック・フィルタ命令の残り部分では不要になり、新しいデータ値で置き換えられる。
次に、第２のMULAによって、Ｘバンク・レジスタ・ゼロの第２のすなわち上の16ビットと、Ｙバンク・レジスタ・ゼロの下の16ビットとが掛け合わされ（これは、図７における,d1　ｘ　c0を表す）。同様に、第３、第４のMULA命令が、d2　ｘ　c0,及びd3　ｘ　c0の乗算を行う。図７から明らかなように、これらの４つの計算が行われると、係数C0は不要となり、レジスタY0.lは、再充填の印がつき、他の係数（c4）で上書きできるようになる。
次の４つのMULA命令は、それぞれ、d1xc1,d2xc1,d3xc1,d4xc1の計算を表す。d1xc1の計算が終了すると、d1は不要になるので、レジスタX0.hは再充填ビットの印がつく。同様に、４つの計算すべてが終了すると、係数c1は不要になるので、レジスタY0.hは再充填用の印がつく。同様に、次の４つのMULA命令は、d2xc2,d3xc2,d4xc2,d5xc2の計算に対応し、最後の４つの計算は、d3xc3,d4xc3,d5xc3,d6xc3の計算に対応する。
上記の実施の形態において、リジスタはリマップできず、各乗算操作は、オペランドで指定される特定レジスタによって明示的に再生されなければならない。16のMULA命令の実行が終了すると、係数c4からc7及びデータ項目d4からd10まで、命令ループを繰り返すことができる。また、ループは、繰り返し１回につき４つの係数値で操作するので、係数値の個数は、４の倍数でなければならず、Z1＝係数/4個の計算が行われる。
本発明の好ましい実施の形態におけるリマッピング・メカニズムを使用することによって、命令ループは飛躍的に減らすことができ、４つの乗算累算命令を含むだけになる。さもなければ16の乗算累算命令が必要になる。このリマッピング・メカニズムを使用すると、コードは以下のように書くことができる。

```
; start with 4 new data values
  (４つの新しいデータ値で始める)
ＺＥＲＯ {Ａ０－Ａ３}        ; Zero the accumulators
                              (アキュミュレータをゼロにする)
ＲＥＰＥＡＴ  Z1, X++ n4 w4 r4, Y++ n4 w4 r4;Z1=(number of coefficients)
                                              (係数の個数)
; Remapping is applied to the X and Y banks.
  (リマッピングがＸとＹバンクに適用される。)
; Four 16 bit registers in these banks are remapped.
  (これらのバンク内の４つの１６ビット・レジスタがリマップされる。)
; The base pointer for both banks is incremented by one on each
; iteration of the loop.
  (両方のバンクのベース・ポインタが、ループの繰り返しごとに１つインクリ
メントされる。)
;The base pointer wraps when it reaches the fourth register in the bank.
  (ベース・ポインタは、バンク内の第４のレジスタに到達するとラップ(wrap)
する。)
ＭＵＬＡ  A0, X0.l^, Y0.l,   A0   ;a0 += d0* c0, and load d4
                                    (そしてｄ４をロードする)
ＭＵＬＡ  A1, X0.h,   Y0.l,   A1   ;a1 += d1* c0
ＭＵＬＡ  A2, X1.l,   Y0.l,   A2   ;a2 += d2* c0
ＭＵＬＡ  A3, X1.h,   Y0.l^,  A3   ;a3 += d3* c0, and load c4
                                    (そしてｃ４をロードする)
ＮＥＸＴ                      ;go round loop and advance remapping
                              (ループを１周し、リマップを進める)
```

先に述べたのと同様に、第１のステップで、４つのアキュミュレータ・レジスタA0－A3をゼロにセットする。次に、REPEATオペコードとNEXTオペコードによって区切られる命令ループに入る。REPEAT命令は、以下のように多数のパラメータを持つ。
Ｘ＋＋：レジスタのＸバンクに、BASEINCが‘1'であることを示す。
n4:REGCOUNTが‘4'であり、従って、最初の４つのＸバンクレジスタX0.1からX1.hがリマップされることを示す。
w4:レジスタのＸバンクに、BASEWRAPが‘4'であることを示す。
Ｙ＋＋：レジスタのＹバンクに、BASEINCが‘1'であることを示す。
n4:REGCOUNTが‘4'であり、従って、最初の４つのＹバンクレジスタY0.1からY1.hがリマ

ップされることを示す。
w4:レジスタのＹバンクに、BASEWRAPが‘4'であることを示す。
r4:レジスタのＹバンクに、REGWRAPが‘4'であることを示す。
尚、Z1の値は、先行技術の例では、係数の個数/4に等しくなるが、ここでは、係数の個数と等しくなる。
命令ループの最初の繰り返しで、ベースポインタの値はゼロであり、リマッピングはない。ただし、次にループが実行される時は、ＸバンクもＹバンクもベース・ポインタの値は‘1'であるから、オペランドは次のようにマップされる。
X0.lはX0.hになる
X0.hはX1.lになる
X1.lはX1.hになる
X1.hはX0.lになる（BASEWRAPが‘4'だから）
Y0.lはY0.hになる
Y0.hはY1.lになる
Y1.lはY1.hになる
Y1.hはY0.lになる（BASEWRAPが‘4'だから）
従って、２回目の繰り返しでは、本発明のリマッピングを含まない先に述べた例における第５から第８番目のMULA命令によって示される計算を、４つのMULA命令が実際に行うことがわかる。同様に、３回目、４回目のループの繰り返しでは、先行技術コードの第９から第12番目、そして第13から第16番目のMULA命令によって実行された計算が行われる。
従って、上記コードは、先行技術のコードと全く同様のブロック・フィルタ・アルゴリズムを行うわけだが、ループ本体内のコード密度を４倍に改善している。つまり、先行技術では16の命令が必要であったのに比較して、４つの命令ですむ。
本発明の好ましい実施の形態に基づくレジスタ・リマッピング技術を使用することによって、以下のような利点が得られる。
1.コード密度を改善する。
2.場合によっては、レジスタを空きとして印をしてからピッコロのリオーダ・バッファによって再充填されるまでのレイテンシー（latency）を隠すこともできる。これは増えるコードサイズを捨ててアンローリングループによって実現される。
3.アクセスされるべきレジスタの数を変化させることができる。ループ繰り返し実行数を変化させることによって、アクセスされるレジスタの数を変化させることができる。
4.アルゴリズム開発を簡単にすることができる。適当なアルゴリズムについて、プログラマはアルゴリズムのｎ番目の段に対する１つのコードを生成して、レジスタ・リマッピングを使用して、その公式をデータのスライディング・セットに適用することができる。
上記レジスタ・リマッピング・メカニズムは、本発明の範囲から離れることなく、ある程度の変形が可能であることが明らかになるであろう。例えば、レジスタ10のバンクは、プログラマによって命令オペランドに指定される以上の物理レジスタを提供することができる。これらの余分のレジスタは直接的にはアクセスできないが、レジスタ・リマッピング・メカニズムでは、これらのレジスタを使用することができる。例えば、先に出した例を考えてみよう。レジスタのＸバンクに、プログラマの使える32ビットレジスタが４つあり、従って８つの16ビットレジスタが論理レジスタ・レファレンスによって指定するこができる。レジスタのＸバンクが、実際には、例えば６つの32ビットレジスタから成る場合、プログラマにとって直接アクセスできない16ビットレジスタが余分に４つあることになる。しかしながら、これらの４つのレジスタは、リマッピング・メカニズムによって使用可能となり、データ項目の記憶のための付加的レジスタを提供する。
以下のアセンブラ・シンタクス（文法）を使用することができる。
＞＞は、論理右シフト、又は、シフト・オペランドが負であれば、左シフトを意味する（下の＜1scale＞を参照）。
－＞＞は、算術右シフト、又は、シフト・オペランドが負であれば、左シフトを意味する（下の＜scale＞を参照）。

RORは、右回転を意味する。
SAT（ａ）は、ａの飽和値を意味する（目的レジスタのサイズによって、16ビット又は32ビットで飽和する）。特に、16ビットで飽和するために、＋0x7fffより大きいどんな値も＋0x7fffで置き換えられ、－0x8000より小さいどんな値も－0x8000で置き換えられる。32ビット飽和は、どうように、極限値＋0x7fffffffと－0x80000000がある。目的レジスタが48ビットである場合も、飽和は32ビットで行われる。
ソース・オペランド１は、次のフォーマットの１つを取ることができる。
＜src1＞は、

〔Rn｜Rn. 1｜Rn. h｜Rn. x〕〔^〕

の短縮形として使用される。別の言い方をするなら、ソース・スペシファイアの７ビットはすべて有効であり、レジスタは32ビット値として（希望すれば、交換される）、または符号拡張した16ビット値として読まれる。アキュムレータに取っては、下の32ビットだけが読まれる。“^"は、レジスタ再充填を指定する。
＜src_16＞は、

〔Rn. 1｜Rn. h〕〔^〕

の短縮形として使用される16ビット値だけが読まれる。
＜src_32＞は、

〔Rn｜Rn. x〕〔^〕

の短縮形として使用される。32ビット値だけが読まれ、上半分及び下半分は希望すれば交換できる。
ソース・オペランド２は、次のフォーマットの１つを取ることができる。
＜src2＞は、３つのオプションの短縮形として使用される。

－　〔Rn｜Rn. 1｜Rn. h｜Rn. x〕〔^〕

の形のソース・レジスタ、プラス最終結果のスケール（＜scale＞）。
－　オプションでシフトされた８ビット定数（＜immed_8＞）、ただし、最終結果のスケールではない。
－　６ビット定数（＜immed_6＞）、プラス、最終結果のスケール（＜scale＞）。
＜src2_maxmin＞は、＜src2＞と同じであるが、ただし、スケールは許可されない。
＜src2_shift＞シフト命令は、＜src2＞の限定的サブセットを提供する詳細は上記を参照。
＜src2_par＞＜src2_shift＞用である。
第３のオペランドを指定する命令に対して：
＜acc＞は、４つのアキュミュレータ・レジスタ

〔A0｜A1｜A2｜A3〕

のいずれかを示す短縮形。48ビットすべてが読まれる。再充填は指定されない。
目的レジスタは次のフォーマットを持つ：
＜dest＞これは、

〔Rn｜Rn. 1｜Rn. h｜. 1｜〕〔^〕

の短縮形。“ ."の拡張はない。
レジスタ全部が書かれる（アキュミュレータの場合は、48ビット）。レジスタへの書き戻しが必要ない場合は、使用されるレジスタは重要でない。アセンブラが、目的レジスタの省略をサポートし、書き戻しの必要がないこと、又は“ .1"つまり、書き戻しは必要ないが結果が16ビット量であるかのようにフラグをセットすべきであることを示す。^は、値が出力FIFOに書き込まれることを示す。

＜scale＞これは、代数スケールの数を表す。14のスケールが使用できる。
ASR＃0,1,2,3,4,6,8,10
ASR＃12から16
LSL＃１
＜immed_8＞これは、符号無し８ビット即値を表す。これは、０、８、16、又は24シフトで左回転された１バイトから成る。従って、0xYZ000000,0x00YZ0000、0x0000YZ00、0x000000YZの値が、任意のYZに対してエンコードできる。回転は、２ビット量としてエンコードされる。
＜imm_6＞これは、符号無し６ビット即値を表す。
＜PARAMS＞これは、レジスタ・リマッピングを指定し、次のフォーマットを持つ：＜BANK＞＜BASIC＞ｎ＜RENUMBER＞ｗ＜BASEWRAP＞

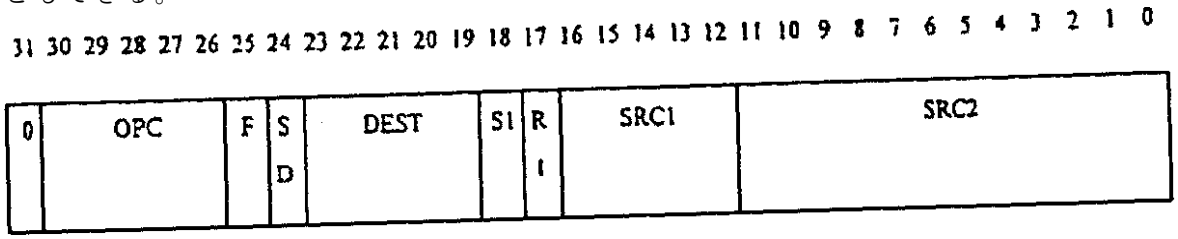
＜ＢＡＮＫ＞	〔Ｘ｜Ｙ｜Ｚ〕を取れる
＜ＢＡＳＥＩＮＣ＞	〔＋＋｜＋１｜＋２｜＋４〕を取れる
＜ＲＥＮＵＭＢＥＲ＞	〔０｜２｜４｜８〕を取れる
＜ＢＡＳＥＷＲＡＰ＞	〔２｜４｜８〕を取れる

＜cond＞という表現は、以下の条件コードの任意の１つの短縮形である。尚、エンコーディングは、ARMと少し異なる。それは、符号無しLS及びHIコードは、より役立つ符号付きオーヴァーフロー／アンダーフローのテストで置き換えられているからである。Ｖフラグ及びＮフラグは、ピッコロ上で、ARMとは違う方法でセットされるので、条件テストからフラグ・チェックへの翻訳も、ARMとは異なる。

0000	ＥＱ	Ｚ＝０	最後の結果はゼロであった。
0001	ＮＥ	Ｚ＝１	最後の結果は非ゼロであった。
0010	ＣＳ	Ｃ＝１	shift/ＭＡＸ操作の後で使用される。
0011	ＣＣ	Ｃ＝０	
0100	ＭＩ／ＬＴ	Ｎ＝１	最終結果が負であった。
0101	ＰＬ／ＧＥ	Ｎ＝０	最終結果が正であった。
0110	ＶＳ	Ｖ＝１	最終結果で符号付きオーバーフロー／飽和
0111	ＶＣ	Ｖ＝０	最終結果でオーバーフロー／飽和なし
1000	ＶＰ	Ｖ＝１　＆　Ｎ＝０	最終結果でオーバーフロー正
1001	ＶＮ	Ｖ＝１　＆　Ｎ＝１	最終結果でオーバーフロー負
1010	未使用		
1011	未使用		
1100	ＧＴ	Ｎ＝０　＆　Ｚ＝０	
1101	ＬＥ	Ｎ＝１｜Ｚ＝１	
1110	ＡＬ		
1111	未使用		

ピッコロが扱うのは符号付き量であるから、符号無しLS及びHI条件は、落とされ、オーバーフローの方向を記述するVPとVNで置き換えられている。ALUの結果は48ビット幅であるから、MIとLTが、同様にPLとGEが同じ機能を行う。
すべての操作は、特に注意書のない限り、符号付きである。
第１条件コード及び第２条件コードは、それぞれ、次のものから成る。
Ｎ　負
Ｚ　ゼロ
Ｃ　キャリー／符号無しオーバーフロー
Ｖ　符号付きオーバーフロー
算術命令は、並列命令と「フル幅」命令の２つに分けることができる。「フル幅」命令というのは、一次フラグをセットするだけであるのに対して、並列オペレータは、結果の上16ビット半分と下16ビット半分とに基づき、一次フラグと２次フラグをセットする。
N,Z,Vフラグは、スケールを適用した後に、目的に書き込まれる前に、ALUの結果に基づいて計算される。ASRは常に、結果を記憶するのに必要なビット数を減らすが、ASLだと、それを増やす。これを避けるために、ピッコロは、ASLスケールが適用された場合、48ビットの結果を削って、ゼロ検出及びオーバーフローが行われるビット数を制限する。
Ｎフラグの計算は、符号付き算術計算が行われると推定して、行われる。それは、オーバーフローが起きた場合、結果の最上位ビットはＣフラグかＮフラグであり、それは、入力オペランドが符合付きか符号無しかによるからである。
Ｖフラグは、選択された目的に結果を書き込んだ結果、精度の損失があるか否かを示す。書き戻しが選択されなかった場合も、「サイズ」は含まれており、オーバーフロー・フラグは正しくセットされる。オーバーフローが起きるのは、次の場合である。
－　結果が、－2^15から2^15－１の範囲にないのに16ビットレジスタに書き込んだ場合。
－　結果が、－2^31から2^31－１の範囲にないのに32ビットレジスタに書き込んだ場合。
並列加算／減算命令は、結果の上半分及び下半分に独立にN,Z,Vフラグをセットする。
アキュミュレータに書き込みを行うと、32ビットレジスタに書き込まれたかのように、Ｖフラグがセットされる。
飽和絶対命令（SABS）も、入力オペランドの絶対値が指定された目的に合わないと、オーバーフロー・フラグをセットする。
キャリー・フラグは、加算と減算命令によりセットされ、MAX/MIN,SABS、CLB命令によって「バイナリー」フラグとして使用される。乗算操作を含む他のすべての命令は、（単数または複数の）キャリー・フラグを保存する。
加算と減算操作については、キャリーは、ビット31又はビット15又は目的が32ビット幅であるか16ビット幅であるかの、結果によって生成される。
標準的算術命令は、フラグのセット方法によって、多くのタイプに分類することができる。
加算命令、減算命令の場合、Ｎビットがセットされると、すべてのフラグが保存される。
Ｎビットがセットされないと、フラグは、次のように更新される。
Ｚがセットされるのは、フル48ビット結果が０だった場合。
Ｎがセットされるのは、フル48ビット結果にビット47のセットがあった場合（負だった場合）。
Ｖがセットされるのは：
目的レジスタが16ビットであり、符号付き結果が16ビットレジスタに合わない（－2^15＜＝ｘ＜2^15の範囲にない）場合
目的レジスタが32/48ビットレジスタであり、符号付き結果が32ビットに合わない場合
＜dest＞が32又は48ビットレジスタである場合でＣフラグがセットされるのは、＜scr1＞と＜scr2＞を合計してビット31からキャリーがある時、又は、＜scr1＞から＜scr2＞を減算してビット31から借り（borrow）が生じない時（ARM上と同じキャリー）。＜dest＞が16ビットレジスタである場合でＣフラグがセットされるのは、合計のビット15からキャリーがある時。

２次フラグ（SZ,SN,SV,SC）は保存される。
48ビットレジスタから乗算又は累算を行う命令の場合。
Ｚがセットされるのは、フル48ビット結果が０だった場合。
Ｎがセットされるのは、フル48ビット結果にビット47のセットがあった場合（負だった場合）。
Ｖがセットされるのは：（１）目的レジスタが16ビットであり、符号付き結果が16ビットレジスタに合わない（－2^15＜＝ｘ＜2^15の範囲にない）場合、（２）目的レジスタが32/48ビットレジスタであり、符号付き結果が32ビットに合わない場合
Ｃは保存される。
２次フラグ（SZ,SN,SV,SC）は保存される。
論理操作、並列加算及び減算、maxおよびmin、シフトなどを含むその他の命令は、以下のようにカバーされる。
加算命令、減算命令は、２つのレジスタを加算又は減算し、結果をスケールして、レジスタに戻して記憶させる。オペランドは、符号付き値として扱われる。不飽和変種に対するフラグ更新は、オプショナルであり、Ｎを命令の最後に付け足すことによって抑制することもできる。



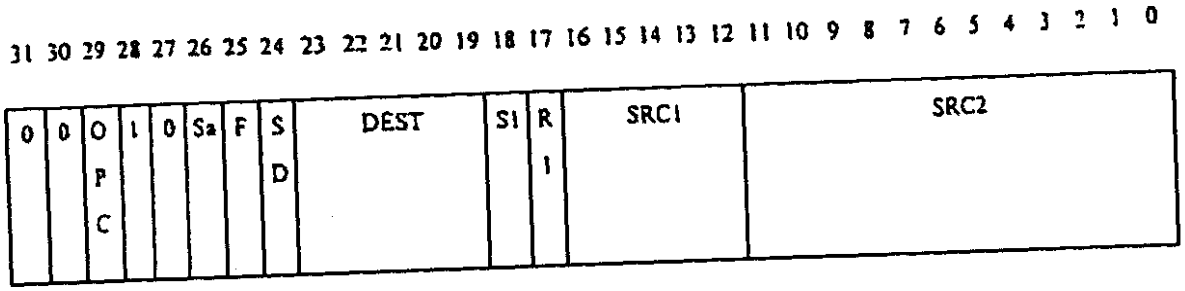
OPCは、命令のタイプを指定する。

100N0	dest=(src1 + src2) (－＞＞ scale) (,N)
110N0	dest=(src1 - src2) (－＞＞ scale) (,N)
10001	dest=SAT((src1 + src2) (－＞＞ scale))
11001	dest=SAT((src1 - src2) (－＞＞ scale))
01110	dest=(src2 - src1) (－＞＞ scale)
01111	dest=SAT((src2 - src1) (－＞＞ scale))
101N0	dest=(src1 + src2 + Carry) (－＞＞ scale) (,N)
111N0	dest=(src1 - src2 + Carry-1) (－＞＞ scale) (,N)

ニューモニックス：

100N0	ＡＤＤ {N}	＜dest＞, ＜src1＞, ＜src2＞ {,＜scale＞}
110N0	ＳＵＢ {N}	＜dest＞, ＜src1＞, ＜src2＞ {,＜scale＞}
10001	ＳＡＤＤ	＜dest＞, ＜src1＞, ＜src2＞ {,＜scale＞}
11001	ＳＳＵＢ	＜dest＞, ＜src1＞, ＜src2＞ {,＜scale＞}
01110	ＲＳＢ	＜dest＞, ＜src1＞, ＜src2＞ {,＜scale＞}
01111	ＳＲＳＢ	＜dest＞, ＜src1＞, ＜src2＞ {,＜scale＞}
101N0	ＡＤＣ {N}	＜dest＞, ＜src1＞, ＜src2＞ {,＜scale＞}
111N0	ＳＢＣ {N}	＜dest＞, ＜src1＞, ＜src2＞ {,＜scale＞}

アセンブラは以下のオペコードをサポートする
CMP＜src1＞, ＜src2＞
CMN＜src1＞, ＜src2＞
CMPは、レジスタ書き込みディスエーブル（disabled）のフラグをセットする減算であり、CMNは、レジスタ書き込みディスエーブルのフラグをセットする加算である。
フラグ：
これについては、上記の通り。
含める理由
ADCは、shift/MAX/MIN操作に続いてレジスタの下にキャリーを挿入するのに使える。また、32/32割算を行うのにも使用される。さらに、拡張精密加算を提供する。Ｎビットを加算することによって、フラグを細かく制御することができ、特にキャリーを制御できる。これにより、１ビットにつき２サイクルで、32/32ビット割算ができる。
飽和加算及び減算が、G.729などに必要である。
カウンタのインクリメント／ディクリメント。RSBは計算シフト（ｘ＝32－ｘが普通の操作）に使える。飽和RSBは、飽和否定（G.729で使用される）に必要である。
加算／減算累算命令は、累算及びスケーリング／飽和を伴う加算及び減算を行う。乗算累算命令と違って、アキュミュレータ番号は、目的レジスタと独立に指定することはできない。目的レジスタの下２ビットは、累算に使う48ビットアキュミュレータの番号、accを与える。従って、ADDA X0,X1,X2,A0及びADDA A3,X1,X2,A3は有効であるが、ADDA X1,X1,X2,A0は無効である。このクラスの命令では、結果はレジスタに書き戻されなければならず、目的領域の書き戻し無しエンコーディングは許可されない。



OPCは、命令のタイプを指定する。以下において、accは（DEST〔1:0〕）である。Saビットは、飽和を示す。

動作（OPC）：

0　dest = {SAT} (acc + (src1 + src2))　{−>> scale}

1　dest = {SAT} (acc + (src1 - src2))　{−>> scale}

ニューモニック：

0　{S} ADDA　<dest>, <src1>, <src2>, <acc >　{,<scale>}

1　{S} SUBA　<dest>, <src1>, <src2>, <acc >　{,<scale>}

コマンドの前のSは飽和を示す。

フラグ：

上記を参照

含める理由

ADDA（加算累算）命令は、１サイクルにつき、整数アレーの２ワードとアキュミュレータ（例えば、それらの平均を見つけるのに）の和を取るのに使える。SUBA（減算累算）命令は、差の和を計算するのに（例えば相関のために）使え、２つの別個の値を減算して、その差を第３のレジスタに加える。

＜acc＞とは異なる＜dest＞を使用することによって、丸め（rounding）をともなう加算をすることもできる。例えば、X0＝（X1＋X2＋16384）＞＞15は、16384をA0に保持しながら１サイクルで行うことができる。丸め付定数加算は、ADDA　X0,X1,＃16384,A0で行うことができる。

ビットの正確な導入には：

sum of（（a_i* b_i）＞＞k）（一般的にはTrueSpeechで使用される）

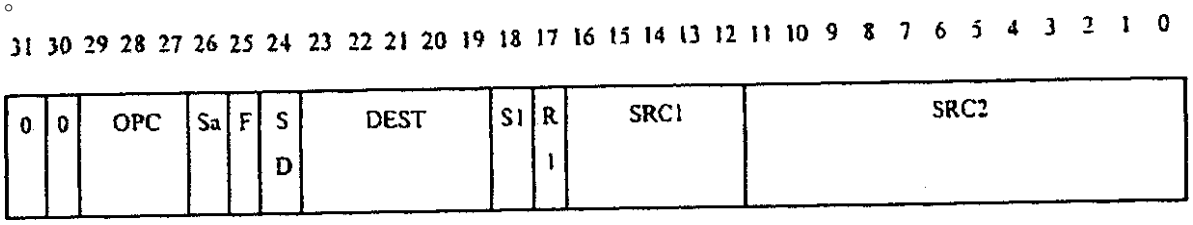
標準ピッコロ・コードは以下のようになる：

```
MUL    t1,  a__0, b__0, ASR#k
ADD    ans, ans, t1
MUL    t2,  a__1, b__1, ASR#k
ADD    ans, ans, t2
```

このコードには２つの問題がある。１つは長すぎること、もう１つは、加算が48ビット精密加算ではなくガードビットが使用できないこと。これに対処するには、ADDAを使うことである。

```
MUL    t1,  a__0, b__0, ASR#k
MUL    t2,  a__1, b__1, ASR#k
ADDA   ans, t1, t2, ans
```

これにより、25%のスピードアップが得られる、48ビット精度が保持される。
並列命令における加算／減算は、32ビットレジスタに対（ペア）で保持される２つの符号付き16ビット量で行われる。一次条件コードフラグは、最上位16ビットの結果からセットされ、二次フラグは、下位半分から更新される。これらｈの命令のソースとして指定できるのは32ビットレジスタだけであるが、値は、ハーフワード交換できる。各レジスタの個々の半分は、符号付き値として扱われる。計算及びスケーリングは、精度損失無しで行われる。従って、ADDADD X0,X1,X2,ASR＃１は、X0の上半分及び下半分における正しい平均を生成する。各命令にはオプショナル飽和が提供され、それには、Saビットをセットする。



OPCが操作を定義する。
動作（OPC）：

```
000   dest.h=(src1.h + src2.h)－＞＞ {scale } ,
  dest.1=(src1.1 + src2.1)－＞＞ {scale }
001   dest.h=(src1.h + src2.h)－＞＞ {scale } ,
  dest.1=(src1.1 - src2.1)－＞＞ {scale }
100   dest.h=(src1.h - src2.h)－＞＞ {scale } ,
  dest.1=(src1.1 + src2.1)－＞＞ {scale }
101   dest.h=(src1.h - src2.h)－＞＞ {scale } ,
  dest.1=(src1.1 - src2.1)－＞＞ {scale }
```

Saビットがセットされている場合、各和／差分は独立に飽和する。
ニューモニック：

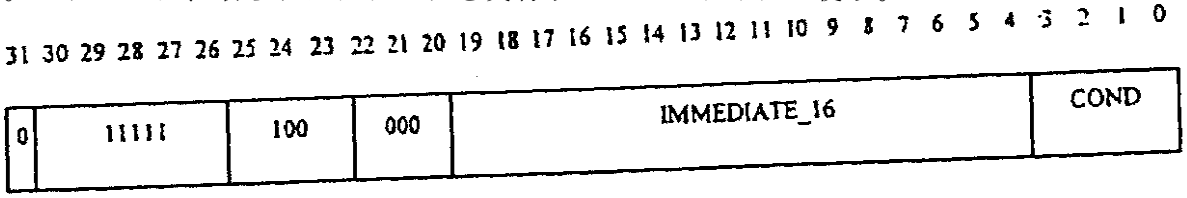
```
000  {S} ADDADD  <dest>, <src1_32>, <src2_32> {,<scale>}
001  {S} ADDSUB  <dest>, <src1_32>, <src2_32> {,<scale>}
100  {S} SUBADD  <dest>, <src1_32>, <src2_32> {,<scale>}
101  {S} SUBSUB  <dest>, <src1_32>, <src2_32> {,<scale>}
```

コマンドの前のＳは飽和を示す。
アセンブラは以下のものもサポートする

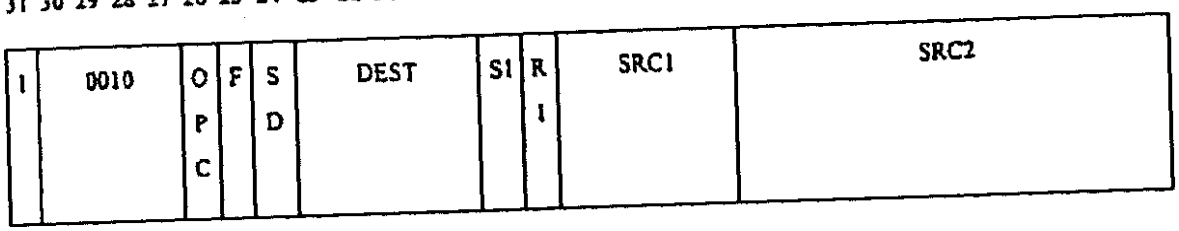
```
CMNCMN     <dest>, <src1_32>, <src2_32> {,<scale>}
CMNCMP     <dest>, <src1_32>, <src2_32> {,<scale>}
CMPCMN     <dest>, <src1_32>, <src2_32> {,<scale>}
CMPCMP     <dest>, <src1_32>, <src2_32> {,<scale>}
```

書き戻しなしの標準命令によって生成される。
フラグ

Ｃがセットされるのは、２つの上の16ビット半分を加算する時に、ビット15のキャリーがある場合。
Ｚ　がセットされるのは、上の16ビット半分の和が０である場合。
Ｎ　がセットされるのは、上の16ビット半分の和が負である場合。
Ｖ　がセットされるのは、上の16ビット半分の符号付き17ビット和が16ビットに当てはまらない（ポスト・スケール）場合。
SZ,SN,SV,SCが、同様に、下の16ビット半分に対してセットされる。
含める理由
並列加算及び減算命令は、単一32ビットレジスタに保持される複素数を操作するのに使用でき、FFTカーネルで使用される。また、16ビットデータのベクトルの単純な加算／減算にも使え、１サイクルで２つの要素を処理することができる。
ブランチ（条件付き）命令は、制御フローにおける条件付き変更を行うことを可能とする。ピッコロは、取られたブランチを実行するのに３サイクル使う。



動作
一次フラグに基づき＜cond＞が保持されれば、オフセットによるブランチ。
オフセットは、符号付き16ビット番号のワードである。この時、オフセットの範囲は、－32768から＋32767ワードに制限される。
アドレス計算は次のようにされる。
目的アドレス＝ブランチ命令アドレス＋４＋オフセット
ニューモニック：
Ｂ＜cond＞＜destination_label＞
フラグ：
影響されない
含める理由：
殆どのルーチンで非常に役立つ。
条件付き加算又は減算命令は、条件付きでsrc2をsrc1へ加算または減算する。



OPCが命令のタイプを指定する。
動作（OPC）：

0 if(carry set)temp=src1-src2 else temp=src1+src2

dest = temp {－＞＞ scale}

1 if(carry set)temp=src1-src2 else temp=src1+src2

dest = temp {－＞＞ scale} BUT if scale is a shift left

(もしスケールが左シフトなら)

then the new value of carry(from src1-src2 or src1+src2)is shifted into the bottom.

(src1-src2 又は src1+src2からの) キャリーの新しい値がボトムにシフトされる)

ニューモニック：

0 ＣＡＳ ＜dest＞, ＜src1＞, ＜src2＞ {,＜scale＞}

1 ＣＡＳＣ ＜dest＞, ＜src1＞, ＜src2＞ {,＜scale＞}

フラグ：

上記参照

含める理由

条件付き加算または減算命令により、効率のよい除算コードを構成することができる。

例1:X0にある32ビット符号無し値を、X1にある16ビット符号無し値で割る（X0＜（X1＜＜16）且つX1.h＝０と仮定する）。

```
ＬＳＬ　Ｘ１，Ｘ１，＃１５　　；除数をシフトアップする
ＳＵＢ　Ｘ１，Ｘ１，＃０　　；キャリーフラグをセットする
ＲＥＰＥＡＴ　＃１６
ＣＡＳＣ　Ｘ０，Ｘ０，Ｘ１，ＬＳＬ＃１
ＮＥＸＴ
```

ループの最後で、X0.1は除算の商を保持する。余りは、キャリーの値に従って、X0.hから復元される。

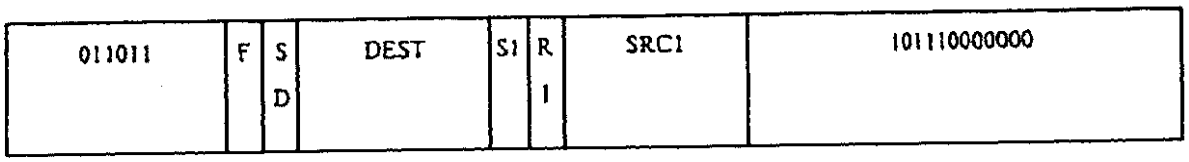
例2:X0にある32ビット正の値を、X1にある32ビット正の値で割り、早く終了する。

```
MOV     X2,#0          ;商をクリアする
LOG     Z0,X0          ;ビット数Ｘ０分シフトできる
LOG     Z1,X1          ;ビット数Ｘ１分シフトできる
SUBS    Z0,Z1,Z0       ;Ｘ１がシフトして１がマッチする
BLT     div__end       ;Ｘ１＞Ｘ０なので答は０
LSL     X1,X1,Z0       ;マッチした場合
ADD     Z0,Z0,#1       ;行うべきテストの回数
SUBS    Z0,Z0,#0       ;キャリーをセットする
REPEAT  Z0
CAS     X0,X0,X1,LSL#1
ADCN    X2,X2,X2
NEXT
```

div__end

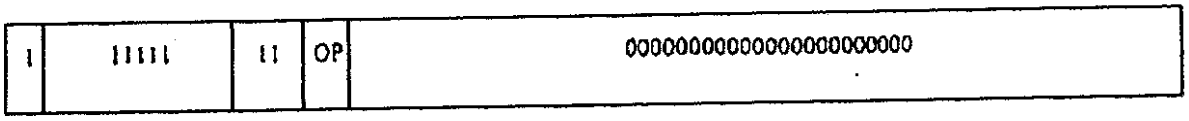
最後に、X2が商を保持し、余りは、X0から復元される。
カウント・リーディング・ビット命令により、データが正規化される。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0



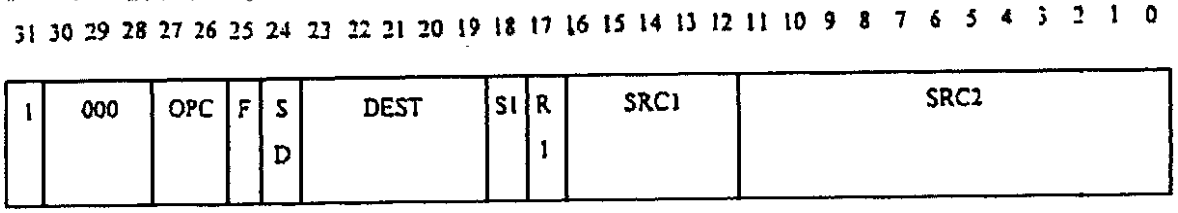
動作
destは、src1にある値が左にシフトされるべき場所数にセットされて、ビット31がビット30と異なるようにする。これは０－30の範囲の値であるが、例外として、src1が－１又は０の場合は、31が戻される。
ニューモニック
CLB＜dest＞, ＜src1＞
フラグ
Ｚ　がセットされるのは、結果が０の時。
Ｎ　はクリアされる。
Ｃ　がセットされるのは、src1が－１又は０の時。
Ｖ　は未使用。
含む理由：
正規化に必要なステップ
ピッコロの実行を止めるには、Halt及びBreakpoint命令がある。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0



OPCは命令のタイプを指定する。

動作（OPC）
０　ピッコロの実行が止められ、Haltビットがピッコロ状態レジスタにセットされる。
１　ピッコロの実行が止められ、Breakビットがピッコロ状態レジスタにセットされ、ARMが中断され、ブレークポイントに到達したことを知らせる。
ニューモニック
０　HALT
１　BREAK
フラグ
影響されない。
論理演算命令は、32又は16ビットレジスタ上で論理演算を行う。オペランドは、符号無し値として扱われる。



OPCは、実行すべき論理操作をエンコードする。
動作（OPC）：

00　dest = (src1 & src2) {−>> scale}

01　dest = (src1 | src2) {−>> scale}

10　dest = (src1 &～src2) {−>> scale}

11　dest = (src1 ^ src2) {−>> scale}

ニューモニック：

00　AND　<dest>, <src1>, <src2>, {,<scale>}

01　ORR　<dest>, <src1>, <src2>, {,<scale>}

10　BIC　<dest>, <src1>, <src2>, {,<scale>}

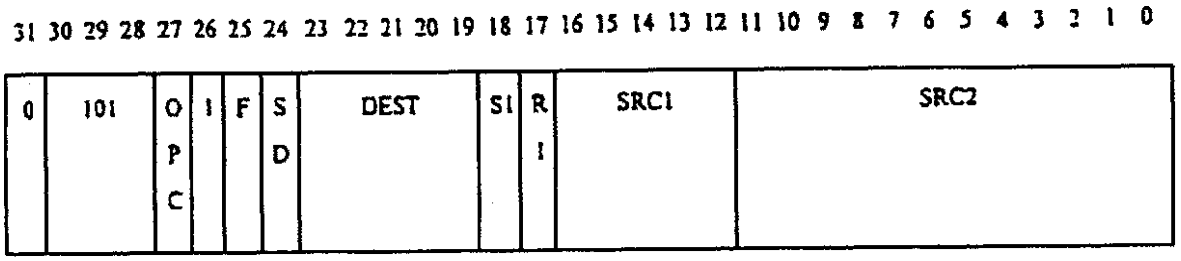
11　EOR　<dest>, <src1>, <src2>, {,<scale>}

アセンブラが以下のオペコードをサポートする

TST　<src1>, <src2>

TEQ　<src1>, <src2>

TSTは、レジスタ書き込みがディスエーブルされたANDである。TEQはレジスタ書き込みがディスエーブルされたEORである。
フラグ
Ｚ　がセットされるのは、結果が全て０の時。
N,C,Vは保存される。
SZ,SN,SC,SVは保存される。
含む理由：
スピーチ圧縮アルゴリズムは、情報をエンコードするために、パックされたビット領域を使用する。ビットマスク命令は、これらの領域の抽出／パック化を助ける。
Max及びMin操作命令は、最大及び最小操作を実行する。



OPCは命令のタイプを指定する。

動作（OPC）：

0　dest = (src1<＝src2) ?src1 : src2

1　dest = (src1>src2) ?src1 : src2

ニューモニック：

0　MIN　<dest>, <src1>, <src2>

1　MAX　<dest>, <src1>, <src2>

フラグ

Z　がセットされるのは、結果が０の時。

N　がセットされるのは、結果が負の時。

C　Maxでは、src2＞＝src1（dest＝src1の場合）の時にセットされる。Minでは、src2＞＝src1（dest＝src2の場合）の時にセットされる。

V　保存される

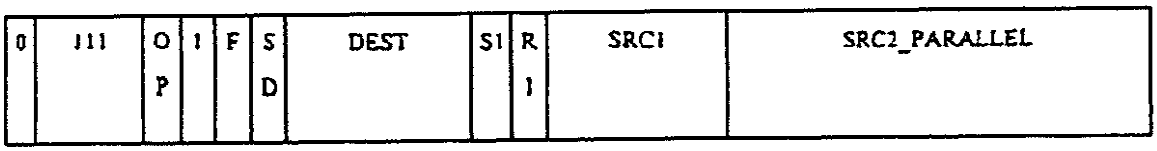
含む理由：

信号の強さを見るために、多数のアルゴリズムがサンプルをスキャンして、サンプルの絶対値の最大／最小を決める。これに、MAX,MIN操作が使用できる。信号の最初の最大値か最後の最大値のどちらかを見つけたいかによって、オペランドsrc1及びsrc2は、交換することができる。

MAX　X0,X0,＃０は、X0を正の数に変換し下をクリップする。

MIN　X0,X0,＃255は、X0の上をクリップする。これは、グラフィック処理に役立つ。

並列命令におけるMAX,MIN操作は、並列16ビットデータ上で最大値最小値操作を行う。



OPCは、命令のタイプを指定する。

動作（OPC）：

0　dest.l= (src1.l<＝src2.l) ?src1.l : src2.l

　　dest.h= (src1.h<＝src2.h) ?src1.h : src2.h

1　dest.l= (src1.l>src2.l) ?src1.l : src2.l

　　dest.h= (src1.h>src2.h) ?src1.h : src2.h

ニューモニック：

0　MINMIN　<dest>, <src1>, <src2>

1　MAXMAX　<dest>, <src1>, <src2>

フラグ

Z　がセットされるのは、結果の上16ビットがゼロの場合。

Ｎ　がセットされるのは、結果の上16ビットが負の場合。
Ｃ　Max:src2.h＞＝scrc1.h（dest＝src1の場合）の時にセットされる。
Min:src2.h＝scrc1.h（dest＝src2の場合）の時にセットされる。
Ｖ　保存される
SZ,SN,SC,SVは、同様に、下16ビット半分用にセットされる。
含む理由：
32ビットMax,Minについて。
Move Long Immeidate Operation命令により、レジスタは、どの符号付き16ビットの符号拡張値をセットされることができる。これらの命令のうち２つは、32ビットレジスタに任意の値にセットすることができる（連続する高位半分と低位半分にアクセスすることによって）。レジスタ間の移動については、選択操作を参照。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

I	11100	F	S D	DEST	IMMEDIATE_15	+/-	000

MOV<dest>, #< imm__16>

アセンブラは、MOV命令を使用して非インターロックNOP操作を提供することができる。つまり、NOPは、MOV,＃０と等価である。
フラグ
フラグは影響されない。
含む理由：
レジスタ／カウンタをイニシアライズする。
乗算累算操作命令は、符号付き乗算を行い、累算または退出（deaccumulation）、スケーリング及び飽和を伴う。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

I	10	OPC	Sa	F	S D	DEST	A I	R I	SRC1	SRC2_MULA

OPC領域は命令のタイプを特定する。
動作（OPC）：
00　dest = (acc +(src1* src2)) {－＞＞ scale}

01　dest = (acc -(src1* src2)) {－＞＞ scale}

各場合、Saビットがセットされていれば、結果は目的に書き込まれる前に飽和される。
ニューモニック：
00　{S} MULA <dest>, <src1 __16>,<src2 __16>,<acc> {,<scale>}

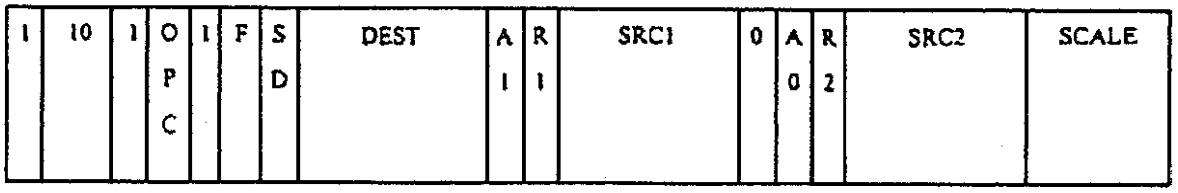
01　{S} MULS <dest>, <src1 __16>,<src2 __16>,<acc> {,<scale>}

コマンドの手前のＳは飽和を示す。
フラグ：
上記を参照。
含む理由：
１サイクル保持されたMULAがFIRコードに必要である。MULSは、FFTバタフライで使用される。また、MULAは、丸め（rounding）付き乗算に役立つ。例えば、A0＝（X0＊X1＋16384）＞＞15は、16384を別のアキュミュレータ（例えばA1）に保持することによって、１つ

のサイクルで行うことができる。FFTカーネルには異なった＜dest＞及び＜acc＞が必要である。

Multiply Double Operation命令は、符号付き乗算を行い、結果をダブルにしてから累算又は退出、スケーリング、飽和を行う。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0



OPCは命令のタイプを指定する。

動作（OPC）：

0　dest=SAT ((acc + SAT (2* src1* src2)) {−>> scale}

1　dest=SAT ((acc - SAT (2* src1* src2)) {−>> scale}

ニューモニック：

0　SMLDA　<dest>,<src1_16>, <src2_16>,<acc> {,<scale>}

1　SMLDS　<dest>,<src1_16>, <src2_16>,<acc> {,<scale>}

フラグ：

上記参照

含む理由：

MLD命令は、G.729など、分数（fractional）算術を使用するアルゴリズムにとって必要である。殆どのDSPは、累算又は書き戻しの前に乗数の出力において１ビット左にシフトさせることのできる分数モードを提供する。これを特定命令としてサポートすることにより、プログラマにはより大きなフレキシビリティーが与えられる。Ｇシリーズの基本操作のいくつかと同等の名前を以下に示す。

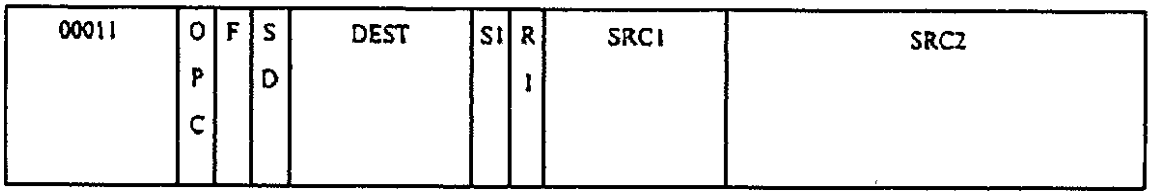
L_msu ＝＞　SMLDS

L_mac ＝＞　SMLDA

これらは、１ビット左シフトする時に乗数の飽和を利用する。一連の分数の乗算・累算が必要な場合、精度のロスなしに、MULAを使うことができ、その和は、33.14フォーマットで保持される。必要なら、左シフト及び飽和を最後に利用して、1.15フォーマットに変換することができ８る。

乗算演算命令は、符号付き乗算、及びオプショナルなスケーリング／飽和を行う。ソース・レジスタ（16ビットのみ）は、符号付き数として扱われる。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0



OPCは命令のタイプを指定する。

動作（OPC）：

0　dest= (src1* src2)　{−>> scale}

1　dest=SAT　(src1* src2)　{−>> scale}

ニューモニック：

0 MUL <dest>, <src1_16>, <src2> {,<scale>}

1 SMUL <dest>, <src1_16>, <src2> {,<scale>}

フラグ：
上記を参照。
含む理由。
符号付き且つ飽和乗算は、多くの処理で必要となる。
Register List操作は、複数のレジスタのセット（集合）に操作を行う時に使用される。Empty and Zero命令は、ルーチンを始める前に、あるいはルーチンとルーチンとの間で、レジスタの選択をリセットするのに使用する。Output命令を使って、レジスタのリストの内容を出力FIFOに記憶することができる。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	11111	0	OPC	00	REGISTER_LIST_16	SCALE

OPCは命令のタイプを指定する。
動作（OPC）：

000 (k=0; k<16; k++)用

if bit k of the register list is set then register k is marked as being empty. (レジスタリストのビットｋがセットされると、レジスタｋは空きの印になる。)

001 (k=0; k<16; k++)用

if bit k of the register list is set then register k is set to contain 0. (レジスターリストのビットｋがセットされると、レジスタｋが０を含むようセットされる。)

010 未定義

011 未定義

100 (k=0; k<16; k++)用

if bit k of the register list is set then (レジスタリストのビットｋがセットされると、) (register k->>scale)is written to the output FIFO. (が出力ＦＩＦＯに書き込まれる。)

101 (k=0; k<16; k++)用

if bit k of the register list is set then (レジスタリストのビットｋがセットされると、) (register k->>scale)is written to the output FIFO and register k is marked as being empty. (が出力ＦＩＦＯに書き込まれ、レジスタｋが空きの印になる。

110 (k=0; k<16; k++)用

if bit k of the register list is set then (レジスタリストのビットｋがセットされると、) ＳＡＴ (register k->>scale)is written to the output FIFO.(が出力ＦＩＦＯに書き込まれる。)

111 (k=0; k<16; k++)用

if bit k of the register list is set then (レジスタリストのビットｋがセットされると、) ＳＡＴ (register k->>scale)is written to the output FIFO and register k is marked as being empty. (が出力ＦＩＦＯに書き込まれ、レジスタｋが空きの印になる。)

ニューモニック：

000 EMPTY <register_list>

001 ZERO <register_list>

010 Unused

011 Unused

100 OUTPUT <register_list> {,<scale>}

101 OUTPUT <register_list>^ {,<scale>}

110 SOUTPUT <register_list> {,<scale>}

111 SOUTPUT <register_list>^ {,<scale>}

フラグ
影響されない
例
EMPTY {A0, A1, X0−X3}

ZERO {Y0−Y3}

OUTPUT {X0−Y1}^

また、アセンブラはシンタクス（文法）をサポートする。

OUTPUT Rn

その場合、MOV^,Rn命令を使ってレジスタを１つ出力することになる。
EMPTY命令は、空であるすべてのレジスタが有効データを含む（すなわち、空きでない）まで、止まっている。
リマッピングREPEATループ内では、レジスタ・リスト操作は使用されるべきでない。
OUPTUT命令が出力用に指定することができるレジスタは８つまでである。
含む理由：
１つのルーチンが終了した後、次のルーチンは、ARMからデータを受け取れるようすべてのレジスタが空きであることを期待する。これを遂行するために、EMPTY命令が必要となる。FIRそのたのフィルタを実行する前に、すべてのアキュミュレータ及び部分的結果がゼロにされなければならない。これには、ZERO命令が助けとなる。これらの命令は、一連の単一レジスタ移動を置き換えることによってコード密度を改善するよう設計されている。OUTPUT命令は、一連のMOV^,Rn命令を置き換えることによってコード密度を改善するべく含まれる。
リマッピング・パラメータ・移動命令RMOVが提供されるので、ユーザ定義のレジスタ・リマッピング・パラメータの構成を取ることができる。
命令エンコーディングは以下の通り。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	11111	101	00	ZPARAMS	YPARAMS	XPARAMS

各PARAMS領域は次のエントリから成る：

6 5 4 3 2 1 0

BASEWRAP	BASEINC	0	RENUMBER

これらのエントリの意味を以下に示す。

パラメータ	説　　明
RENUMBER	リマッピングを行う１６ビットレジスタの数であって、値０，２，４，８を取ることができる。 ＲＥＮＵＭＢＥＲ未満のレジスタはリマップされ、それ以上のレジスタは直接アクセスされる。
BASEINC	各ループの最後でベースポインタがインクリメントされる量。
BASEWRAP	ベースラッピング・モジュラスが取れる値は2，4，８。

ニューモニック：

RMOV＜PARAMS＞，［＜PARAMS＞］

＜PARAMS＞領域は次のフォーマットを取る。

＜PARAMS＞::＝＜BANK＞＜BASEINC ＞n＜RENUMBER＞w＜BASEWRAP＞

＜BANK＞::＝［X｜Y｜Z］

＜BASEINC＞::＝［++｜＋1｜＋2｜＋4］

＜RENUMBER＞::＝［0｜2｜4｜8］

＜BASEWRAP＞::＝［2｜4｜8］

RMOV命令の使用がリマッピングのアクティブ中だと、その挙動は、UNPREDICATABLE（予測不可）である。

フラグ

影響されない。

Repeat命令は、４つのゼロ・サイクル・ループをハードウェアで提供する。REPEAT命令は、新しいハードウェア・ループを定義する。ピッコロは、最初のREPEAT命令にハードウェア・ループ０を使用し、最初のrepeat命令に埋め込まれた（nested）REPEAT命令にハードウェア・ループ１を使用し、以下同様である。REPEAT命令は、どのループが使用されているかを指定する必要はない。REPEAT命令は厳密に埋め込まれなければならない。深さ４を越える埋め込みを試みると、挙動は、予想不可となる。

各REPEAT命令は、（REPEAT命令の直後の）ループ内の命令の数を指定し、そのループを何回巡るかの回数（定数またはピッコロレジスタから読み込まれる）を指定する。

ループ内の命令の数が小さい（１又は２）場合、ピッコロはそのループをセットアップするために余分のサイクルを使っても良い。

ループ・カウントがレジスタ指定であれば、32ビットアクセスという意味になる（S1＝１）が、下の16ビットだけが意味を持ち、その数は符号無しであるとされる。ループ・カウントがゼロの場合、ループの動作は未定義である。ループ・カウントのコピーが取られ、レジスタはループに影響せずに直接再利用（又は、再充填さえ）できる。

REPEAT命令は、ループ内でレジスタ・オペランドが指定される方法を変えるメカニズムを提供する。詳細は上記の通り。

ループ数がレジスタ指定されたREPEATのエンコーディング：

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	11110	0	RFIELD_4	00	0	R 1	SRC1	0000	#INSTRUCTIONS_8

固定されたループ数のREPEATのエンコーディング：

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	11110	1	RFIELD_4	#LOOPS_13	#INSTRUCTIONS_8

RFIELDオペランドは、ループ内でどの16リマッピングパラメータ構成を使用すべきかを指定する。

RFIELD	リマッピング操作
0	リマッピングは行われない
1	ユーザ定義のリマッピング
2..15	プリセット・リマッピング構成TBD

アセンブラは、ハードウェア・ループを定義するためにREPEATとNEXTという２つのオペコードを提供する。REPEATはループの始めに行き、NEXTはループの最後を区切ることによって、アセンブラはループ本体内にある命令の数を数えることができる。REPEATにとって必要なことは、ループの数を定数あるいはレジスタとして指定すればよいだけである。例えば：

```
REPEAT    X0
MULA      A0, Y0.l, Z0.l, A0
MULA      A0, Y0.h^, Z0.h^, A0
NEXT
```

これは、２つのMULA命令をX0回実行する。また、

```
REPEAT    #10
MULA      A0, X0^, Y0^, A0
NEXT
```

は、10回乗算累算を行う。

アセンブラは、次のシンタクス（文法）をサポートする。

REPEAT#iterations [, ＜PARAMS＞]

REPEATのために使用するリマッピング・パラメータを指定する。必要なリマッピング・パラメータが前もって定義されたパラメータのセットの１つと等しい場合は、適当なREPEATエンコーディングが使用される。そうでなければ、アセンブラはRMOVを生成してREPEAT命令に続くユーザ定義パラメータをロードする。RMOV命令及びリマッピング・パラメータ・フォーマットの詳細については前記を参照。

ループの繰り返し（iteration）の回数が０であれば、REPEATの動作はUNPREDICATABLE（予想不可）である。

命令数領域が０にセットされると、REPEATの動作は、予想不可である。

ループに１つの命令しかなく、その命令がブランチであれば、予想不可能の挙動をする。
REPEATループの範囲からそのループの外へのブランチは、予想不可である。
飽和絶対命令は、ソース１の飽和絶対値（saturated absolute）を計算する。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	10011	F	S D	DEST	S1	R 1	SRC1	100000000000

動作：
dest＝SAT（（src1＞＝０）?src1:－src1）．値は常に飽和する。特に、0x80000000の絶対値は0x7fffffffであり、0x80000000ではない。
ニューモニック：
SABS＜dest＞，＜src1＞
フラグ
Ｚ　がセットされるのは、結果が０の時。
Ｎ　は保存される。
Ｃ　がセットされるのは、scr＜０（dest＝_scr1の場合）。
Ｖ　がセットされるのは、飽和が生じた時。
含む理由：
多くのDSPアプリケーションで役立つ。
選択（select）操作（条件付き移動）は、条件付きでソース１またはソース２を目的レジスタに移動させる。選択は、常に、移動と等価である。並列加算／減算の後で使用するための並列操作もある。
尚、両方のソースオペランドは、導入理由のための命令によっても読み出すことができるので、一方が空きであれば、そのオペランドが絶対的に必要であるかどうかに関係なく、命令は止まる。

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	011	OPC	F	S D	DEST	S1	R 1	SRC1	SRC2_SEL

OPCは命令のタイプを指定する。
動作（OPC）：

00 If＜cond＞holds for primary flags then dest=src1

（もし＜cond＞が一次フラグのためであれば）

else dest=src2.

01 If＜cond＞holds for the primary flags then dest.h=src1.h

（もし＜cond＞が一次フラグのためであれば）

else dest.h=src2.h,

If＜cond＞holds for the secondary flags then dest.l=src1.l

（もし＜cond＞が二次フラグのためであれば）

else dest.l=src2.l.

10 If＜cond＞holds for the primary flags then dest.h=src1.h

（もし＜cond＞が一次フラグのためであれば）

else dest.h=src2.h,

If＜cond＞fails for the secondary flags then dest.l=src1.l

（もし＜cond＞が二次フラグのためでなければ）

else dest.l=src2.l.

11 予約済

ニューモニック：

00 ＳＥＬ＜cond＞ ＜dest＞, ＜src1＞, ＜src2＞

01 ＳＥＬＴＴ＜cond＞ ＜dest＞, ＜src1＞, ＜src2＞

10 ＳＥＬＴＦ＜cond＞ ＜dest＞, ＜src1＞, ＜src2＞

11 未使用

レジスタが再充填の印になっていると、それは、無条件で再充填される。また、アセンブラ、次のニューモニックも提供する。

ＭＯＶ＜cond＞ ＜dest＞, ＜src1＞

ＳＥＬＦＴ＜cond＞ ＜dest＞, ＜src1＞, ＜src2＞

ＳＥＬＦＦ ＜cond＞＜dest＞, ＜src1＞, ＜src2＞

MOV＜cond＞A,Bは、SEL＜cond＞A,B,Aと等価である。SELFTとSELFFは、SELTF,SELTTを使用して、src1とsrc2を交換することによって得ることができる。

フラグ

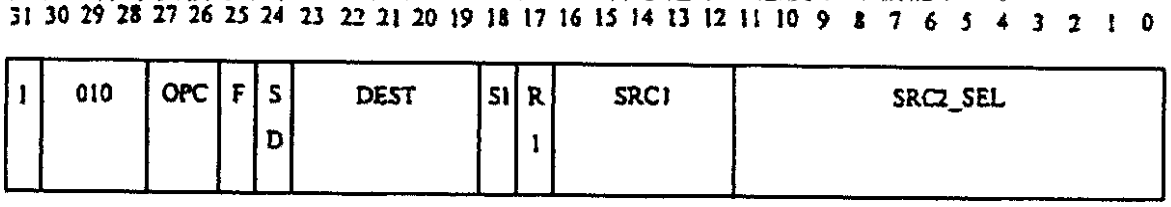
すべてのフラグは、一連の選択が行われるよう保存される。

含む理由：

簡単な決定をブランチに頼ることないインラインにするために使用される。最大要素のためにサンプル又はベクトルをスキャンする時に、そしてビタビ（Viterbi）アルゴリズムによって使用される。

シフト操作命令は、指定量の左右の論理シフト、右算術シフト、回転（rotate）を提供する。シフト量は、レジスタの内容の下８ビットから取られた－128から＋127の間の符号付

き整数、又は、＋１から＋31の範囲内の即値である。負の量のシフトは、ABS（シフト量）分反対方向にシフトさせる。
入力オペランドは、32ビットに符号拡張され、結果の32ビット出力は、書き戻し前に48ビットに符号拡張され、48ビットレジスタへの書き込みが感度よく機能する。



OPCは、命令のタイプを指定する。
動作（OPC）：

00　dest = (src2>=0) ? src1 <<src2 : src1 >>-src2

01　dest = (src2>=0) ? src1 >>src2 : src1 <<-src2

10　dest = (src2>=0) ? src1 - >>src2 : src1 <<-src2

11　dest = (src2>=0) ? src1 ROR src2 : src1 ROL -src2

ニューモニック：

00　ＡＳＬ　<dest>, <src1>, <src2_16>

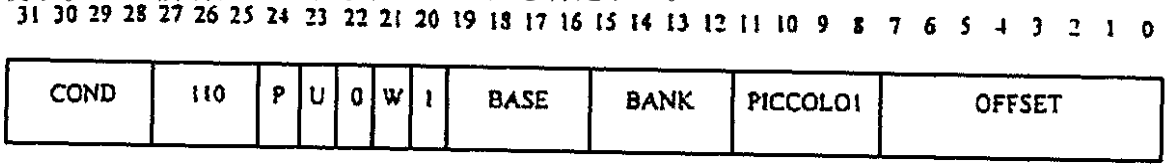
01　ＬＳＲ　<dest>, <src1>, <src2_16>

10　ＡＳＲ　<dest>, <src1>, <src2_16>

11　ＲＯＲ　<dest>, <src1>, <src2_16>

フラグ
Ｚ　がセットされるのは、結果が０の時。
Ｎ　がセットされるのは、結果が負の時。
Ｖ　は保存される。
Ｃ　は、最後にシフトされて出た（ARM上として）ビット値にセットされる。
レジスタ指定されたシフトの挙動は以下の通り。
－32によるLSLの結果は０で、src1のビット０にＣがセットされる。
－32を越えるものでのLSLは、結果が０で、Ｃは０にセットされる。
－32によるLSRの結果は０で、src1のビット31にＣがセットされる。
－32を越えるものでのLSRは、結果が０で、Ｃは０にセットされる。
－32以上でのASRの結果は充填され、Ｃはsrc1のビット31に等しい。
－32でのRORの結果はsrc1に等しく、Ｃがsrc1のビット31にセットされる。
－32を越えるｎによるRORは、ｎ－32によるRORと同じ結果とキャリーアウト（carry out）になるので、量が１から32の範囲内になるまで、繰り返し32をｎから引く。上記参照。
含む理由：
２のべき乗による乗算／除算。ビット及び領域抽出。シリアル・レジスタ。
未定義の命令が、上記命令セットリストで挙げてある。それらの実行により、ピッコロは、実行を停止し、状態レジスタにＵビットをセットし、それ自身をディスエーブルする（制御レジスタ内のＥビットがクリアされたかのように）。これにより、命令が将来拡張された場合も、それがトラップされて、オプショナルに、既存の手段上でエミュレートされることが可能である。
ARMからピッコロ状態へのアクセスは以下の通り。状態アクセス・モードを使用して、ピッコロの状態を観察／変更する。このメカニズムが提供されるのは次の２つの理由からである。

－文脈（Context）切替え
－デバッグ
ピッコロは、PSTATE命令を行うことで、状態アクセスモードになる。このモードでは、ピッコロの状態を退避して、一連のSTC及びLDC命令で復元される。状態アクセスモードに入ると、ピッコロ・コプロセッサID　PICCOLO1の使用が変更されて、ピッコロの状態にアクセスできるようになる。ピッコロの状態には７つのバンクがある。特定バンク内のすべてのデータは、単一のLDC又はSTCでロードし記憶することができる。
バンク0:プライベート・レジスタ
－　ピッコロIDレジスタ（Read Only）の値を含む１つの32ビットワード
－　制御レジスタの状態を含む１つの32ビットワード
－　状態レジスタの状態を含む１つの32ビットワード
－　プログラム・カウンタの状態を含む１つの32ビットワード
バンク1:汎用レジスタ（GPR）
－　汎用レジスタの状態を含む16個の32ビットワード
バンク2:アキュミュレータ
－　アキュミュレータ・レジスタの上の32ビットを含む４つの32ビットワード（注:GPR状態の複製が復元に必要だということは、さもないとレジスタバンク上で別の書き込みイネーブルを意味する）。
バンク3:レジスタ／ピッコロROB/出力FIFO状態
－　どのレジスタが再充填用の印（各32ビットレジスタにつき２ビット）になっているかを示す32ビットワードが１つ。
－　ROBタグ（ビット７から０に記憶されている７ビット項目８つ）の状態を含む32ビットワード８つ。
－　連合していない（unaligned）ROBラッチ（ビット17から０）の状態を含む32ビットワード３つ。
－　出力シフトレジスタ内のどのスロットが有効データを含むかを示す32ビットワードが１つ（ビット４は空きを示し、ビット３から０は、使用中のエントリの数をエンコードする）。
－　ラッチ（ビット17から０）を保持する出力FIFOの状態を含む32ビットワード１つ。
バンク4:ROB入力データ
－　32ビットデータ値が８つ。
バンク5:出力FIFOデータ
－　32ビットデータ値が８つ。
バンク6:ループハードウェア
－　ループ開始アドレスを含む32ビットワード４つ。
－　ループ最終アドレスを含む32ビットワード４つ。
－　ループ回数（ビット15から０）を含む32ビットワード４つ。
－　ユーザ定義リマッピング・パラメータその他のリマッピング状態を含む32ビットワードが１つ。
LDC命令は、ピッコロが状態アクセスモードにある時にピッコロの状態をロードするのに使う。BANK領域はロードされるバンクを特定する。



次の一連の動作により、ピッコロのすべての状態がレジスタR0内のアドレスからロードされる。

LDP B0, [R0], #16！；プライベート・レジスタ

LDP B1, [R0], #64！；汎用レジスタをロードする。

LDP B2, [R0], #16！；アキュミュレータをロードする。

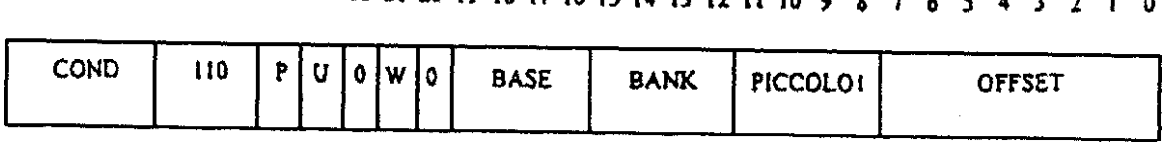
LDP B3, [R0], #56！；レジスタ／ROB／FIFO状態をロードする。

LDP B4, [R0], #32！；ROBデータをロードする。

LDP B5, [R0], #32！；出力FIFOをロードする。

LDP B6, [R0], #52！；ループハードウェアをロードする。

STC命令は、ピッコロが状態アクセスモードにある時にピッコロの状態を記憶させるのに使う。BANK領域はどのバンクが記憶されるかを特定する。



次の一連の動作により、ピッコロのすべての状態がレジスタR0内のアドレスから記憶される。

STP B0, [R0], #16！；プライベートレジスタを退避する。

STP B1, [R0], #64！；汎用レジスタを退避する。

STP B2, [R0], #16！；アキュミュレータを退避する。

STP B3, [R0], #56！；レジスタ／ROB／FIFO状態を退避する。

STP B4, [R0], #32！；ROBデータを退避する。

STP B5, [R0], #32！；出力FIFOを退避する。

STP B6, [R0], #52！；ループハードウェアを退避する。

デバッグ・モードーピッコロは、ARMによってサポートされているものと同じデバッグ・メカニズム、すなわち、DemonとAngelを介したソフトウェア、及び埋め込まれたICEを備えたハードウェア、に応答しなければならない。ピッコロのシステムをデバッグするためのいくつかのメカニズムがある。

－ARM命令ブレークポイント
－データ・ブレークポイント（ウオッチポイント）
－ピッコロ命令ブレークポイント
－ピッコロ・ソフトウェア・ブレークポイント

ARM命令ブレークポイント及びデータ・ブレークポイントは、ARM埋め込みICEモジュールによって扱われる。ピッコロ命令ブレークポイントは、ピッコロ埋め込みICEモジュールによって扱われる。ピッコロ・ソフトウェア・ブレークポイントは、ピッコロ・コアによって扱われる。ハードウェア・ブレークポイント・システムは、ARMとピッコロの両方がブレークポイントされるように構成される。

ソフトウェア・ブレークポイントを扱うのは、ピッコロ命令（Halt又はBreak）で、ピッ

コロに実行を止めさせ、デバッグ・モードに入れ（状態レジスタのＢビットがセットされる）、自身をディスエーブルする（ピッコロがPDISABLE命令によってディスエーブルされたようになる）。プログラム・カウンタは有効のままで、ブレークポイントのアドレスが回復できる。ピッコロは、それ以上、命令を実行しなくなる。
Single stepping Piccoloは、ピッコロ命令ストリーム上に次々にブレークポイントをセットすることによって行われる。
ソフトウェア・デバッグ－ピッコロによって提供される基本的機能は、状態アクセスモードにある時、コプロセッサ命令を介して、すべての状態をメモリーにロード及び退避させる能力である。これにより、デバッガーは、すべての状態をメモリーに退避させ、それを読み出し，及び／又は更新し、それをピッコロに復元することができる。ピッコロの記憶状態メカニズムは、非破壊的であり、つまり、ピッコロの状態を記憶する動作は、ピッコロの内部状態を駄目にすることはない。つまり、ピッコロは、その状態をダンプした後、それを復元することなしに、再開できる。
ピッコロ・キャッシュの状態を見つけるメカニズムを決定しなければならない。
ハードウェア・デバッグ－ハードウェア・デバッグは、ピッコロのコプロセッサ・インターフェース上のスキャン・チェインによって行うことができる。
ピッコロは状態アクセスモードになり、スキャン・チェインを介して、その状態を調査／変更してもらう。
ピッコロの状態レジスタは、ブレークポイント付き命令を実行したことを示す単一ビットを含む。ブレークポイント付き命令が実行されると、ピッコロは、状態レジスタにＢビットをセットし、実行を中止する。ピッコロに質問をするには、デバッガーは、ピッコロをイネーブルし、次のアクセスが起きる前に、制御レジスタに書き込むことによって、状態アクセスモードにしなければならない。
図４は、Hi/LoビットとSizeビットに応答して、選択されたレジスタの適当な半分をピッコロ・データパスに切り換えるマルチプレクサ構成を示す。Sizeビットが16ビットなら、符号拡張回路が必要に応じてデータパスの高次ビットに０か１を入れる。

【図７】

FIR（ブロック・フィルタ・アルゴリズム）

	d0	d1	d2	d3	d4	d5		
A0=	c0	c1	c2	c3	c4	c5		
A1=		c0	c1	c2	c3	c4	c5	
A2=			c0	c1	c2	c3	c4	c5
A3=				c0	c1	c2	c3	c4

Fig.7

【図１】

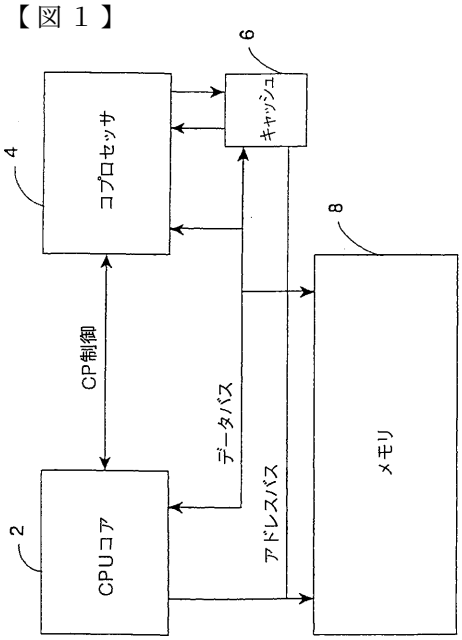


Fig.1

【図２】

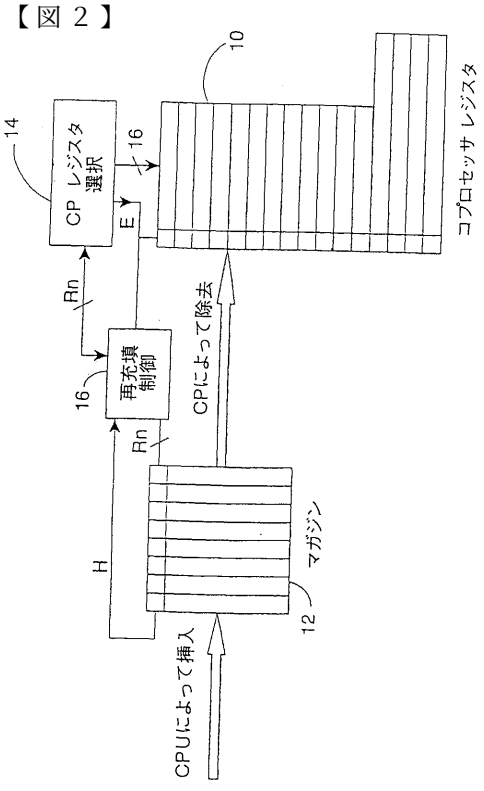


Fig.2

【図３】

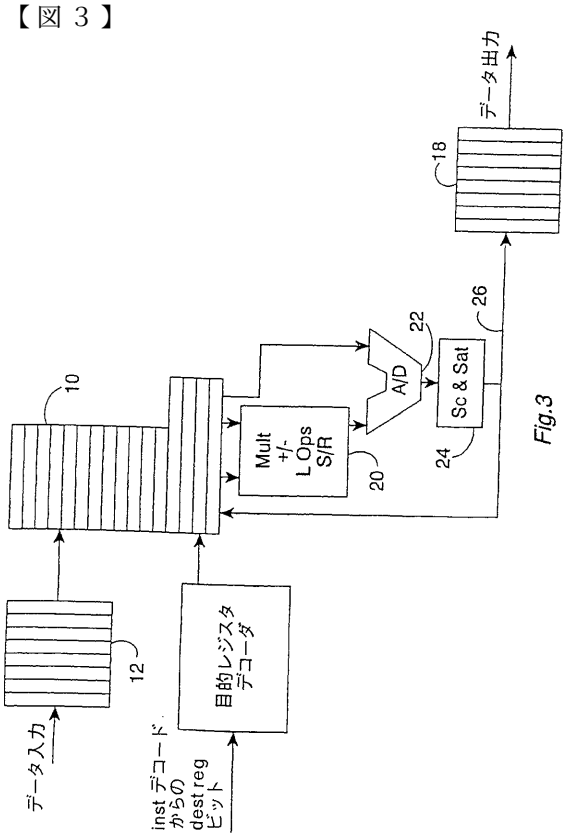


Fig.3

【図４】

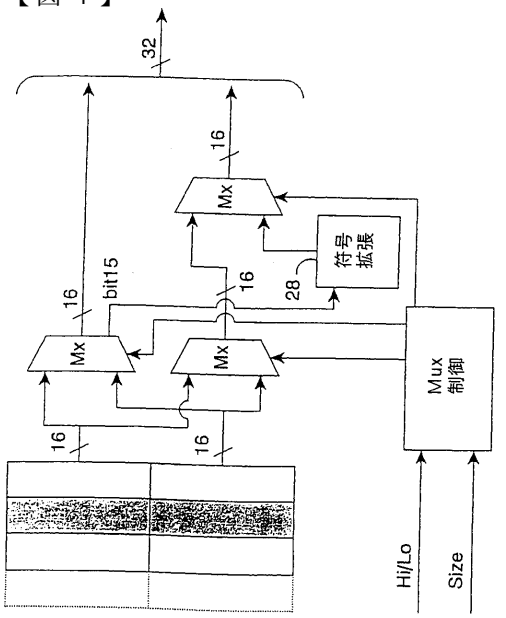


Fig.4

【図５】

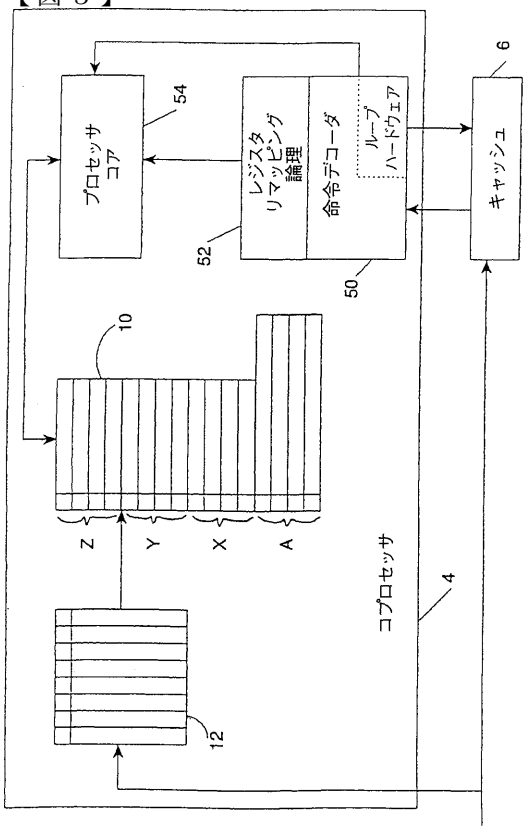


Fig.5

【図６】

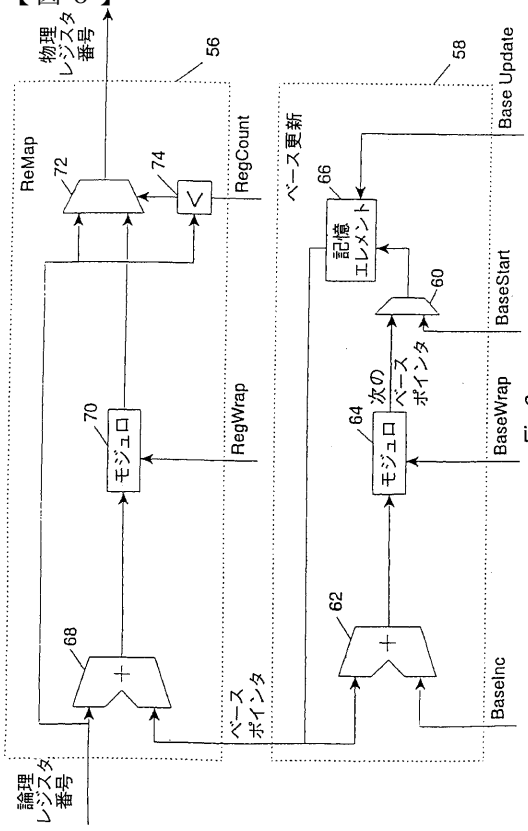


Fig.6

フロントページの続き

(72)発明者　ジャガー，デビッド，ビビアン
イギリス国　シービー１　４ティエヌ　ケンブリッジ，チェリー　ヒントン，マンドリル　クロース　４８
(72)発明者　グラス，サイモン，ジェームズ
イギリス国　シービー１　５ワイエイチ　ケンブリッジ，チェリー　ヒントン，コルツフット　クロース　２７

審査官　後藤　彰

(56)参考文献　特開平７－２９５８１１（ＪＰ，Ａ）
特開平７－２００２６０（ＪＰ，Ａ）
特開平４－２７８６３８（ＪＰ，Ａ）
特開平３－２６８０２４（ＪＰ，Ａ）

(58)調査した分野(Int.Cl.[7]，ＤＢ名)
G06F　9/30 -　9/38