Pegasus Prolog プロセッサにおける並列データパス操作の導入†

瀬尾和男†† 横田隆史††

Prolog を指向した RISC アーキテクチャを構築する上での問題点として，命令セットの低水準化に伴うコード量の増大と推論性能の低下が挙げられる．これらは主としてデータ型判定による処理分岐，特に Prolog 変数の取り扱いに関する条件分岐処理によってもたらされる．本論文では，これらの処理分岐を削減するとともにデータパス構成要素の利用効率を向上させる手法として，最新版の Pegasus プロセッサ (Pegasus-II) において導入された並列データパス操作について述べていく．まず，処理頻度の高い Prolog 操作実行時の各データパス動作に注目し，それらのリソース使用状況を考慮した並列化を検討した．その結果，リソース競合を起こさない命令の複合化に加え，並列に実行されているデータパス操作を動的なデータ型判定によって選択する「動的実行切り替え」手法の導入を行った．この手法を用いることにより，処理負荷の高いヘッド・ユニフィケーション操作を単一命令として実現することが可能となる．また，命令の複合化によって，変数セルの初期化や汎用ユニフィケーション・ルーチンへのエントリィ操作を高速化できる．これらの手法を実現した Pegasus-II チップを開発し，Warren ベンチマークを実行させることによって，ここで提案する方式の有効性を確かめた．

1. はじめに

D. H. D. Warren による Warren 抽象マシン (WAM)[1] の提案以来，多くの WAM に基づく Prolog マシンが研究開発されてきた[2),3)]．初期に開発されたマシンではマイクロプログラムによって WAM の各命令を実現するといった方式がほとんどであったが，その後 VLSI 設計／製造技術の急速な進歩を背景とし，Prolog 処理のより本質的な VLSI サポートを目指した研究が行われるようになった．Prolog を指向した RISC アーキテクチャの研究もこの流れの中に位置づけることができる．RISC アーキテクチャは，VLSI 時代におけるシステム構築のありかたについて新たな視点からの提案を行ったものであり，目的とする言語，システムに応じたリソース（シリコン領域）割り当てによる総合的な性能向上を目指している．性能向上のキーとなるのはパイプライン処理であり，これを乱す処理の分岐をいかに抑えるかが主要な課題となる．Prolog ではデータ型による処理の分岐が頻繁に起こるため，単純なタグ付きデータを扱う RISC 命令を用意しただけでは不十分である．特に，Prolog の変数については参照連鎖をたどるデリファレンス操作[4)]や，バックトラックにそなえ変数への値の代入履歴を保持するかどうかを判定するトレール操作[4)]といった特別な処理が必要であり，効率化のためにはなんらかの VLSI サポートが要求される．

われわれの研究所では，当初から，RISC による Prolog の効率的な実行を目指した Pegasus プロセッサ[5)~11)]の研究開発を行ってきた．Pegasus は，タグ付きデータに対する RISC 命令セットを基本に，Prolog 特有のユニフィケーションやバックトラックといった操作の高速化機構を組み込んだアーキテクチャを備えている．Prolog の変数に対するサポートとしては，参照連鎖を自分自身を繰り返し実行しながらたどっていくデリファレンス命令や，タグ／データの複合比較と条件分岐時の遅延スロットの無効化（スカッシング[12)]）を組み合わせてトレール処理を高速化する手法を導入している[10)]．

本論文では，これまでの一連のチップ／システムの開発と評価に基づき，さらに Prolog 処理を効率化する手法として，最新版 Pegasus プロセッサ (Pegasus-II) におけるデータパス操作の並列化について述べていく．Prolog において処理頻度の高い操作に着目し，実行時のデータパスの使用状態を解析するとともに，操作を並列化した場合の命令形式への影響について検討した．この結果，リソース競合を起こさない命令の複合化に加え，並列に実行されているデータパス操作を動的なデータ型判定によって選択する「動的実行切り替え」手法の導入が有効であることがわかった．この手法により，処理負荷の高いヘッド・ユニフィケーション操作を単一の命令として実現することが可能となる．また，命令の複合化によって，変数セルの初期

† Introduction of Parallel Datapath Operations for Pegasus Prolog Processor by KAZUO SEO and TAKASHI YOKOTA (Information & System Science Dept., Central Research Lab., Mitsubishi Electric Corp.).

†† 三菱電機(株)中央研究所システム基礎研究部

化や汎用ユニフィケーション・ルーチンへの分岐といった出現頻度の高い操作が高速化できる．これらのデータパス操作の並列化手法を実現した Pegasus-II チップを開発し，Warren ベンチマークを実行させることによって，ここで提案した方式の有効性を確かめた．

以下では，まず，2章と3章で，それぞれ，動的なデータ型判定に基づく実行切り替えと命令の複合化による処理効率の向上について検討する．これらを受けて4章では，パイプライン構成や命令セットの概要といった Pegasus-II の全体的なアーキテクチャについて述べる．5章で Warren のベンチマークに対する性能評価を今回導入した並列データパス操作を中心に行った後，6章でまとめを行う．

2.「動的実行切り替え」による高速化

前述のように WAM では，ユニフィケーション操作の効率化のために引数の型を特定した数種類の操作を用意している．それらは，Prolog の節に対して，ヘッド部のユニフィケーションを行う get 命令，ボディ部のユニフィケーションを行う put 命令，構造体を扱う場合にその各引数のユニフィケーションを行う unify 命令に大別される．ここでは基本となる get 命令の効率化によって全体としての性能向上を実現していく．

WAM で定義される get 命令には次のものがある[1]．

get_variable　get_value
get_constant　get_nil
get_list　get_structure

これらの中で get_variable と get_value はヘッド部の変数に対応し，前者がその初回出現時の処理を提供し，後者が2回目以降の出現時の処理を提供する．初回出現時の処理は単なる値の引渡しだけですみ，レジスタ転送命令またはストア命令で実現できる．これに対して2回目以降の処理では，すでにその変数になんらかの値が設定されているために，あらゆる組合せに対処できる汎用ユニフィケーション・ルーチンへの分岐が必要となる．この汎用ユニフィケーション処理の効率化については，3章の命令複合化の中で説明する．

これに対して，残りの get 命令はヘッド部の定数や構造体に対応し，引数のデータ型に応じてリード・モードの処理（変数の場合）またはライト・モードの処理（それ以外）が選択的に実行される．以下では，get 命令をプリミティヴな RISC 命令で実現した場合の処理分岐にかかわるオーバヘッドを説明した後，「動的実行切り替え」によるその効率化について述べていく．

2.1 get 命令と処理分岐

図 1 に，定数データに対する get 命令をタグ操作をそなえたプリミティヴな RISC 命令で記述した例を示す．図において，BTC と BTR はタグ部に対する，BVR はデータ部に対する条件比較命令，LDR と STR はそれぞれロード命令とストア命令，OVC は演算命令である．また，Rは引数レジスタ，CはRとユニファイされる定数を保持するレジスタである．H，HB，Bはいずれもトレール判定に用いられ，Hとの比較で変数のつくられた領域を判定し，さらにそれぞれの領域のバックトラック位置を保持する HB およびBとの比較によってトレール・スタックにその履歴を残すかどうかを判定する．最終行中の TR はトレール・スタックの先頭へのポインタを保持するレジスタである．また，Tref，Tvar はそれぞれ参照，変数を表すデータ・タグである．なお，分岐命令によるペナルティは1サイクルとしている．

get 命令では，まず引数データに対するデリファレンス操作が行われた後に，データ型判定が行われ，2つのモードへと分岐する．すなわち，引数が変数以外であれば，単純にタグを含めた一致比較が行われる．これに対して，引数が変数であれば，その変数に定数

```
        /* Dereference Operation */
label1: BTC.≠    R, Tref, label2
        BTC.=    R, Tref, label1
        LDR      R, R, 0
        /* Mode Check */
label2: BTC.=    R, Tvar, label3
        NOP
        /* Read Mode Operation */
        BTR.≠    R, C, fail
        NOP
        BVR.=    R, C, exit
        NOP
        JMP      fail
        NOP
        /* Write Mode Operation */
label3: BVR.>    R, H, label4
        STR      R, C, 0
        BVR.≥    R, HB, exit
        NOP
label4: BVR.≥    R, B, exit
        NOP
        STR      R, TR, 0
        OVC.add  TR, TR, 1
exit:   .....
```

図 1 get 命令のプリミティヴな RISC による記述
Fig. 1 Primitive RISC coding for "get instruction."

がユニファイされる．この時，バックトラック処理に備えてトレール判定が行われ，必要ならば変数のアドレスがトレール・スタックに格納される．get 命令の処理をオペランド・レジスタRに保持されるデータの型によって整理すると**図2**のようになる．

処理に要するサイクル数は，デリファレンス処理のループ回数をNとすれば，リード・モードでフェイルしない場合には $3N+8$ サイクル，ライト・モードではトレール処理に応じて $3N+9$ から $3N+12$ サイクル要することになる．通常，リード・モードでは $N=0$ であり8サイクル，ライト・モードでは $N=1$ であり12から15サイクルかかることになる．

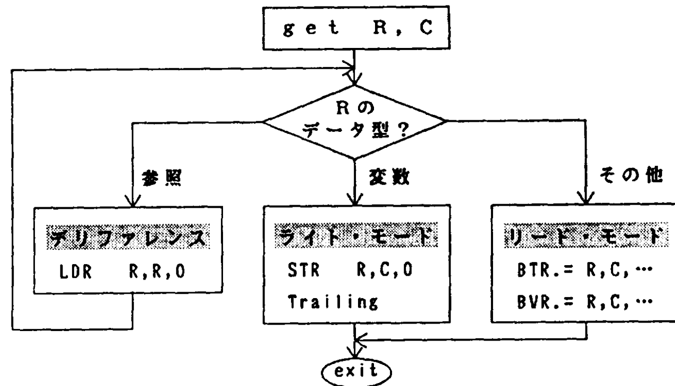


図2 データ型に応じた get 操作
Fig. 2 Get operations for each data type.

get 命令において処理負荷の高いデリファレンス操作やトレール操作を個別に効率化する方法については文献10) で示した．デリファレンス専用命令の導入や，タグ／データの複合比較とスカッシングを用いたトレール判定の高速化によって，get 命令は**図3**に示されるようにコーディングできる．図中，ラベル部に記された {*sq*} は分岐時に次命令を無効化するスカッシングを指定し，{*cc*} はタグ／データの同時比較を指定する．また，最後の STR 命令に付けられた '+1' は，ストア処理と並行してレジスタ TR をインクリメントするオプションである．処理に要するサイクル数は，デリファレンス処理のループ回数を N として，リード・モードでフェイルしない場合には $N+4$ サイクル，ライト・モードではトレール処理に応じて $N+6$ から $N+8$ サイクルと改善される．

```
          DRF     R
{sq}      BTC.=   R, Tvar, label1
{sq, cc}  BVR.=   R, C, exit
          JMP     fail
          NOP
label1:   STR     R, C, 0
{sq, cc}  BVR.>   R, HB, exit
{cc}      BVR.>   R, B, exit
          NOP
          STR.+1  R, TR, 0
exit:     .....
```

図3 Pegasus の前命令セットによる get 命令の記述
Fig. 3 Coding for "get instruction" by previous version of Pegasus instruction set.

2.2 「動的実行切り替え」による get 命令の実現

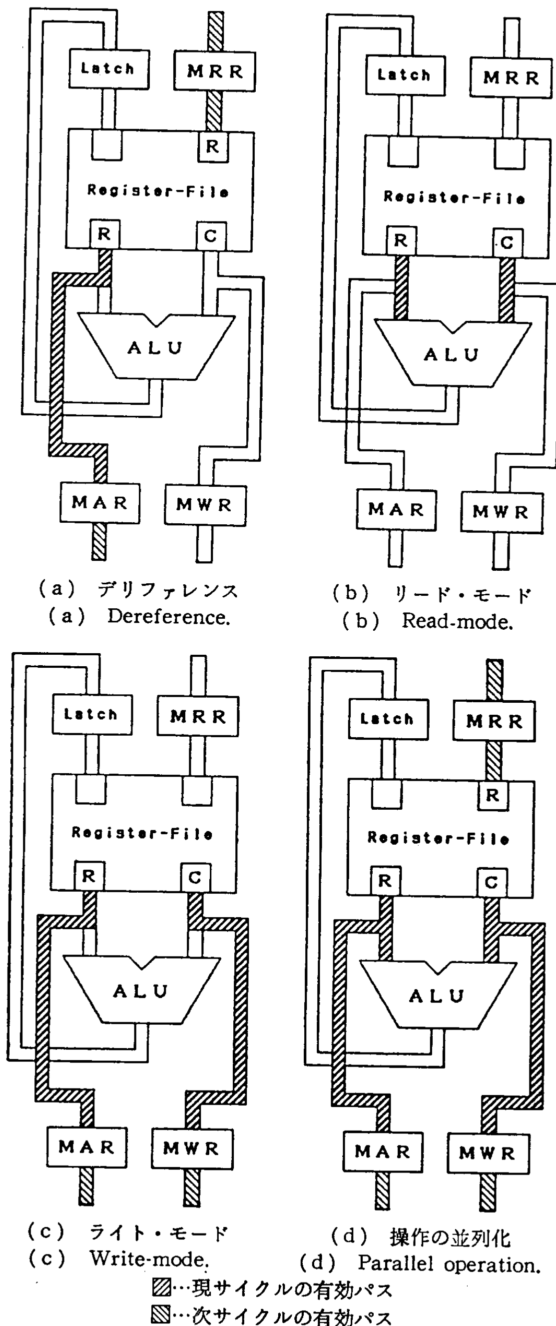
get 命令をさらに効率的に実現するために，操作実行時のデータパス状態を解析する．**図4**(a)〜(c)に，図2の各ブロックに対応する操作実行時のデータパス状態を示す．説明の都合上データパスは必要な部分だけに簡略化されて描かれている．図中，現在選択されているパスを斜線で示し，以降のサイクルで使われるパスを逆向きの斜線で示す．レジスタ・ファイルはリード／ライトとも2ポートをもつため，下側に読み出されるレジスタを表示し，上側に書き込まれるレジスタを表示する．

まず，図4(a)のデリファレンスの場合には，Rから読み出されたデータはメモリ・アドレス・レジスタ(MAR)にセットされ，次サイクルにメモリ読み込みが行われる．読み込まれたデータはその次のサイクルでレジスタ・ファイルに書き込まれるが，バイパス回路の導入により同時に参照可能となる．したがって，デリファレンス処理は1ループあたり2サイクル* かかり，参照以外の型のデータが読み込まれるまで繰り返される．次に図4(b)のリード・モードの処理では，RおよびCから読み出されたデータは ALU の2つのポートにセットされ，タグを含めた一致比較が実行される．判定結果によって次命令にいくか，フェイルするかが決定される．最後に図4(c)のライト・モードの処理では，2つのポートから読み出されたデータは，それぞれ，MAR およびメモリ・ライト・レジスタ (MWR) に送られ，メモリの書き込みが処理される．これに伴ってトレール判定が起動される．

ここで重要なことは，これらすべての操作においてレジスタ・ファイルからのデータの流れが共通であり，しかもリソース競合がないということである．こ

* Pegasus-II では，ハーバード・アーキテクチャの採用によって前の版の Pegasus チップに比べパイプラインが密になっている．このため，ロードしたデータを直後に参照する場合にはペナルティが1サイクル生じる．

れにより，図4(d)に示すようにこれらの操作をデータパス内で並列に実行することが可能となる．この場合，どの操作が選択されるかはRから読み出されたデータの型によって決まる．したがって，専用のタグ比較器をレジスタ・ファイルの出力パス上に置き，これらの操作と並行してデータ型を判定し，どの操作を有効化するかを決定すればよい．他の操作に比べタグ判定にかかる時間は短く，実現上の問題はない．また，オペランドが共通化されているため，3オペランドを基本とする命令形式上も問題なく表現できる．以降では，この命令をGET命令と呼び，WAMのget命令と区別する．



(a) デリファレンス
(a) Dereference.
(b) リード・モード
(b) Read-mode.
(c) ライト・モード
(c) Write-mode.
(d) 操作の並列化
(d) Parallel operation.
▨…現サイクルの有効パス
▧…次サイクルの有効パス
図 4 get 命令実行時のデータパス動作
Fig. 4 Datapath activities while executing "get instruction."

GET命令の実現上の問題となるのはライト・モードで起動されるトレール判定である．トレール判定をGET命令から独立させて実現した場合には，それを起動しないリード・モードではこの判定を条件分岐しなければならず，GET命令の効果が半減してしまう．したがって，Pegasus-IIでは図5に示されるトレール判定専用回路を導入し，これをライト・モードのSTR命令から起動するといった方法をとる．トレール判定回路によってトレール・スタックへの登録処理が必要と判定されると，付加的なSTR命令が実行されることになるが，必要となるオペランドについてはトレール判定を起動したSTR命令からとることができる．

このGET命令の実行サイクル数は，デリファレンスの繰り返し回数を N として，トレールを行わない場合には $2N+1$，行う場合には $2N+2$ と改善される．

2.3 構造体データに対するGET命令

WAMでは構造体はヒープ領域につくられ，それをポインタで参照する．構造体データに対するWAMのget命令は，ライト・モードでは変数にこのポインタを代入し，リード・モードではこのポインタをたどって構造体の名前および引数の個数を比較するところまでを行う．構造体の各引数に対する処理は後続の

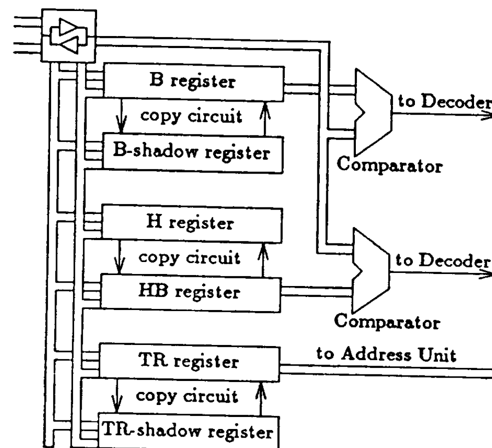


図 5 トレール判定回路
Fig. 5 Circuit for trail check.

unify 命令で行われることになり，その場合の処理モードは get 命令で決定されたモードが引き継がれる．

unify 命令はヒープ上の構造体を扱うため，get 命令の場合と異なり2つのモード間でデータパス操作を共通化できない．したがって，リードとライトの処理を独立したブロックに分割し，実装することになる．構造体に対する Pegasus-II の GET 命令は，第3のオペランドとして8ビット・オフセットをとり，モードが異なる場合には他方のブロックへと条件分岐する機能をもつ．

3. 命令の複合化による高速化

前章では，データパス操作を並列に行い，その中の1つを選択する手法について述べた．これに対して，本章で説明する命令の複合化は，同時に複数の命令を実行するものである．

複合命令の導入はリソースの競合および命令形式との整合性の2つの観点から検討されなければならない．すなわち，リソース競合のない命令どうしでも，それらがまったく異なるオペランドを参照しているような場合には，それらの複合命令を表現できないか，あるいは特別なデコード回路が必要となる．以下では，これらを考慮しながら今回導入した複合命令について説明する．

3.1 変数セルの初期化

Prolog では新たに変数セルが作成された場合，**図 6** に示されるように，メモリ上に変数タグをもち自分自身をポイントする変数セルがつくられるとともにそれへの参照ポインタを引数レジスタへ格納する．この場合，変数セルの作成は図7(a)のようにストア命令で，参照ポインタのレジスタへの格納は図7(b)のようにレジスタ転送命令で処理される．ここで重要なのは2つの命令でソース・レジスタが共通であることで，これによって複合化が可能となる．ただし，2つの命令で独立したタグの付け替えを必要とする．このため，付け替えるべきタグを命令のオペコード部で指定できるようになっている．複合化後のデータパスの状態を図7(c)に示す．

3.2 ヒープからの引数のロード

問い合わせの引数をヒープからレジスタにロードする処理において引数が変数の場合には，そのタグを参照に付け替えなければならない．通常なら条件判定を

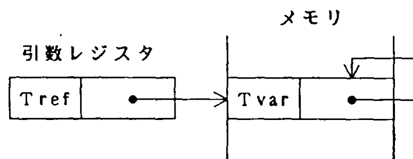
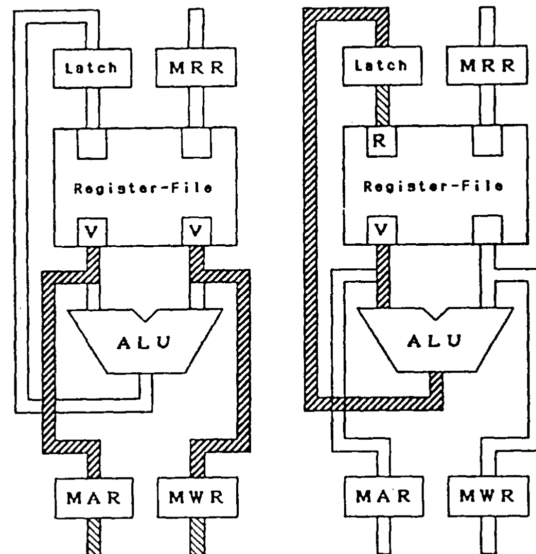
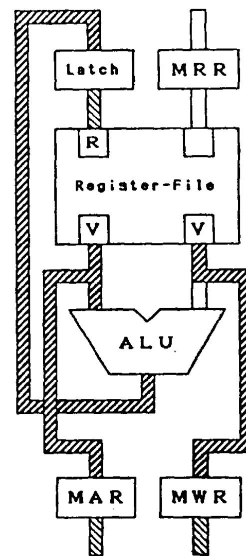


図 6 Prolog 変数と参照ポインタ
Fig. 6 Prolog variable and reference pointer.



(a) 変数の初期化
(a) Variable initialization.

(b) 変数への参照
(b) Reference to variables.



(c) 命令の複合化
(c) Compound instruction.
▨…現サイクルの有効パス
▧…次サイクルの有効パス

図 7 変数の初期化処理時のデータパス動作
Fig. 7 Datapath activities for variable initialization.

要するこの処理をマスク操作だけで実行可能なように，Pegasus では変数／参照の区別のために独立したタグ・ビットを割り当てている[10]．Pegasus-II では，このマスク操作を引数のロードと複合化することによってさらに高速化している．

3.3 汎用ユニフィケーション・ルーチンへの分岐

特殊な複合化として汎用ユニフィケーション・ルーチンへのエントリィ（分岐）命令が導入されている．これは，2つのオペランド・レジスタの基本データ型（変数，アトム，リスト，構造体）の組合せに応じて16方向の分岐を行うディスパッチング命令の各分岐先を汎用ユニフィケーション・ルーチン内の各処理のアドレスへと固定化するとともに，タグを含めた一致比較結果が等しい場合には分岐を無効化するという命令である．文献13) で報告されているように，汎用ユニフィケーション・ルーチンは，変数への代入やアトムどうしの比較といった単純な場合の処理が高速化されるように構成することが好ましい．この複合命令はそれら単純な場合の高速化を実現しており，非常に効果的である．

4. 並列データパス操作の実現

2章で説明したように，Pegasus-II の GET 命令は動的データ型判定による並列データパス操作の選択によって実現されている．本章では，パイプライン処理の中でそれをどのようにデコードし，実行していくかについて述べる．さらに，Pegasus-II の命令セットの概要とそれを実現するデータパス構成を説明する．

4.1 パイプライン構成と動的データ型判定

図 8 に Pegasus-II のパイプライン構成を示す．F ステージに命令フェッチが行われ，EX サイクルにそのデコードと実行が行われる．WM サイクルに実行結果のレジスタへの書き込みが，メモリへのアクセスと並行して処理される．メモリからのロードが実行さ

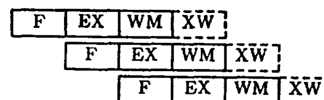


図 8 パイプライン・ステージ
Fig. 8 Pipeline stages.

表 1 Pegasus-II 命令セットの概要
Table 1 Summary of Pegasus-II instruction set.

Opcode	Function (*EA*=Effective Address)
[Prolog Instructions]	
UFY	Unify Two Registers
SW	Choice point creation, Main→Shadow Copy, Invoke Write-mode Access
SR	Choice point deletion, Shadow→Main Copy, Invoke Read-mode Access
GT	get_nil, get_constant, get_list, get_structure
JTD	Tag-dispatching Jump (Long Format), PC:=PC+dispatch(RegS1,RegS2)
STD	Tag-dispatching Jump (Short Format), PC:={PC+1,PC+Offs1,PC+Offs2,FailVector}
TSW	Switch on Tag-type, PC:={PC+1,*EA*,FailVector}
[Jump/Branch Instructions]	
BV/BT	Conditional Branch on Value/Tag-part, if TRUE then PC:=*EA*
FV/FT	Conditional Fail on Value/Tag-part, if FALSE then PC:=FailVector
JP	Unconditional Jump to any Address Space
HJP	Hashed Jump, PC:=PC+(Reg&Mask)
TRP	Software Trap, PC:=TrapVector+VectorNumber
[Data-Transfer Instructions]	
LD/ST	Load/Store a Word
LDX/STX	Move a Word from/to Coprocessor
M	Move Registers and Manipulate Tag/Value Part, RegD:=RegS
LA	Load Effective Address (Long Format), Reg:=*EA*
MAD	Load Effective Address (Short Format), Reg:=*EA*
[Logical/Arithmetic Operation Instructions]	
OV/OT	ALU Operation on Value/Tag, RegD:=RegS<op>RegS, optionally traps if non-integer.
MUL/DIV	Integer Multiplication/Division
[Compound Instructions]	
LRJ/SRJ	Load/Store and Jump
LDM/STM	Load/Store and Move
[Miscellaneous]	
NOP	No Operation
HLT	Halt
STI/CLI	Set/Clear Interrupt Mask
STM/CLM	Set/Clear GC-tag Mask
X	Set Special Purpose Registers/Vectors

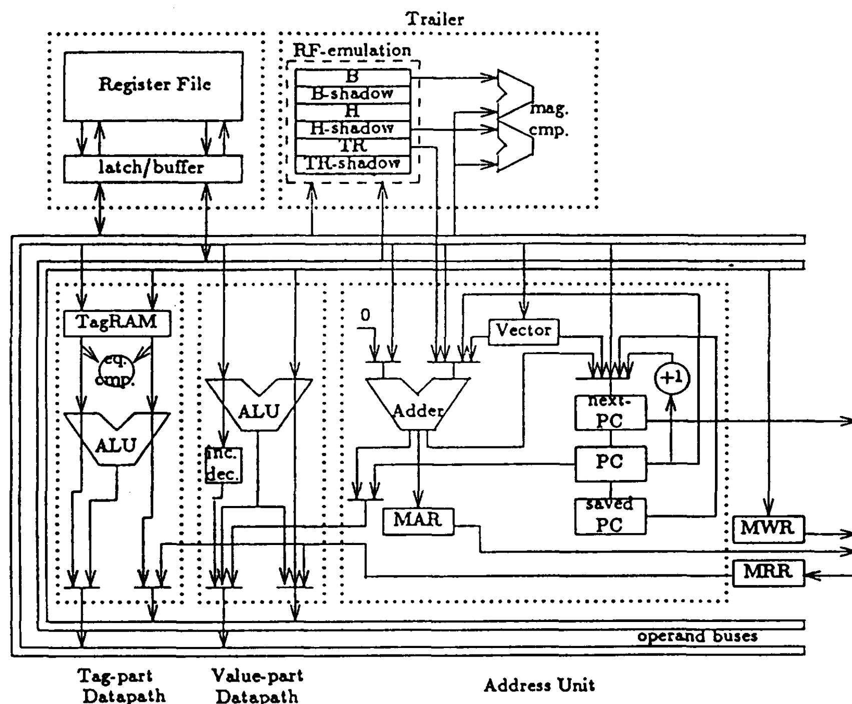


図 9 データパス構成
Fig. 9 Datapath construction.

れた場合には，読み込まれたデータをレジスタに格納するために，付加的に XW サイクルが実行される．

2章で述べた「動的実行切り替え」を行う命令のデコードは，EX サイクル中に2段階に分けられて実行される．すなわち，前半のサイクルでオペランド・レジスタが識別され，実行される可能性があるすべてのデータパス要素にデータが供給される．これらのデータは同時にタグ判定回路やトレール回路といった有効パスを選択する回路へと供給され，サイクルの後半に入る以前にすべてのデータパス上の有効パスが確定することになる．

4.2 命令セットとデータパス構成

表1に Pegasus-II の命令セットの概要を示す．命令セットは基本的には前の Pegasus チップと共通化されているが，2章で述べた GET 命令および3章の複合命令が新たに追加されている．このほか，WAM のインデキシング命令に対応する多方向分岐命令の種類が増えるとともに，整数の乗除算用の専用命令が追加された．

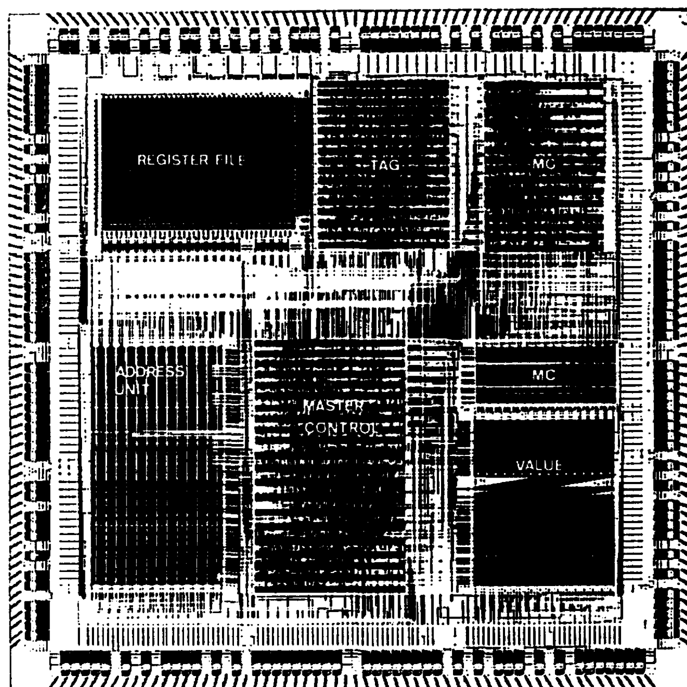


図 10 Pegasus-II チップ
Fig. 10 Pegasus-II chip.

図9に Pegasus-II のデータパス構成を，**図10**に開発されたプロトタイプ・チップを示す．今回行った

命令セットの拡張とハーバード・アーキテクチャの採用により，トランジスタ数は前チップの約 1.8 倍の 144,000 に増え，制御ロジックもかなり複雑化された．それでも，制御ロジックの割合は全体の 20% 以下となっている．これは，並列データパス操作の導入において，データパスの構成や命令形式への影響を十分に検討した結果であるといえる．設計期間の短縮のために高位のコンパイル・ツールを用いて設計したことにより，このチップの性能は最適化されてたものではなく，現在はサイクル時間 100 ns で動作させている．データパス内のクリティカルな構成要素をシンボリック・レイアウトなどを用いて高速化すれば，サイクル時間を半分以下にすることは十分に可能である．

5. 処理効率の改善

並列データパス操作の導入による処理効率の改善を評価するために，Pegasus-II チップを搭載したテスト・ボードを作製し，実行サイクル数などの評価がとれる実験環境を構築した．図 11 に Pegasus-II チップによって Warren ベンチマーク[14]を実行した結果 (P2) を，前の Pegasus チップ[10]による実行結果 (P1)，およびそのプリミティヴな命令だけで実行した結果 (P0) と対比して示す．この図では，2 章，3 章で導入した高速化手法の効果が明らかになるようにサイクル数で比較しているが，Pegasus-II ではハーバード・アーキテクチャの採用によりパイプライン構成が密になっているため，実行速度で比較すれば前チップに比べさらに 2 倍高速になる．

図 11 のグラフでは Warren ベンチマークの各プログラムに対する実行サイクル数の比較を示すとともに，それぞれの場合に GET 命令と複合命令によってどの程度のサイクル数の改善がなされたかを，Pegasus-II のサイクル数を基準として示す．いずれのベンチマーク・プログラムにおいても並列データパス操作の導入効果は大きく，かなりのサイクル数改善が達成されていることがわかる．ただし，query だけは乗除算が支配的であるため，乗除算命令の導入による改善が大きな比率を占めているが，その残りの部分については GET 命令が有効である．

ここで注意を要するのは，第 1 引数に対する GET 命令は，節のインデキシング[4]によってモード判定までが処理されるために，サイクル数改善への貢献がほとんどないことである．したがって，図 11 に示されたサイクル数の改善も，第 1 引数以外に対する GET 命令によるものである．GET 命令がその有効性を最も発揮できるのは，データベースへの応用などにおいて多くの引数をもつ事実節が扱われる場合である．

乗除算が支配的な query を除けば，前チップに対するサイクル数の改善 (P2/P1) の 50〜80% が並列データパス操作によって実現されている．それ以外の改善については，インデキシングの効率化のために多方向分岐命令の種類を増やしたことや，GET 命令以外でのデリファレンス操作やトレール操作の高速化によるものである．

図 12 に Warren ベンチマーク全体をコンパイルした時のコード量の比較を示す．コード量はプリミティヴな RISC に比べ 1/4，前のチップに比べ 1/2 になっている．マイクロプログラム方式をとる UCB の PLM では，その命令水準は WAM とほぼ同じである．文献 15)を参考に PLM の Warren ベンチマーク

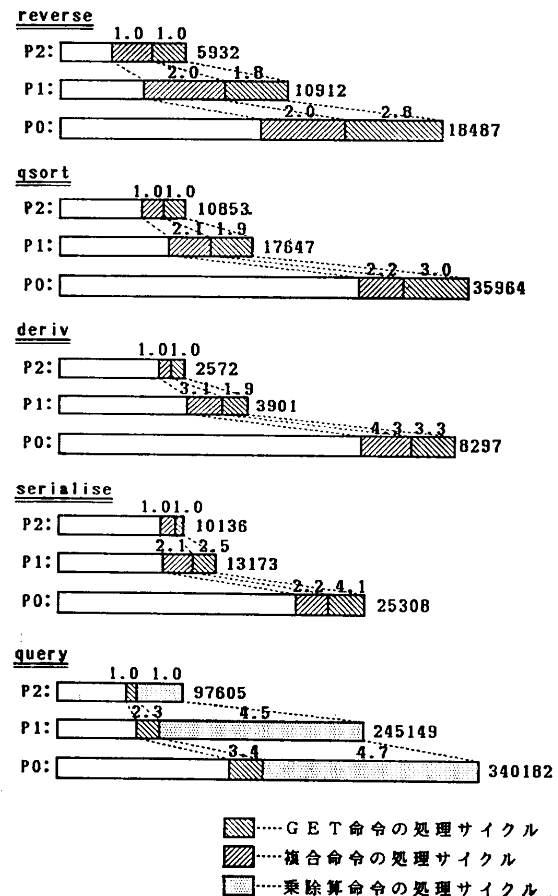


図 11　Warren ベンチマークに対する実行サイクル数の比較
Fig. 11　Comparison of execution cycles for Warren benchmarks.

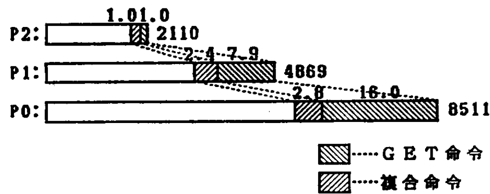


図 12 Warren ベンチマークに対するコード量の比較
Fig. 12 Comparison of code size for Warren benchmarks.

全体に対する命令数を計算すると約900命令になる．Pegasus-IIでは2,110命令であり，PLMの約2.35倍になっている．これは，文献15)で示されたPLMを他のRISCプロセッサと比較した場合の14倍という値に比べ大幅に改善されており，RISCでPrologを実行する場合の課題であるコード量の問題をほぼ解決することができたといえる．図12からも明らかなように，このコード量の改善にはGET命令の導入効果が大きい．

6. おわりに

Prologを指向したRISCアーキテクチャを構築する上で問題となる処理の分岐を削減するとともに，データパスの利用効率を向上させる手法として，Pegasus-IIで導入されている2種類の並列データパス操作について説明した．Warrenベンチマークを用いた評価からも明らかなように，その導入によって推論性能の向上とコード量の削減を同時に実現している．

これらによって，現在テスト・ボードで用いている100 nsのマシン・サイクルによる動作でも，Warrenベンチマークに対する推論性能は約300〜850 KLIPS (Kilo Logical Inferences Per Second)と高い．データパス内のクリティカルな構成要素をシンボリック・レイアウトなどを用いて高速化すれば，推論性能をさらに倍以上に上げることが可能である．また，コード量についてもマイクロプログラム方式のマシンに比べ約2.35倍と改善されている．

現在，VMEバスを介してUNIXマシンに接続されるボードやそれに対するソフトウェア環境の開発を進めており，今後はそれらを用いたシステムとしての評価を行っていく．

謝辞 本研究をまとめるにあたり，日頃ご指導いただいている三菱電機(株)中央研究所平山正治グループ・マネージャならびにご討論いただいたシステム基礎研究部の諸氏に感謝いたします．

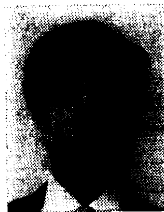
参考文献

1) Warren, D. H. D.: An Abstract Prolog Instruction Set, Technical Note 309, AI Center, SRI International (1983).
2) 金田悠起夫，松田秀雄：逐次型推論マシンのアーキテクチャ，情報処理，Vol. 32, No. 4, pp. 450-457 (1991).
3) 中島 浩：VLSI記号処理プロセッサ，情報処理，Vol. 31, No. 4, pp. 485-491 (1990).
4) 横田 実：論理型言語の逐次実行処理方式，情報処理，Vol. 32, No. 4, pp. 421-434 (1991).
5) 瀬尾和男，横田隆史：Prolog指向RISCプロセッサ "Pegasus"，情報処理学会アーキテクチャ研究会資料，63-5 (1986).
6) 瀬尾和男，横田隆史：Prolog指向RISCプロセッサ "Pegasus"—プロトタイプ開発とその評価—，情報処理学会アーキテクチャ研究会資料，69-11 (1988).
7) 瀬尾和男，横田隆史：Prolog指向RISCプロセッサPegasus—動的命令差替えによるProlog処理の効率化—，情報処理学会コンピュータアーキテクチャシンポジウム資料，pp. 73-80 (1988).
8) 横田隆史，瀬尾和男：Pegasus Prologプロセッサ—VMEbusボードによる評価—，情報処理学会アーキテクチャ研究会資料，77-11 (1989).
9) Seo, K. and Yokota, T.: PEGASUS: A RISC Processor for High-Performance Execution of Prolog Programs, *VLSI 87*, pp. 261-274, North-Holland (1987).
10) Seo, K. and Yokota, T.: Design and Fabrication of Pegasus Prolog Processor, *VLSI 89*, pp. 265-274, North-Holland (1989).
11) Yokota, T. and Seo, K.: Pegasus—An ASIC Implementation of High-Performance Prolog Processor, *Proc. EURO ASIC '90*, pp. 156-159 (1990).
12) Chow, P.: MIPS-X Instruction Set and Programmer's Manual, Technical Report No. CSL-86-289, CSL, Stanford University (1986).
13) Touati, H. and Despain, A.: An Empirical Study of the Warren Abstract Machine, *Proc. 4th SLP*, pp. 114-124 (1987).
14) Warren, D. H. D.: Applied Logic—Its Use and Implementation as Programming Tool, Technical Note 290, AI Center, SRI International (1983).
15) Borriello, G. et al.: RISCs vs. CISCs for Prolog: A Case Study, *Proc. 2nd ASPLOS*, pp. 136-145 (1987).

(平成3年4月25日受付)
(平成3年9月12日採録)

瀬尾　和男（正会員）

1956年生．1979年慶應義塾大学工学部電気工学科卒業．1981年同大学院修士課程修了．同年三菱電機(株)入社．以来，同社中央研究所において，計算機アーキテクチャ，特にVLSIを指向したアーキテクチャの研究に従事．1987年本学会研究賞，1989年IFIP VLSI '89 2nd Paper賞受賞．電子情報通信学会，ACM，IEEE各会員．



横田　隆史（正会員）

1960年生．1983年慶應義塾大学工学部電気工学科卒業．1985年同大学院修士課程修了．同年三菱電機(株)入社．現在，同中央研究所にて，推論マシンおよび並列処理アーキテクチャの研究開発に従事．1989年IFIP VLSI '89 2nd Paper賞受賞．電子情報通信学会会員．