Prologを指向したRISCプロセッサのパイプライン構成†

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主要な課題として，Prolog処理に適したパイプラインの設計が挙げられる．すなわち，データ型判定に基づく処理分岐のためのコントロール・ハザードや，スタックから取り出したデータを処理する場合に起こるデータ・ハザードの発生ができる限り少なくなるようなパイプライン構成をとることが必要となる．本論文では，Prologを指向したRISCプロセッサのためのパイプライン構成として，命令フェッチ，レジスタ読み出し，ALU操作／メモリ・アクセス，レジスタ書き込みの4つのステージからなり，分岐処理をレジスタ読み出しステージに組み込んだパイプライン構成を提案する．このパイプラインによってProlog処理が効率よく実行できることを示すために，まず，Prolog特有のデータ型に基づく処理分岐がパイプライン・ピッチに影響を与えることなく実現されることを示す．さらに，ロード／ストア命令のオフセット修飾なしでも，各スタックへのアクセスを行う処理を効率よくプログラム可能な方法を示す．これらを受けて，提案したパイプライン構成のWarrenベンチマークを用いた評価を行い，Prolog処理に対するその有効性を検証した．

1. はじめに

Prologは第五世代プロジェクトでの採用を契機として普及しつつある論理型プログラミング言語であり，人工知能応用，特に，自然言語処理やプロダクション・システムの構築などに適している．Prologにおける処理は，ユニフィケーションと呼ばれる述語呼出しに伴う引数間の同一化操作の繰返しと，それに失敗したときに起こるバックトラックと呼ばれる状態の復旧とに集約される．このようにPrologの処理自体は比較的画一化されたものではあるが，その実現には複数のスタックを対象とした非数値処理が必要となるため，汎用計算機では実行性能に限界があり，専用のアーキテクチャの構築が必要となる．

D. H. D. WarrenによるWarren抽象マシン(WAM)[1]の提案以来，初期に開発されたProlog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主流であった．これに対して，VLSI設計／製造技術の急速な進歩を背景とし，RISC方式によるProlog処理の効率化を目指した研究も多く行われるようになってきた[2),3)]．われわれの研究所でも，RISCによるPrologの効率的な実行を目指したPegasusプロセッサ[4)~12)]の研究開発を進めている．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．これまでに実施した数回のチップ／システム開発を通して，適切なVLSIサポートとコンパイラによる最適化によってRISC方式の欠点であるコード量の増加を抑えるとともに，高い推論性能を達成できることが実証できた．

RISCプロセッサにおいて効率向上の鍵を握るのはパイプライン処理である．すなわち，各パイプライン・ステージの処理負荷が均等化されるように，どのような並びでステージを配置していくかによって全体としての処理性能が大きく影響される．一般的に，パイプラインを乱す要因としては，分岐操作にかかわるコントロール・ハザードの発生や，メモリ・ロードおよびALUの各操作にかかわるデータ・ハザードの発生が考えられる[13)]．PrologではCなどの汎用言語に比べ，データ型による処理分岐が頻繁に起こるとともに，スタック処理などでメモリ・アクセスの頻度が高くなるといった特徴をもつ．したがって，これら2種類のパイプライン・ハザードの発生による遅延をできる限り抑えることが重要である．

本論文では，Prologを指向したRISCアーキテクチャを構築する上で主要な課題となるパイプライン構成について，データ型分岐およびメモリ・アクセスにかかわる遅延をどのように最小化していくかについて論じていく．まず，2章ではデータ型分岐による遅延を取り上げ，分岐操作をレジスタ読み出しステージ内に組み込むことを提案するとともに，その場合にPrologを指向した分岐命令をどのように構築すればパイプライン・ピッチへの影響がないかを述べる．次いで，3章では，ロード遅延の改善方法を示すとともに，

† Pipeline Structure for Prolog-oriented RISC Processors 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

それによってロード／ストア命令のオフセット修飾ができなくなることにどう対応していくかについて述べる．これらを受けて，4章では提案したパイプライン構成のWarrenベンチマークによる評価を行い，5章でまとめを述べる．

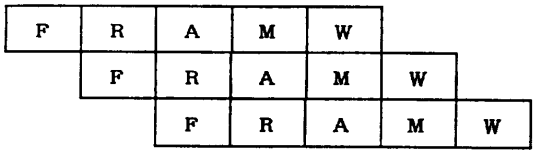
2. 分岐遅延の改善

図1に文献13)などで説明されているRISCプロセッサのための基本的な5ステージのパイプライン構成を示す．以下の2章では，このパイプラインを用いてPrologを指向したRISC命令セットを実行した場合の問題点を分析し，その改善方法を示していく．ただし，本論文では，Prologを指向したRISC命令セットとして，タグ付きデータに対するプリミティヴなRISC命令だけを考えるのではなく，Prologの実行に有効なものであれば高度な命令も導入することを前提としている．これは主として，Prologプロセッサにおいて推論性能とともに重要な性能指標であるコード量を考慮してのことである．本章では，まず，分岐遅延について考える．

2.1 分岐遅延の発生

分岐遅延は，分岐のための条件判定と後続命令のフェッチ操作とのずれによって生じる．Prologを指向したRISC命令セットの場合，手続き型言語を対象とするのに比べ，サポートすべき分岐処理が複雑になる．したがって，資源の有効利用を考慮し，分岐処理にALUを使用する構成が考えられる[5]．この場合には，図1のAステージ以降で分岐することになる．Aステージで分岐するとすれば，図2に示されるように，パイプライン中に命令実行が行われない遅延スロットが2つ空くことになり，分岐遅延は2となる☆．

通常，RISCプロセッサでは，遅延分岐手法[13]を用



F： 命令フェッチ
R： レジスタ読み出し
A： ALU操作
M： メモリ・アクセス
W： レジスタ書き込み

図1 汎用RISCプロセッサのパイプライン構成
Fig. 1 Pipeline structure for general purpose RISC processors.

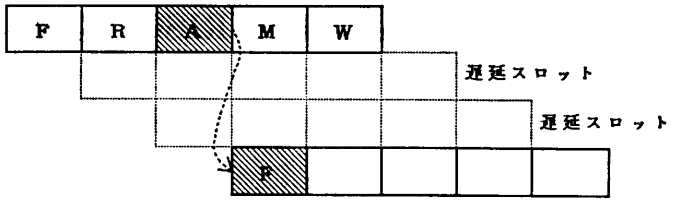
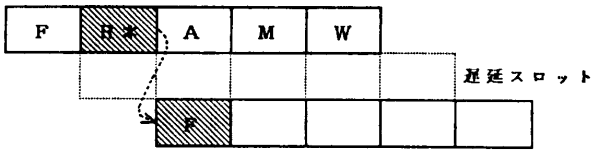


図2 分岐遅延
Fig. 2 Branch delays.

いてこれらの遅延スロットを分岐によって影響されない命令で埋めるといった最適化が行われる．しかしながら，Prologではデータ型による処理分岐の頻度が高く，これらの遅延スロットを埋める命令を見つけ出すのは非常に困難である．特に，PegasusのようにProlog専用の高度な命令が導入されている場合には，それらの命令を遅延スロットで実行することは複雑な例外制御を必要とすることになる．また，分岐／非分岐時に遅延スロット内の命令を無効化するスカッシング手法[14]の導入も有効であるが，分岐／非分岐の予測が困難な場合やPrologにおいて実行頻度が高いデリファレンス操作[15]のように短いループを形成する場合には対処しきれない．

これらにより，分岐判定はできる限り早いステージで行うことが望まれる．Prologでは，データ型の判定を始め，ほとんどの判定は等号比較であり，大小比較を必要とするのは組込み述語などに限定される．したがって，大小比較を減算と符号ビットの判定に分けて行うことにすれば，それ以外の比較についてはRステージ内で十分処理可能となる．すなわち，Rステージにおいてレジスタ読み出し操作と連続して条件判定を行い，分岐する場合には分岐アドレスをプログラム・カウンタにセットする．

図3に示されるように，分岐操作のRステージへの



R*： レジスタ読み出し＋分岐操作

図3 分岐遅延の改善
Fig. 3 Reduction of branch delays.

☆ 汎用のRISCの場合，大小比較などの負荷の大きい分岐処理に対して，条件コードを導入し，判定と分岐を分離することによって遅延スロット数を減らすといったことが行われている．Prologを指向した分岐では，大小比較に比べ比較と分岐が一体化しているために，比較結果を複数の分岐命令で共有できる場合や，比較命令で他の分岐命令の遅延スロットを埋めることができる場合が少ない．したがって，判定と分岐を1命令で実現した場合に比べ，あまり有効とはいえない．

移行により，分岐による遅延は1と改善される．分岐操作を持ち込んだことによってどの程度Rステージの負荷が増えるかについては，その分岐判定処理の内容に応じて変わってくる．レジスタ・ファイルの読み出しは比較的高速に行えるため，単純な等号比較だけならば，ALU 操作やメモリ・アクセスとの兼ね合いで全体のパイプライン・ピッチに影響を与えることはない．問題は Prolog の高速化に必要となる多方向分岐などをどう実現するかである．以下では，Pegasus プロセッサにおけるデータ型による処理分岐の実現を例として，この問題について述べていく．

2.2 Pegasus プロセッサにおける処理分岐の実現

WAM では，ユニフィケーションの効率化のために，引数のデータ型に応じた各種の操作命令が用意されている[1]．これらの命令の中でインデキシング命令やヘッド・ユニフィケーション命令などを効率よく実現するには，データ型による多方向分岐が必要となる．最も単純な方法としては，ユニフィケーションされる2つのオペランドのデータ型を表すタグ部の組合せで命令に続いて格納されている表を引き，そこに置かれたディスパッチ用アドレスに分岐するタグ・ディスパッチ命令を用意すればよい[6]．しかしながら，この方法では，次章で述べるメモリ・ロードによる遅延を引き起こし，処理効率を低下させるとともに，命令ごとにディスパッチ表をかかえるためにコード量が多くなるといった欠点がある．

最新版の Pegasus プロセッサ[4]では，間接分岐による性能低下を避けるために，命令内の分岐用オフセットにより多方向に分岐する STD (Short Tag Dispatching) 命令や，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行う UFY (UinFY) 命令が導入されている．さらに，分岐する代わりに，並列にデータパス操作を実行し，同時に行われるデータ型判定によってそれらのうちの1つを有効化するといった「動的実行切り替え」を実現した GET 命令が導入されている．

以下では，本章で提案したRステージ内で分岐操作を行うパイプラインによって，Prolog を指向したこれらの命令をパイプライン・ピッチに影響を与えないように実現する方法を示していく．

2.2.1 STD 命令の実現

WAM で定義された節のインデキシング処理[1]は，第1引数の基本データ型（変数，アトム，リスト，構造体）によって行われる．したがって，その実現には分岐しない型を除いて他の3つの型に対する分岐オフセットが必要となる．しかしながら，4つの型すべてとユニファイ可能な場合は少なく，また，その場合には命令内の小さい分岐オフセットでは扱いきれない可能性が高い．したがって，Pegasus の STD 命令では分岐オフセットを2つだけ用意し，残りの1つのデータ型については特殊アドレスを保持するアドレス・ベクタ内のフェイル・アドレスに分岐するように実現されている．

図 4 に STD 命令の分岐操作を示す．この分岐操作をRステージ内で実現するために，2つのオフセット計算専用の加算器の導入が必要となる．Rステージ内での処理としては，レジスタ・ファイルからのオペランド読み出しと並行して，2つの分岐オフセットのプログラム・カウンタ出力との加算，アドレス・ベクタからのフェイル・アドレスの読み出しが行われる．レジスタ・ファイルからのオペランドの読み出しが完了すると，タグ・デコード回路によって型判定が行われ，分岐アドレスが選択される．

この分岐操作のクリティカル・パスは，分岐オフセット用の加算器を通る経路である．Pegasus では，データ語長は32ビットなのに対して，アドレス空間は26ビットで表現されている．Aステージにおいてデータ語長の符号付き加減算が行われることを考慮すれば，それよりも演算ビット長が短く，加算専用の回路を使用する分岐オフセットの計算をマルチプレクサを通過する遅延を含めてAステージでの処理時間以内に実現することは可能である．こういった実現方法を採るならば，STD 命令の分岐操作がパイプライン・ピッチに影響することはない．

2.2.2 UFY 命令の実現

UFY 命令は，2つのオペランド・レジスタの基本

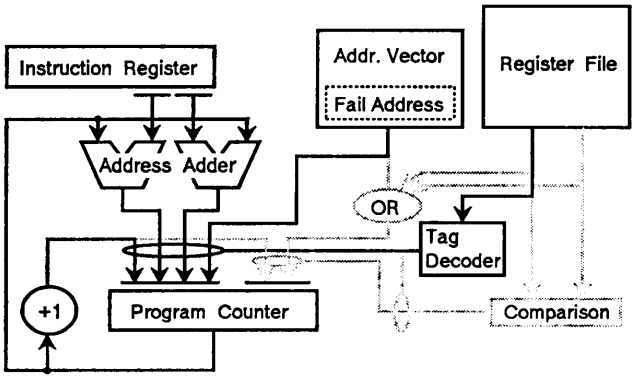
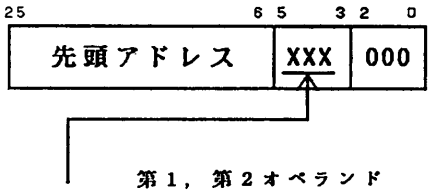


図 4 STD 命令の分岐操作
Fig. 4 Branch operation for STD instruction.

データ型の組合せに応じて，汎用ユニフィケーション・ルーチン内の各処理アドレスへ分岐する多方向分岐命令である．さらに，この分岐操作と並行して行われる2つのオペランドに対するタグを含めた一致比較の結果が等しい場合には，分岐を無効化する．文献16)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．この命令はそれらの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．

この命令の実現において，各分岐先アドレスをアドレス・ベクタ内に格納しておく方法も考えられるが，それではレジスタ読み出しを行った結果によってアドレス・ベクタを検索することになり，Rステージがかなり延びる可能性がある．これを回避する方法として，Pegasus では汎用ユニフィケーション・ルーチンの各エントリィを8語ごとに配置し，図5に示すように分岐アドレスを簡単なビット操作で求められるよう



第1，第2オペランド
000 … 変数，変数
001 … 変数，非変数
010 … 非変数，変数
011 … アトム，アトム
100 … リスト，リスト
101 … 構造体，構造体
111 … その他の組合せ

図5 UFY 命令の分岐アドレス計算
Fig. 5 Branch address calculation for UFY instruction.

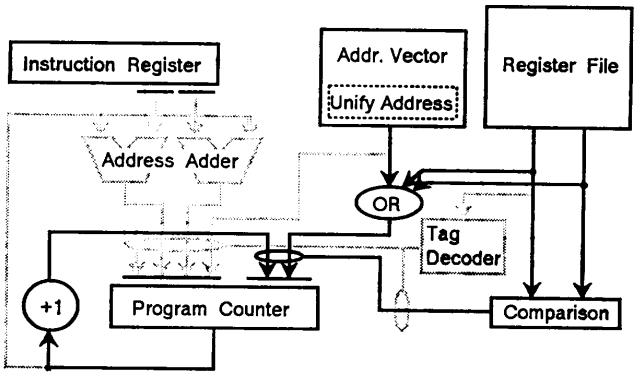


図6 UFY 命令の分岐操作
Fig. 6 Branch operation for UFY instruction.

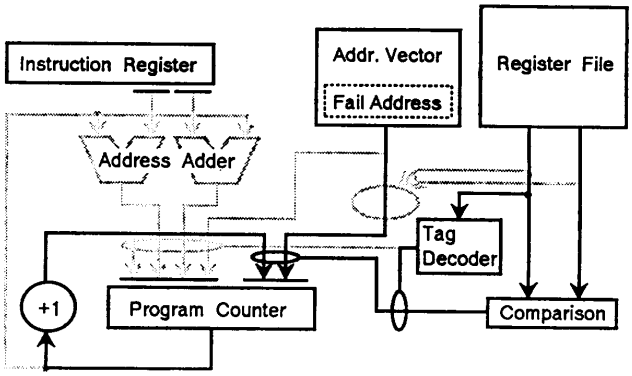


図7 GET 命令の分岐操作
Fig. 7 Branch operation for GET instruction.

にしている．

図6にこの命令の分岐操作を示す．分岐アドレスの計算をビット操作で行うことにより，この分岐操作のクリティカル・パスは，2つのオペランドがまったく同一の場合に分岐を無効化する経路となる．したがって，通常のタグ／データ部に対する分岐操作と同様に，一致判定結果によってプログラム・カウンタの更新を制御するだけの負荷で処理可能であり，この命令の分岐操作がパイプライン・ピッチに影響することはない．

2.2.3 GET 命令の実現

ヘッド・ユニフィケーション処理では，オペランドのデータ型に応じて，ロード(デリファレンス)，ストア（ライト・モード)，一致比較（リード・モード）の各操作が選択される[4]．これらの操作は，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ことから，すべての操作をデータパス内で並列に実行し，データ型が判定された時点でそれらのうちの1つを選択するといった「動的実行切り替え」が可能となる．

定数に対するヘッド・ユニフィケーションを実行する GET 命令のRステージでは，レジスタ・ファイルからのオペランドと定数の読み出しが行われた後，それらがロード／ストアの経路および一致比較回路へと供給され，さらにオペランドについてはデータ型判定が実行される．これらの操作におけるクリティカル・パスは，図7に示された一致比較の結果をデータ型判定で無効化する経路であるが，これは単に最終段のマルチプレクサの制御線上に論理ゲートをもう1段追加するだけで実現できる．したがって，この命令の「動的実行切り替え」操作がパイプライン・ピッチに影響することはない．

以上のように，分岐判定や分岐オフセットの計算ができる限り並列に実行されるように構成することによって，Prolog を指向した分岐操作をパイプライン・ピッチへの影響なしにRステージ内で効率よく実現できる．

3. ロード遅延の改善

本章では，Prolog を指向した RISC プロセッサのパイプラインにおけるロード遅延の改善について述べていく．

3.1 ロード遅延の発生

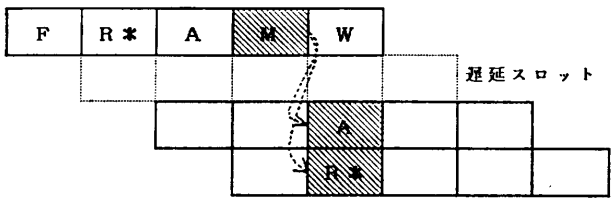
ロードによる遅延は，ロードしてきたデータを直後の命令で参照するような場合に生じる．ロードしたデータを直接演算オペランド・バスに引き渡すバイパス回路が導入されているとしても，MステージとAステージのずれによって遅延スロットが1つできてしまう．さらに，前節で提案したRステージで分岐操作を行うパイプラインでは，その判定に用いられるオペランドをロードするような場合には遅延スロットは2となる（図8参照）．したがって，メモリ・アクセスはできるだけ早いステージで実行することが望まれる．

図1のパイプラインにおいて，MステージがAステージの後にきているのは，主としてメモリへのアクセスにおいてオフセット修飾をサポートするためである．以下では，このオフセット修飾の要否という観点から WAM で定義されるメモリ・アクセスについて述べていく．

3.2 WAM におけるメモリ・アクセス

WAMでは，プログラム領域に加えて，構造体やグローバル変数を格納するヒープ領域，制御フレームを格納するローカル・スタック領域，変数への値のバインド履歴を保持するトレール・スタック領域，構造体のユニフィケーションに用いられるプッシュダウン・リスト領域の4つのデータ領域が定義されている．

これらのデータ領域へのアクセス形態を調べれば，



R＊： レジスタ読み出し＋分岐操作

図 8 ロード遅延
Fig. 8 Load delays.

オフセット修飾が必要となるのは，

・ローカル・スタック上の制御フレームへのアクセス

・他のスタック領域でのポップアップ操作

だけに限定されることがわかる．すなわち，スタック・ポインタは常に次にデータが積まれる位置を指しているため，プッシュダウン操作はポスト・インクリメントで処理できるが，ポップアップ操作はプリ・デクリメントでなければ処理できない．したがって，オフセット修飾付きのメモリ・アクセスが必要となる．ただし，ヒープ領域については，バックトラックまたはガベージ・コレクションが起こった時のみ解放されるため，ポップアップ操作は行われない．このようにオフセット修飾付きメモリ・アクセスが限定されている理由としては，WAM においてメモリ・アクセスのため各種レジスタが定義されていることが考えられる．

以下では，これらの各場合について考察し，メモリ・アクセスのオフセット修飾なしでそれらを効率よくプログラムする方法を示していく．

3.2.1 制御フレームへのアクセス

ローカル・スタック上には，選択点フレームと環境フレームと呼ばれる2種類の制御フレームが格納される[15]．このうち，バックトラック用の状態を保持する選択点フレームについては，フレーム内を順次アクセスするためオフセット修飾付きメモリ・アクセスは必要でない．これに対して，環境フレームは複数の節呼出しにかかわる変数を退避しておくためのもので，順次アクセスされるとは決まっていないために，そのアクセスにはオフセット修飾が必要となる．

オフセット修飾をなくすことによる実行命令数の増加を抑えるために，次のような最適化を行うことができる．すなわち，ある節呼出しから次の節呼出しの間についてはレジスタの内容がこわされることはないので，環境フレームのアクセス用に専用のレジスタを割り当て，最初の一度だけオフセットの計算を独立した演算命令で行えば，それ以降については1つ前のアクセス時にアクセス用のレジスタをポスト修飾することによって処理できる．したがって，ペナルティ（実行命令数の増加）は高々節呼出し回数分だけとなる．

3.2.2 トレール・スタックのポップアップ操作

トレール・スタックがポップアップされるのはバックトラック処理時である．すなわち，トレール・ポインタが1つ前の状態を示す位置にくるまでポップアッ

プが繰り返され，そのつど取り出された変数への値の代入を無効化していく．この処理において，まずトレール・ポインタを戻すべき位置を一時レジスタにコピーし，そこから現在の位置まで逆向きに無効化を行っても支障はない．こうすることによって，メモリ・アクセスのオフセット修飾を使わずに，トレール・スタックによる状態復旧を効率よく処理できる．この場合，オフセット修飾がないことによるペナルティは，トレール・ポインタの前の位置を一時レジスタにコピーする1命令だけですむ．

3.2.3 プッシュダウン・リストのポップアップ操作

プッシュダウン・リストは，汎用ユニフィケーション・ルーチン内において構造体どうしをユニフィケーションする場合に用いられる．したがって，複雑な構造体どうしがユニフィケーションされる場合などには，ポップアップされる回数も多くなる．しかしながら，その場合ポップアップされたデータは直後にデリファレンスされることになり，それによって引き起こされるロード遅延のペナルティを考えるとオフセット修飾による効果は打ち消される．むしろ，汎用ユニフィケーション・ルーチン全体でみた場合には，ロード遅延を改善することによる効率向上の方がはるかに大きい．

以上により，AステージとMステージを，どちらかの操作が選択的に実行されるステージへと統合したパイプライン形態を採ることが考えられる．この場合，オフセットの加減操作を要するメモリ・アクセスは2命令で実現するようになるが，ここで述べてきたようにPrologではその頻度は高くない．**図9**に，本論文で提案するパイプラインの最終構成を示す．AステージとMステージの統合により，ロードによる遅延が改善されている．

また，**図10**に，図9のパイプラインの2層クロックに対する実現例を示す．2章で説明した各分岐操作は，この実現例ではRステージの前半で実行されることになる．また，メモリ・アクセスの負荷を考慮し，命令／データともにそのアクセスに1.5サイクル費やすことが可能な構成となっている．

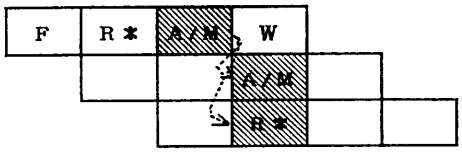


図9 最終的なパイプライン構成とロード遅延の改善
Fig. 9 Proposed pipeline and reduction of load delays.

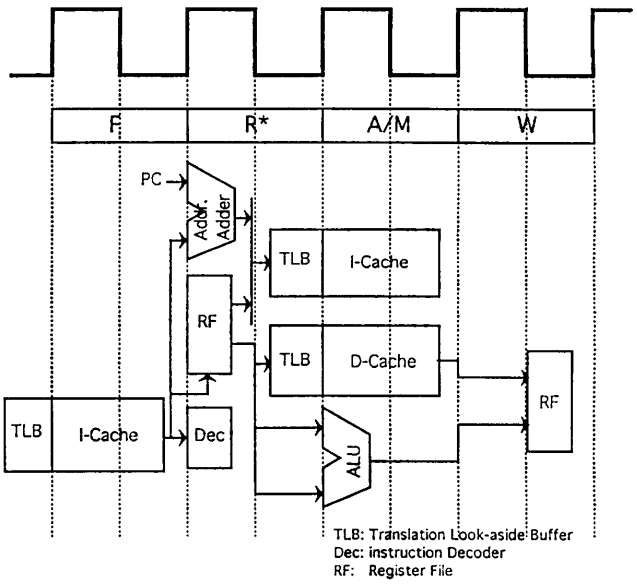


図10 提案したパイプライン構成の2層クロックに対する実現例
Fig. 10 2-Phase clocking implementation of proposed pipeline structure.

4. Warren ベンチマークによる性能評価

2章，3章で提案した4ステージのパイプライン構成について Warren ベンチマーク[17]を用いた評価を行った．この評価では，パイプライン上で実行される命令セットとして Pegasus プロセッサ[4),5)]の命令セットを使用し，図1のパイプラインで実行した場合と比較した．評価のデータは，Pegasus プロセッサによる実行結果を，比較する2つのパイプライン構成を考慮して解析することにより求めた．

評価項目は以下のとおりである．

・分岐遅延の改善
・ロード遅延の改善

結果を**表1**にまとめる．改善の単位はサイクルであり，全実行サイクルに対する改善率もあわせて示した．また，ロード遅延の改善については，減少したサイクル数からメモリ・アクセスのオフセット計算ができなくなることによるサイクル数の増加分を差し引く必要があり，その計算を**表2**にまとめる．この表2では，オフセット計算による実行サイクル数の増加について，3章での考察にあわせてそれらがどこで発生したか（環境フレームへのアクセス，フェイル・ルーチ

表 1 Warren ベンチマークによる提案したパイプライン構成の評価
Table 1 Evaluation of proposed pipeline structure using Warren benchmarks.

評価項目 / ベンチマーク	総実行サイクル数	分岐遅延の改善サイクル数 (改善率)	ロード遅延の改善サイクル数 (改善率)
nreverse	6,052	991 (16.4%)	436 (7.2%)
qsort	11,675	977 (8.4%)	204 (1.7%)
deriv			
time 10	793	66 (8.3%)	29 (3.7%)
divide 10	910	66 (7.3%)	29 (3.2%)
log 10	419	53 (12.6%)	32 (7.6%)
ops 8	569	47 (8.3%)	19 (3.3%)
serialise	10,671	975 (9.1%)	487 (4.6%)
dbquery	101,787	3,901 (3.8%)	5,699 (5.6%)

表 2 ロード遅延の改善に関する評価
Table 2 Evaluation on load delay reduction.

評価項目 / ベンチマーク	ロード遅延の減少 A	オフセット計算による実行サイクル数増加合計B (フレーム・アクセス, フェイル, ユニファイ)	差引の改善 A−B
nreverse	496	60 (60, —, —)	436
qsort	701	497 (375, 122, —)	204
deriv			
time 10	56	27 (27, —, —)	29
divide 10	56	27 (27, —, —)	29
log 10	52	20 (20, —, —)	32
ops 8	41	22 (19, 3, —)	19
serialise	856	369 (267, 86, 16)	487
dbquery	8,301	2,602 (1,977, 625, —)	5,699

(単位はサイクル)

ン，汎用ユニフィケーション・ルーチン）も示した．

まず，分岐遅延については，主にインデキシングの分岐，ユニフィケーションのモード分岐，変数の種別に関する分岐などによるものであり，分岐／非分岐時のスカッシング（遅延スロット内の命令の無効化）による最適化も効かなかった場合である．すべてのベンチマークで分岐遅延が改善されることがわかる．したがって，2章で説明したように，オフセット計算のための専用回路を導入したり，分岐先アドレスを固定するなどのコストをかけてRステージ内で分岐操作を実現するならば，Prolog の実行に関しては提案したパイプライン構成が効率的であることが理解できる．

この評価では，Pegasus の「動的実行切り替え」を行うような複雑な命令でも遅延スロット内で実行可能であるとして，遅延分岐による最適化の対象とした．これによって，遅延スロットが2つの場合でも，それらを埋めることができる確率が高くなっている．しかしながら，例外制御などの実装を考えると，遅延スロットが2つになった場合にこれらの複雑な命令の実行を許すことは困難である．したがって，実際には，分岐遅延の改善はさらに大きくなるものと予想される．

次にロード遅延についても，多少の差はあるがすべてのベンチマークにおいて改善がみられる．これに対して，オフセット修飾付きのメモリ・アクセスがなくなることによる実行命令数の増加も無視できないが，全体としてみた場合にはロード遅延の改善効果のほうが大きい．これには，3章で説明したオフセット修飾がない場合の効率的なプログラム方法が大きく寄与し

ている.

以上により，本論文で提案したパイプライン構成のProlog 処理に対する有効性が確かめられた.

5. おわりに

本論文では，Prolog 処理を指向した RISC プロセッサのパイプライン構成について述べた．データ型判定による分岐遅延の軽減を目的とした分岐操作のレジスタ読み出しステージ内での実行や，ロード遅延を排除するためのメモリ・アクセスと ALU 操作ステージの統合を提案し，Warren ベンチマークを用いた解析によってそれらの有効性を検証した.

提案したパイプライン構成は非常にシンプルであり，われわれが開発を行っている Pegasus プロセッサのように，RISC をベースに Prolog を指向した高度な命令を含む命令セットを構築していくような場合には，例外制御の実現などにおいて非常に効果的である.

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) 瀬尾和男，横田隆史：Pegasus Prolog プロセッサにおける並列データパス操作の導入，情報処理学会論文誌，Vol. 32, No. 11, pp. 1457-1466 (1991).
5) 横田隆史，瀬尾和男：Prolog プロセッサ Pegasus-II のアーキテクチャとチップ実装，情報処理学会アーキテクチャ研究会資料，92-6 (1992).
6) 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).
7) 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).
8) 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論文集，Vol. 88, No. 3, pp. 73-80 (1988).
9) 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).
10) Seo, K. and Yokota, T.: PEGASUS: A RISC Processor for High-Performance Execution of Prolog Programs, *VLSI 87*, pp. 261-274, North-Holland (1987).
11) Seo, K. and Yokota, T.: Design and Fabrication of Pegasus Prolog Processor, *VLSI 89*, pp. 265-274, North-Holland (1989).
12) Yokota, T. and Seo, K.: Pegasus—An ASIC Implementation of High-Performance Prolog Processor, *Proc. EURO ASIC '90*, pp. 156-159 (1990).
13) Hennessy, J. L. and Patterson, D. A.: *Computer Architecture: A Quantitative Approach*, p. 594, Morgan Kaufmann Publishers (1990).
14) Chow, P.: MIPS-X Instruction Set and Programmer's Manual, Technical Report No. CSL-86-289, CSL, Stanford University (1986).
15) 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).
16) Touati, H. and Despain, A.: An Empirical Study of the Warren Abstract Machine, *Proc. 4th SLP*, pp. 114-124 (1987).
17) Warren, D. H. D.: Applied Logic—Its Use and Implementation as Programming Tool, Technical Note 290, AI Center, SRI International (1983).

(平成3年12月24日受付)
(平成4年7月10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㈱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 年同大大学院修士課程修了．同年三菱電機㈱入社．現在，同社中央研究所にて，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.