人工知能向き高級言語マシンの方式研究

瀧, 和男

(Degree)
博士（工学）

(Date of Degree)
1987-01-23

(Date of Publication)
2010-01-08

(Resource Type)
doctoral thesis

(Report Number)
乙1059

(URL)
https://hdl.handle.net/20.500.14094/D2001059

神戸大学博士論文

人工知能向き高級言語マシンの方式研究

昭和61年10月

瀧　　和　　男

目　次

第１章　緒　論

人工知能の研究は、機械に人間の頭脳の処理を模倣させるための方式の基礎研究として1960年前後に開始され、1970年代にはいると現実の世界に役立てるための応用研究の流れも加わり、現在では自然言語処理、エキスパートシステムなどの実用例をはじめとして、数多くの分野で活発な研究が行なわれている。この流れの中でLISP言語は当初から記述言語として使用され人工知能研究の発展を支えてきたとともに、研究の進展につれて言語仕様にも改良が加えられていった。一方Prolog言語は、探索問題の記述や不完全データ構造の取扱いなどの点でLISPよりも強力な言語として最近になって注目を集め出し、活発な研究が始められている。

これらの言語は、実行時のデータ型判定などの動的処理やリスト操作によるメモリの消費などのために、汎用計算機上で使用した場合の実行時間の遅さとメモリ容量不足が問題となった。またLISP言語の発展の中から産まれた対話型のプログラミング開発環境の考え方を十分にサポートするためにも汎用計算機の機能は十分でなかった。これらの問題点を解決するために、大容量のメモリを持ちLISPを高速実行するパーソナル型の専用マシンとして米国で研究され始めたのがLISPマシンであり、近年になって同じ問題を抱えるProlog言語用に研究の始められたのがPrologマシンである。近年の人工知能応用分野の急激な拡大につれ、これらの人工知能向き高級言語専用マシンの必要性は格段に高まりつつある。

ところがこれらの専用マシンを実現しようとした場合、選択すべき方式上の明確な指針がなく、ある規模のハードウェアを用意したときに得られる速度の目安もなかった。また大きな投資をして開発した専用マシンが同規模の汎用マ

シンに比べてどの位高速かも不明であった。

本論文では、これらの人工知能向き高級言語専用マシン実現に関する疑問を一掃するため、LISP向きおよびPrologを拡張した言語向きの２つの専用マシンの試作と評価を行なうことにより、次に示す４項目すなわち、

(1) LISP言語およびProlog言語の高速実行向きアーキテクチャとはいかなるものか。また両者に違いはあるか。

(2) 実用マシンを指向したときのアーキテクチャ上の特徴は何か。また実験マシンに比べてのハードウェア量の増加はいかなる程度か。

(3) インタプリタ向きに設計した専用マシンとコンパイルコードの実行向きに設計した専用マシンの性能差はどの程度か。

(4) 同規模のハードウェアで構成された専用マシンと汎用マシンの性能差は、LISPやPrologに限った場合にいかなる程度となるか。

について検討と考察を行ない、設計上のアーキテクチャに関する指針と、性能やハードウェアコストに関する数値的な目安を与えようとするものである。

本論文の章構成は次のとおりである。

第２章では、人工知能向き言語用の専用マシンに対する要求項目と、専用マシン実現のための基本的方式を示し、マイクロプログラムによるインタプリタ方式とコンパイル方式の長所欠点、スタンドアロン方式とバックエンド方式各々の特徴について述べる。

第３章では、LISP言語の高速実行向きアーキテクチャの研究を主目的に、バックエンド方式とマイクロプログラム化インタプリタ方式によるLISPマシンを設計・試作し、性能とハードウェアアーキテクチャに関する評価を加え、考察のためのデータを提示する。本研究は神戸大学にて行ったものである。

第４章では、Prologを拡張した言語向きの専用マシンについて、高速化と実

用化の両方を目的にスタンドアロン方式とマイクロプログラム化インタプリタ方式で設計し、試作・評価を行なう。本研究は、日本の第５世代コンピュータプロジェクトの一環として（財）新世代コンピュータ技術開発機構にて行ったものである。また評価結果からは、LISP向きとProlog向きアーキテクチャについて、さらに実用マシンのためのアーキテクチャとハードウェアコストについて考察する。

第５章では、２つの専用マシンに対して中間言語命令とコンパイラを設計し、各マシンにおける性能を評価する。またLISPマシンについては直接マイクロコード生成形のコンパイラについても検討を加える。また２つの処理系の比較から、インタプリタ向きに設計した専用マシンとコンパイルコード向きに設計した専用マシンの性能差、同規模の汎用マシン上の処理系との性能差について考察する。

第２章

人工知能向き高級言語マシンと方式

２．１　緒　言

本章でははじめに、人工知能向き言語として代表的なものにLISPとPrologがあり、LISP言語の発展の中で産まれてきた専用マシン化への要求がProlog言語にも当てはまることを述べる。つぎに高級言語専用マシンを設計する場合の基本的な選択点として考えるべき実現方式について述べ、次章以後で採用した方式の基礎を与える。

専用マシンへの要求事項には、高速化、メモリの大容量化、パーソナル化、プログラミング環境の整備などがあげられる。高速化を実現する為には、言語の機能とマシンハードウェアの機能の開きすなわちセマンティックギャップを小さくすることが必要であり、そのために３つの方式が存在する。また実行メカニズムに関してはファームウェアによるインタプリタ方式と言語向き機械語命令へのコンパイル方式があり、各々長所欠点がある。ハードウェアの実現形態についてはパーソナル化に適するスタンドアロン方式と製作工数の小さいバックエンド方式があり、実現しようとするシステムの目的に応じ、それぞれ方式の組合せを使い分ける必要がある。

２．２　人工知能向き言語と専用マシン

２．２．１　人工知能向き言語

人工知能という言葉は、「機械が頭脳の処理を模倣できるような方法」を捜し求めるという分野に対して1956年にJ.McCarthyにより命令されたものである。人工知能向き計算機言語とは、すなわちこの分野の研究の道具として発明され発展してきたものといえる。

人工知能の研究には、知識の表現、探索、推論などの非数値的な処理が不可欠であり、このための計算機言語は数値以外の記号操作を主な目的とすることから記号処理言語とも呼ばれている。

記号処理言語のもっとも初期のものには、簡単な定理証明システムの記述用に作られたIPL があるが、本格的な言語は1958年にJ.McCarthyにより考案されたLISPが最初のものであり、1962年にLISP 1.5としてIBM7090 型計算機上にインプリメントされている[McCarthy 66][Terashima 85] 。LISP言語は、

(1) リストというデータ構造とそれを操作する基本演算

(2) リスト表記のためのＳ式表現

(3) ラムダ計算法に基づいた関数の評価方式

(4) 条件式とそれを用いる再帰関数定義

などをベースに多くの実用的機能が付加されたものであるが、LISPが人工知能研究用の中心的言語として改良を加えられながら使われつづけてきた第１の理由は、リストにより複雑なデータ構造が容易に表現され、データ構造操作の為の関数定義もきわめて容易に行なえるところにあると考えられる。実際に人工知能の研究分野の中で、ゲーム、知識表現、自然言語処理、知識ベースシステ

ム（エキスパートシステム）などは、主に米国の研究機関で開発されたいくつものLISP言語の方言にもとづいて研究が進められてきたものである。またこれらの研究分野からの要求によりLISP言語自体も改良され、Inter LISP、MacLISP をはじめとする多くの実用的な処理系が生まれてきた[McCarthy 78] 。

LISP以外の主な人工知能向き言語には、問題解決と定理証明向けにC.Hewittらによって1968年に考案されたPLANNER 、PLANNER から発展した言語で1972年にG.J.Sussman により開発されたCONNIVERがある[Yokoi 76-a,-b] 。PLANNERではパターンマッチによるデータベースの検索や手続きの呼出し、自動後戻りによる検索（バックトラック）を実現しており、CONNIVERでは探索の制御性の向上とコルーチン機能を持たせ、いずれもLISPより高いレベルでの問題記述を可能としている。ただし両言語とも処理系の記述にはLISPが使われており、LISPの記号処理言語としての汎用性を示しているともいえる。

このような関係からLISPは、1970年代までの人工知能向き言語、そして汎用記号処理言語としての中心的役割を果たしてきたものといえる。

一方LISP以外のもう１つの人工知能向き言語として、1980年代にはいって注目されだしたものにPrologがある。Prologは1971年にフランスのA.Colmerauerにより考案されたが、当初は試行錯誤過程を組み込んだ構文解析プログラムとして実現され、これが述語論理における証明過程として説明できることからPrologの名が与えられたといわれている[Furukawa 84] 。その後1974年に英国のR.KowalskiによりPrologのプログラミング言語としての性格付けが行なわれ[Kowalski 74] 、1977年にはエジンバラ大学のD.H.D.WarrenらによりDEC 社のPDP-10用にコンパイラも備えた効率の良い処理系（DEC-10 Prolog ）[Warren 77][Bowen 81] が作られ、実用的なプログラミング言語としての地位を確立し

た[Robinson 83] 。その後Prologはヨーロッパを中心に徐々に広まっていったが、1982年に日本の第５世代コンピュータプロジェクトで基本言語として採用されたのをきっかけに、急速にその知名度を高め、利用できる処理系の数も増えた[Mizoguchi 84-a][Nakamura 84] 。

Prologの特徴としては、

(1) 導出原理[Robinson 65] にもとづく自動定理証明方式に論理的基礎を置いていること。

(2) ユニフィケーションと呼ばれるパターンマッチ機構を有し、パターンによるデータベース検索、手続きの呼出しや、双方向の引数受渡しなどを可能としていること。

(3) 自動後戻り（バックトラック）機構を用いた深さ優先の解探索メカニズムを備えていること。

(4) ユニフィケーションを利用して不完全データ構造の取扱いが自然に行なえること。

などが上げられる。これらから、Prologは記号論理学上の基礎を持つことによるプログラムの正当性の証明やプログラム変換への期待が持たれること、探索問題に関してはLISPに比べて数段簡潔な記述が可能なこと、完全には決まっていないデータ構造を結果とするようなLISPにないプログラミングスタイルを取り得ることなど、多くの優れた点を持つものと期待されている。実際に自然言語処理[Tanaka 84] 、エキスパートシステム[Mizoguchi 84-b]、CAD[Uehara 84] その他多くの分野で着実に利用者を増やしており、同じ問題をLISPからPrologに書きなおしたことにより大幅にプログラムサイズが縮小されたという報告[Mizoguchi 83]もなされている。

このようにPrologは、近年になってLISPと並び称される人工知能向き言語に

成長して来たということができる。さらに米国ではLISPが主流であり、ヨーロッパと日本ではPrologが優勢であるという構図はしだいに弱まり、米国においてもPrologの研究と利用が盛んになりつつある。今後当分の間は、歴史とプログラムの蓄積を持つLISPと、新しいが強力なPrologとは、２大人工知能向き言語として共存してゆくことが期待される。

２．２．２　専用マシン化への要求

再びLISP言語の開発初期の頃へ話しを戻す。当初J.McCarthyにより作られたLISP処理系はバッチ処理方式であったが、1963年にL.P.Deutsch が対話方式による最初の処理系をDEC 社のPDP-1 型計算機の上に作成した。その後DEC 社の計算機がLISPのインプリメントに都合の良い語構造やスタック命令、ポインタ操作機能などを積極的に取り入れたこともあって、LISP処理系は主にDEC 社のPDP-6 、PDP-10といった対話処理の可能な計算機の上で開発され改良されていった。その後２大LISPと呼ばれるようになったマサチュセッツ工科大学(MIT)のMacLISP [Moon 74] 、Xerox 社PARCのInterLISP [Teitelman 74]も、PDP-10の上で開発されている。

LISPは早い時期から解釈実行系（インタプリタ）を取り込んだが、対話処理の可能な計算機へ移ることによってその利点が生かされ、考えてはプログラムを追加し試すという新しいプログラミングスタイルを発展させることになった。これは考えることとプログラムを作ることとテストすることを一体化した新しい計算機の利用形態を開拓したという点で意義が大きく、後にプログラム開発環境という考え方のもととなったものであり、人工知能研究者のための強力な道具ともなった。

第２章

人工知能の研究が進むにつれてプログラムのサイズも大きくなり、計算機の処理速度の低さと記憶容量の小ささが問題となり始め、特にタイムシェアリングで計算機を使っている場合には深刻な問題となった。スピードとメモリ容量がLISPを使う上で特に問題となるのは、実行時のデータ型判定や再帰呼出しにともなう実行環境の保存などの動的処理（コンパイラで静的に決められない処理）が多いこと、またリストを基本データ型としているため、リスト操作によりメモリ領域を動的にどんどん消費してゆくといった理由による。

これらの問題意識から、

(1) 動的処理をハードウェアでサポートし、LISPを高速実行できる計算機

(2) 大容量のメモリが使える計算機

(3) １人で占有できる計算機

さらに

(4) より良い対話型のプログラム開発環境を提供してくれる計算機

といった新しい計算機像が生まれ、半導体技術の進歩に支えられて、小型で安価なLISP言語専用計算機（LISPマシン）の実現の可能性が高まってきた。

LISPマシンの先駆けとしては、1973年Xerox PARCのL.P.Deutsch の提案したバイト命令形のLISPマシン[Deutsch 73]があり、その改良版はInterLISP のインプリメンテーションの１つとして、1970年代の後半にXerox のAltoマシン[Siewiorex 82]上にインプリメントされ[Deutsch 79]、後に発表されるXeroxのLISPマシンシリーズに多くの影響を残している。一方MIT でも、1974年にCONSと呼ばれるLISPマシン[Knight 74][Gereenblatt 74] が発表され、翌年には稼働し始めている。さらにCONSの後継機であるCADR[Knight 81] の設計も進められ、後に商品化されるLISPマシンの原形となった。

一方Prolog言語については実用化されてからの歴史がまだ浅いが、1982年著者がPrologを使い始めた当時、もっとも処理速度が速く実用的であったDEC 2060計算機上のDEC-10 Prolog コンパイラを用いた場合でも、すでにメモリ容量不足やタイムシェアリングで使用した時の遅さが目立ち始めていた。これは実行時のデータタイプの判定や実行環境保存のためのメモリの動的割付などの実行時処理、構造体データを取扱うための動的メモリ割付けに起因する大量のメモリ消費などLISPの場合とほとんど同じ原因による問題であった。したがってこれらを解決するためには、LISPマシンと同じ方向の専用マシン化が必要であるといえる。

Prolog言語用専用マシン（Prologマシン）については、高速化のためのアーキテクチャを追求する方向の研究として２，３の例があり[Warren 83][Tick 84][Tamura 84-b]、高速化のための研究は進みつつある。しかしながらLISPマシンと対比できるような実用マシンレベルの方式の研究はこれからの課題として残されている。

以上のようにLISPとPrologという２つの人工知能向き言語それぞれのための専用マシンを考える場合、専用マシン化への要求はほとんど同じであり、その基本となるものは、

(1) 高速化への要求

(2) メモリの大容量化への要求

であり、その実現を容易にさせる為に

(3) パーソナル化の要求

があり、さらにLISPの発展の中で生まれたプログラム開発環境の考え方を生か

す為に

(4) 対話型入出力装置の強化の要求

があり、最後にパーソナルマシンの弱点を補強する為に

(5) ネットワーク化の要求

が存在するものと考えることができる。

２．３　高級言語専用マシンの実現方式

２．３．１　セマンティックギャップの解消による高速化

高級言語と呼ばれるものには、 FORTRAN、 ALGOL、 COBOL、PL/1、PASCAL、 C、 Ada、LISP、PROLOGなど多くのものがあり、目的に応じて使い分けられている。これらの生まれた背景は言語毎に異なっているが、例えば機械語でプログラムを組む労力を軽減し、より人間の思考に近い水準でプログラムできるようにするといったすでに存在しているマシンの側から出発したものや、アルゴリズムの記述を容易かつ明確にしようとする立場のもの、抽象データ型などの新しいプログラミングの概念を導入しようとしたもの、あるいは記号処理などの特定の分野向けに機能が決められていったものなどがあり、その目的に応じて言語の持つ機能も異なっている。

例えば初期のFORTRAN などのように、言語の機能がほとんど実在する計算機を意識して決められたものでは、言語上の機能と機械語命令で実現されている機能との差が比較的小さく、プログラムで表現された処理は機械語命令の簡単

な組み合わせに変換される。ところがALGOL のようにネストした制御構造やブロック構造にもとづく変数のスコープを持つ言語では、レジスタとメモリおよびその間の演算と転送しかなかった計算機では、言語の持つ機能を実現するのに長い機械語命令のステップ数を必要とした。

すなわち、言語の要求する実行制御機能、データ操作機能、データ領域の管理機能と、計算機ハードウェアの持つ機能との差が大きいほど、プログラム実行時のステップ数を必要とすることになる。これらの差のことを一般にセマンティックギャップと呼んでおり、目的とする言語と計算機ハードウェアとの間のセマンティックギャップを小さくするほど、実行速度は高まるということができる。

特定の言語向けに専用マシンを作って処理速度を上げようという方向は、実にこのセマンティックギャップを小さくした計算機を設計しようという動きである。そのための第１の方法は、ハードウェア機能を高め言語からの要求に近づけることであり、これは言語向けのハードウェアアーキテクチャの探求という研究テーマとなる。第２の方法は、IBM360型以来ハードウェアと機械語命令の隙間を埋める手段として定着したマイクロプログラム手法を活用することであり、命令語の機能レベルを高いめに設定しておき、マイクロプログラムの最適化により実行ステップの短縮化をはかって、見かけ上ハードウェアの機能レベルを高めようとするものである。第３の方法は、ソースプログラムを分析することによって動的な変数割付けを局所的な静的割付けに変えたり、手続きの呼出し順を変えて入れ子構造をなくしたりして、よりハードウェアとのセマンティックギャップの小さい言語にプログラム変換することである。この方法は普通はとられないように見えるが、実は高級言語のコンパイラの中では意味上これと同じことが行なわれているのであり、最適化コンパイラを設計するとい

うことは、ハードウェアとのセマンティックギャップの少ない言語系を決めそこへのプログラム変換方法を設計しているともいえるのである。

以上に述べたように、セマンティックギャップに着目した高級言語専用マシンの高速化のアプローチとしては、

(1) 言語向きハードウェアアーキテクチャの探求

(2) マイクロプログラムによる最適化と適度な命令レベル（機能）の設定

(3) 最適化コンパイラの設計

の３つがあるといえる。

２．３．２　インタプリタ方式とコンパイル方式

汎用マシン上の処理系の方式には、ソースプログラムのマシン内部表現をインタプリタプログラムで解釈実行するインタプリタ方式と、ソースプログラムを機械語命令列へ変換してから実行するコンパイル方式がある。またこれらの中間に当たるものとして、ソースプログラムを中間言語にコンパイルしてそれを実行サポートプログラム（単純なインタプリタ）で実行するものもある。

LISPやPrologのコンパイラを持つ処理系では通常インタプリタも備えており、後者はソースプログラムとの対応やデバッグ情報の管理が容易であるため主にデバッグに使用される。インタプリタでデバッグサポートなどに凝らないものは設計が容易であるが、コンパイラに比べてプログラムの実行速度は通常10倍以上遅い。コンパイラは最適化処理を行なおうとするほど設計が難しくなるが、実行速度はきわめて速い特徴を持つ。

専用マシン上の処理系でも、汎用マシンと同じくインタプリタ方式とコンパイル方式が考えられる。専用マシンではマイクロプログラムが自由に使えるた

め、インタプリタプログラムもマイクロプログラム化することにより高速化が図れる。一方コンパイル方式では適当な中間言語命令を設計しそれをターゲットとしたコンパイラを作ることになる。インタプリタでは実行時の処理をマイクロプログラムレベルで最適化するのに対し、コンパイル方式ではマイクロプログラムレベルの最適化に加えてコンパイル段階で静的な最適化をかけるため速度面で有利になる。

インタプリタ方式は設計が易しく、マイクロプログラム化すればそれなりの速度が得られるが、デバッグ機能などを充実させてゆくと、速度の低下と設計工数の増大を招く。一方コンパイル方式は適切な中間言語命令と最適化コンパイラの設計に時間と経験を要するが、完成すると高速化が期待できる。

これらにより専用マシンを設計する場合に手軽に作りたければインタプリタ方式が良いが、その場合に処理速度を低下させたくなければデバッグ用のインタプリタを分離することが必要であり、また速度の限界を極める必要がある場合には工数と経験を積んでコンパイラを作る必要があるといえる。またコンパイル方式でもデバッグ用のインタプリタを別に用意する必要がある。

２．３．３　スタンドアロン方式とバックエンド方式

専用マシンにおけるスタンドアロン方式とは、専用マシン自体が入出力装置を有し、専用マシンのCPU で入出力装置の制御やオペレーティングシステムの実行をすべて行なうものである。一方バックエンド方式とは適当な汎用計算機をホストプロセッサとして持ち、入出力制御やオペレーティングシステム実行のある程度の部分をホストプロセッサに任せてしまう方法である。

前者は後者に比べ、専用マシン側のハードウェア量と設計コストが増えるほ

か、多重割込みの機能やオペレーティングシステム実行のための多重プロセスのサポートなどアーキテクチャの変更が必要と予想される。一方後者のバックエンド方式はホストプロセッサとの交信機能は単純で入出力制御プログラムの開発も必要なく、言語実行の高速化に注力した設計が可能となるが、ホストプロセッサを含めたハードウェア量は、前者より大きくなる傾向を持つ。

これらから、パーソナル型の人工知能向き言語専用マシンには、入出力装置の制御をきめ細かくできることや総合したハードウェアサイズの小さいことから、スタンドアロン方式が適しており、言語の高速実行向きアーキテクチャだけを追求する実験マシンには、実現の容易なバックエンド方式が適するといえる。

２．４　結　言

本章でははじめに、人工知能向き言語として代表的なものにLISPとPrologがあることを述べ、LISP言語の発展の中から産まれてきた専用マシン化への要求をまとめるとともに、それがProlog言語についても当てはまることを示した。つぎに高級言語専用マシンを設計する場合にもっとも基本的な選択点として考えるべき実現方式について述べた。

LISPとPrologの両言語に共通する専用マシン化への要求事項は、まず記号処理言語専用マシンへの基本的な要求として、

(1) 高速化への要求

(2) メモリの大容量化への要求

があり、つづいて使い易さやプログラム開発環境の充実の点から

(3) パーソナル化への要求

(4) 対話型入出力装置強化の要求

があり、最後にパーソナルマシンの能力を補完する為に

(5) ネットワーク化の要求

がある。

高級言語専用マシンの実現方式のうちまず高速化の方式に関しては、言語の機能とマシンハードウェアの機能の開きすなわちセマンティックギャップを少なくする方法として、

(1) 言語向きのハードウェアアーキテクチャを設計すること

(2) マイクロプログラムによる最適化を行ない、それに適した命令レベル（機能）を設定すること

(3) 最適化コンパイラを設計すること

の３つがある。また基本的な実行メカニズムとしてはマイクロプログラムによるインタプリタ方式と、適当な機械語命令を設定しそこへコンパイルして実行するコンパイル方式の２つがあり、前者は設計がし易く、後者は最適化により高速化が図れるという特徴を持つ。またハードウェアの実現形態からは、入出力装置の制御も全部専用マシンで行なうスタンドアロン方式と、汎用マシンをホストプロセッサとしたバックエンド方式がある。前者はマンマシンインタフェースを重視したパーソナルマシンに適するが言語実行以外の部分の設計工数を必要とするのに対し、後者は特定言語実行の高速化に注力して設計できるため実験マシンに適するという特徴をもつ。以上により高級言語マシンを設計する場合には、その目的に合わせた実現方式の組み合せを選択することが重要であるといえる。

第３章

インタプリタ方式・バックエンド方式によるLISPマシンの試作と評価

３．１　緒　言

本章では、２大人工知能向き言語の１つであるLISP言語用の専用マシンについて、処理の高速化を目的に、ハードウェアアーキテクチャとファームウェアの検討を行なう。さらにマシンの試作をとおして、採用したハードウェアとファームウェアに評価を加え、LISP言語向きマシンアーキテクチャのあるべき姿を追求する。

本マシンは基本方式として、第１に設計の見通しの良さからマイクロプログラムによるインタプリタ方式を、第２に実験マシンとしてハードウェア量を押さえ早期に完成させるためにバックエンド方式を採用した。

LISP言語向きのハードウェア機能としては、

(1) 高速かつ強力なスタック操作機能

(2) フィールド処理およびビット処理機能

(3) データタイプの判定による多方向分岐や多様な条件ジャンプ機能

(4) 高速なメモリアクセス機能

(5) パイプライン処理や並行処理機能

を盛り込み、ビットスライスプロセッサエレメントなどのLSI を中心にハード

ウェアを実現した[Taki 78-a][Taki 79-d]。インタプリタは、その全体をマイクロプログラム化し、ハードウェアスタック他の高速性の活用に努めた[Taki 78-b] 。

評価用プログラムとして第２回LISPコンテスト[Takeuchi 78] の問題により、処理性能とインタプリタの動特性を測定し、他システムとの性能比較とハードウェア各部の有効性の評価を行なった[Taki 79-b][Taki 79-c][Taki 79-e] 。

３．２　LISPマシンシステムの設計

３．２．１　方式の選択と設計方針

高速化を目指した専用マシンを試作しようとするときの重要な要素として、実験的に作るか実用マシンとして作るかの選択がある。本システムでは、限られた時間内でLISPの高速化のためのアーキテクチャを検証することが最大の目的であることから、実験マシンに徹した方式選択を行なった。

まず基本方式としては、マイクロプログラムによるインタプリタ方式とバックエンド方式を採用した。インタプリタ方式は、LISPの実行メカニズムの解析により、効率的なマイクロプログラム化の見通しが得られるのに対し、コンパイル方式ではLISP用の命令セットを決めるのに十分な処理系作成の経験が必要であったこと、またバックエンド方式は、高速化の研究に関係の少ない機能をホスト計算機上で実現することにより試作ハードウェア量の縮小と処理系の作成期間の短縮が期待できたことにより採用を決めた。

LISP言語向きのハードウェアの導入に当たっては、LISP言語の中の主に次の処理に着目した。

(1) ネストした関数の呼出し復帰の管理

(2) データ型の判定による処理の分岐

(3) リストセルへのアクセス

(4) 変数のバインディング

(5) ガーベジコレクション

これらをもとにインタプリタを高速化するハードウェアをマイクロプログラムから使い易い形で、可能な限り実装することに努めた。

また実装に当っては早期の完成を目指して市販のLSI を多用し、以下の特徴を持たせることとした。

(1) ビットスライスプロセッサエレメントと大容量ICメモリチップの使用

(2) マイクロプログラムシーケンサLSI と書きかえ可能なマイクロプログラムメモリの実装によるマイクロプログラム制御の導入

(3) 高速高機能ハードウェアスタック、マッピングメモリ、フィールド／ビット処理回路などの採用

(4) 水平形マイクロプログラム命令を用いてインタプリタのマイクロプログラム化を行ない高速化を図る。

３．２．２　システム構成

開発したLISPマシンシステムの構成図を図 3.1に示す。システムの中核となるLISPプロセッサはプロセッサモジュールとメモリモジュール（32ビット64k語）から構成されミニコンピュータ(DEC社LSI-11) のバスに接続されている。

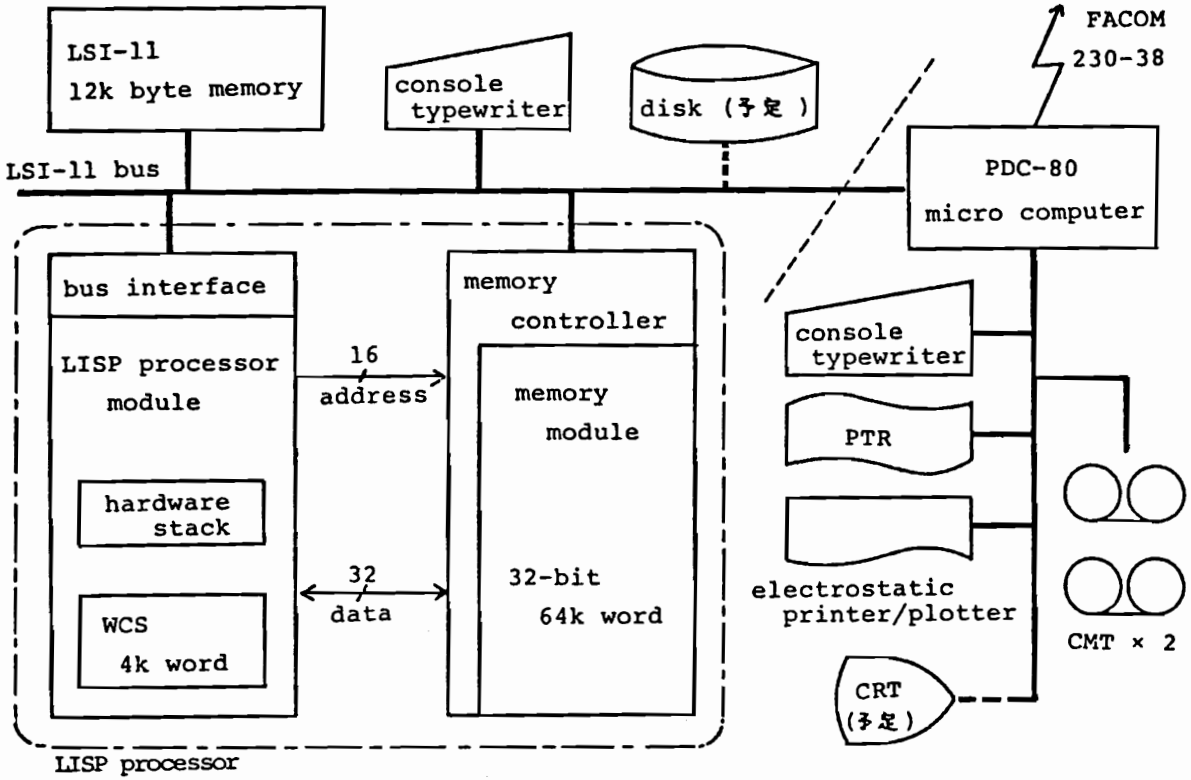

図3．1　LISPマシンシステムのハードウェア構成

ミニコンピュータはページモードでメモリモジュールにアクセスできるとともに独自の主記憶を持ちLISPプログラムと並行して動作できる。仕事は２つあり一つは始動時や保守時にマスターCPU として、LISPプロセッサのプログラムモードや初期化を行なう。もう一つは実行時に、LISPプロセッサからの割込を受付け、入出力プログラムの一部を担当する。そのほか時間監視や、外部からの緊急要求の受付けも行なう。

プロセッサモジュールは、処理幅が16ビット、メモリモジュールとのデータ受け渡しは32ビット幅で行なわれる。すなわち主記憶 1語にcar 部とcdr 部16ビットずつを持ち同時に読み書きを行なう。また書込はバイト単位でも可能である。

３．２．３　LISPプロセッサのハードウェア

３．２．３．１　LISPプロセッサの構成

LISPプロセッサの高速処理に必要とされるハードウェア機能としては、

(1) 高速かつ強力なスタック操作機能

(2) フィールド処理およびビット処理機能

(3) 多様な条件ジャンプ機能

(4) 高速なメモリアクセス機能

(5) パイプライン処理や並行処理機能

などが考えられる。

本LISPマシンは上述の５つの機能を念頭に置いて、市販のビットスライスプロセッサエレメント(ALU) とマイクロプログラムシーケンサLSI を用いて設計している。LISPプロセッサ部のハードウェア構成を図 3.2に示す。高速かつ高

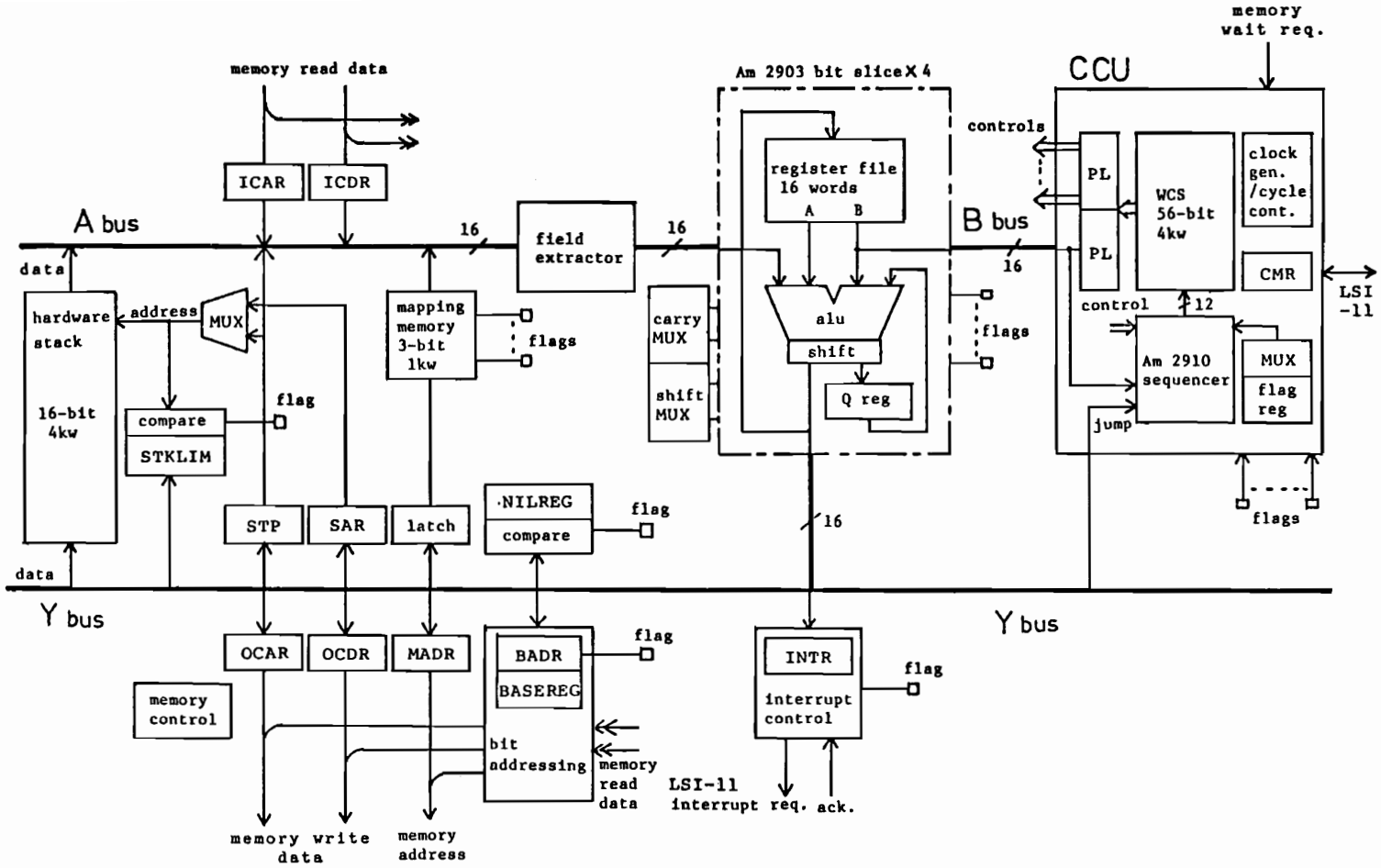

図3．2　プロセッサモジュールのハードウェア構成

機能のハードウェアスタック、フィールド抽出回路、ビットアドレッシング回路、マイクロプログラムレベルでのマルチウェイジャンプ、間接ジャンプ機能、豊富な条件テスト回路、メモリ番地を 3ビットコードに変換するマッピングメモリ回路がALU のバスに接続されており、マイクロプログラム用の4k語のWCSとシーケンサから構成されるコンピュータコントロールユニット(CCU) の制御下に置かれている。

スタックは4k語 (1語16ビット) の固定長であり、内部バス構成はA バス、B バス、Y バスの各16ビットの 3バス構成である。

A バスがデータのソースバスであり、Y バスがデスティネーションバス、またB バスにはマイクロプログラム中の定数が与えられる。ALU での演算は、

(1) ALU レジスタどうし

(2) ALU レジスタとB バスデータ

(3) ALU レジスタとA バスデータ

(4) A バスデータとB バスデータ

との間でそれぞれ可能である。またマイクロプログラムシーケンサはY バスとも結ばれていて、演算結果の値が示す番地へジャンプすることができる。このことによりマルチウェイジャンプやスタックに保存しておいた番地へのリターンが許されマイクロプログラムレベルでの再帰呼出しを可能にしている。またALU での演算結果や、ビットアドレッシング回路からの信号はフラグレジスタを介してマイクロプログラムシーケンサに結ばれ、条件ジャンプに利用される。

３．２．３．２　ビットスライスプロセッサエレメント

ALU はAdvanced Micro Devices社の 4ビットスライスであるAm2903を 4個使用している。その特徴は、(i) チップ上に16語のレジスタファイルを持ち任意

の２つをデータソースとして独立に指定できる。(ii) ALUに対するソースデータの２つをともに外部から与えることができる（A バスとB バス）などである。(ii)の機能はALU をレジスタと切離して利用できるため便利であり、A バスソースとマイクロプログラム上の定数との間の演算に利用している。最小サイクルタイムは100nsec 程度である。zero, negative, carry, overflow のフラグ用の出力端子を備えており条件ジャンプのテストに用いている。

３．２．３．３　コンピュータコントロールユニット

マイクロプログラム制御部CCU はAm2910マイクロプログラムシーケンサ、WCS 、PL（パイプライン）レジスタ、CMR 、フラグレジスタ等から成り立っている。 1レベルのパイプライン制御を行なっており、ALU の実行中に次のマイクロ命令の読出しを並行して行なう。

Am2910は 4,096語のWCS に対するアドレス付けが可能で、μ-PC （マイクロプログラムカウンタ）の他にサブルーチン用の 5段のスタック、ループカンタを持っている。

WCS は 1語56ビットを 4,096語実装しており素子はアクセスタイム150nsecの4kビットMOS メモリである。

CMR はLSI-11から読書きできる 8ビットのレジスタである。LISPプロセッサの実行、停止、初期化の制御に用いられ、またattention 0,1 の各ビットはフラグレジスタに結ばれていて、LSI-11からLISPプログラムに対して何らかの要求があることを示す。

３．２．３．４　ハードウェアスタック

70 nsec の高速メモリによる4k語の固定長スタックである。スタックへのア

クセスは２つのスタックポインタレジスタSTP とSAR により、SAR は主にスタック内部へのアクセスに用いる。いずれもカウンタタイプのレジスタで、スタックをソースとしたときの自動減少、デスティネーションとしたときの自動増加のモードを有する。これらによりALU とSTP 、SAR 間のデータ転送の回数を減らすことができる。STP 、SAR を用いることにより、スタックからのデータの読出し、書込みとALU での演算処理が 1マイクロサイクルで可能である。またスタックからスタックへのデータ転送を 1マイクロサイクルで完了できる。LISPインタプリタでは関数の呼出しやリターンのとき、また引数の移動の際などスタックに対するデータ転送は多く、これらの機能の活用によりマイクロプログラムのステップ数を減少させることができる。

スタックリミットレジスタ(STKLIM)にはあらかじめ値をセットしておき、この値を越えてスタックが伸びたときはフラグがセットされて、スタックオーバーフローが簡単に調べられる。

３．２．３．５　フィールド抽出回路とマッピングメモリ

フィールド抽出回路はA バスとALU との間に入りデータのマスキングとシフトを同時に行なう。データの素通りを含めて 4種類の固定パターンを有する。

マッピングメモリは 3ビット×1k語の書きかえ可能なメモリで、主記憶アドレスを入力として 3ビットのコードを出力する。すなわち主記憶を64語ごとに 1,024区画に分割し、各区画に利用種別を表わす 3ビットのタグを付けたと考えることができる。図 3.3がLISPマシンのメモリ割当であるが、マッピングメモリによりアドレスから利用種別のコードがただちに得られる。このコードやフィールド抽出回路の出力を定数（アドレス）と加算することによりインデックスジャンプのような形のマルチウェイジャンプが実現できる。

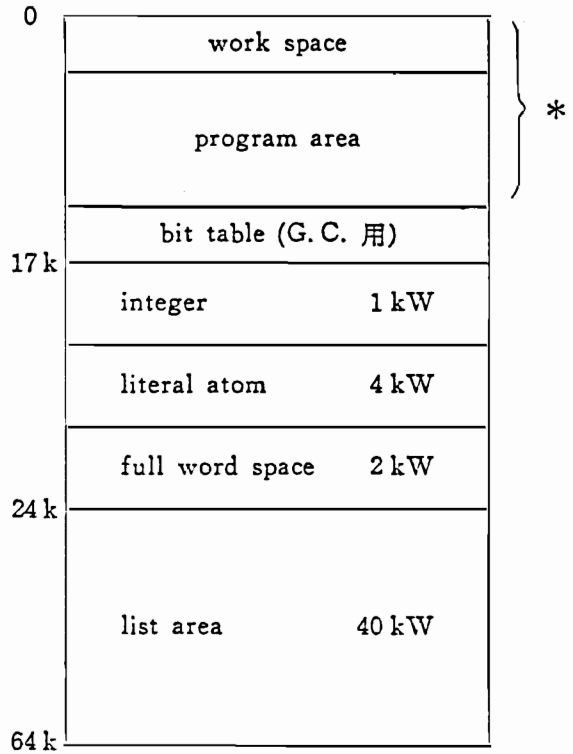

図3．3　メモリ割当て

データ型	語数	表現
小整数	0	ポインタの値そのもの，$-8192 \leqq n \leqq 8191$
整数	1	31 30 … 0 sign | (2の補数表現で整数エリアに格納)
文字アトム	4	31 … 16 15 12 11 … 0 関数本体 | 属性 | 属性別飛び先 flag | 引数個数 | top-level-value property-list | print-name 未使用 | 未使用
リスト要素	1	31 … 16 15 … 0 cdr | car

図3．4　データ型とその表現

マッピングメモリの出力はA バスの他にデコーダをとおしてフラグレジスタに接続されておりリスト、文字アトム、数値などの判定が簡単に行なえる。

３．２．３．６　メモリ動作とその他のレジスタ

メモリとデータの受け渡しはICAR(Input Car Register)、ICDR(Input Cdr Register) 、OCAR(Output Car Register) 、OCDR(Output Cdr Register) を通して行なう。MADR(Memory Address Register) にアドレスを書込むとメモリ動作は自動的に始まる。アクセスタイムは375nsec 、サイクルタイムは675nsecである。メモリ動作が進行している間の 1サイクルはメモリに関係のあるレジスタを除いて普通のALU オペレーションが可能である。メモリオペレーションはreadとread while writeの 2種で、read while writeはメモリセルの内容の読出しと同時に別のデータの書込が行なわれるモードである。

またLSI-11に割込を発生させるためのレジスタとしてINTRがある。INTRにデータを書込むとLSI-11に対して割込みが発生しそのデータが割込ベクタアドレスとして送られる。受付けられるとフラグレジスタのiackフラグがセットされる。

３．２．３．７　ビットアドレッシング回路

主記憶上のビットテーブルに対してビットテストとビットセットを行なう。ガーベジコレクション時にスタックを用いたマーキングを行うので１語に対して 1ビットのマークビットを用意すると64k ビットすなわち2k語のビットテーブルとなる。ガーベジコレクションルーチンではテーブルをすべてクリアしたあとBASEREG にテーブルの先頭アドレスを書込む。以後マーキングアルゴリズムに移り、例えば 8,000番地のリスト要素に対してマークの有無が知りたけれ

ばBADRにTEST指定で 8,000を書込む。するとメモリ参照が起り、 2つ後のマイクロ命令サイクルでフラグレジスタにマークの有無が現れる。ビットセットの場合にも同様にSET 指定でBADRに書込めば対応ビットがセットされる。

３．２．３．８　マイクロ命令と 4つのジャンプ命令

ALU 、マイクロプログラムシーケンサ、スタック、外付けのレジスタ類を制御するために56ビットのマイクロコードを使用する。マイクロ命令は 2つのタイプに分かれ、タイプ 1はALU に対して16ビットの定数を与えて定数演算できるタイプ、タイプ 2はフラグをテストして条件ジャンプのできるタイプである。ALU の動作についてはいずれのタイプも同様に指定できる[Taki 78-a] 。

このようにマイクロ命令を56ビットと長くとったため、本システムでは 4種類のジャンプ命令を用意することが可能となった。

(1) 無条件ジャンプ命令：タイプ 2の命令で、他のALU 演算と並行オペレーションが可能である。

(2) 条件ジャンプ命令：タイプ 2の命令で条件フラグで示された条件が満たされたときジャンプする。他は(1) と同様。

(3) インデックスジャンプ命令：タイプ 1の命令で、マイクロ命令中の定数にマッピングメモリまたはフィールド抽出回路からの出力データをALUで加算した番地へジャンプする命令。この命令を用い出力データによって異なる飛び先番地にジャンプするマルチウェイジャンプを実現している。

(4) 間接ジャンプ命令：タイプ 1の命令でALU での演算結果の番地へジャンプする命令。

これらの命令群はインタプリタの高速化に大きく寄与している。マイクロ命

令の実行サイクルは基本的に300nsec である。

３．２．４　LISP言語の仕様

本処理系で現在用意しているデータ型としては、小整数、整数、文字アトム、リスト要素の 4種類がある。これらの表現を図 3.4で示す。

文字アトムをリストの形で表現せずに連続した領域に情報を置くことにより、属性のチェック、属性値の取出しを容易にした。また変数の値はold-value をスタック上に退避するshallow-binding を採用しているので、自由変数の取出しについては速度の向上が期待できる。主記憶上の領域をデータ型によって分割し、マッピングメモリ回路により高速に型判別を行なっている。

関数型は、EXPR, FEXPR とシステム関数用としてマイクロコードで記述されたMSUBR, MRSUBR, MRFSUBRを用意した。MSUBR だけは再帰呼出しを許さない関数型である。

MSUBR　：CAR, CDR, CONS, EQ, NULL, ATOM, ADD1, etc.

MRSUBR　：EVAL, APPLY, EVLIS, EQUAL, MAPCON, etc.

MRFSUBR ：COND, PROG, AND, OR, LIST, DE, SETQ, etc,

３．２．５　ファームウェアインタプリタ

開発したLISPマシンにおいてはインタプリタ全体がマイクロプログラム（ファームウェア）化されており、特に処理速度の向上を求めている。ここではインタプリタの構造とその高速化のため取入れられた試みについて論じる[Taki 78-b] 。

３．２．５．１　式の読込みと結果のプリント

プログラムは式の読込み、評価、結果のプリントのサイクルを繰返すことにより実行される。式の評価はLISPプロセッサが実行するが、式の読込みと結果のプリントはLSI-11とLISPプロセッサが相互通信を行いながら実行している。LSI-11は入力の場合は、入力文字列を括弧、ドット、アトムのストリングに分離し、アトムの登録を行ないLISPプロセッサに制御を渡す。LISPプロセッサはストリングを 2進木に変換し、式の評価を行なう。一方出力の場合は、括弧、ドット、アトムへのポインタの形式をしたLISPプロセッサの実行結果をLSI-11は出力文字ストリングに変換しながら出力する方式をとっている。

３．２．５．２　式の評価方式と高速ハードウェアスタックの利用

本システムは高速のハードウェアスタックを実装しているが、スタック上には関数の実行に対応してフレーム(frame) という領域が作られる。フレームの構造を図 3.5に示す。

フレームヘッド(frame head)には関数本体へのポインタ、プログラムカウンタ(PC)、後述するマイクロプログラムカウンタ(MPC) を保存する領域(old PC とold μ-PC)および 1つ前のフレームヘッドを指す旧フレームポインタ(old FP) が含まれている。ローカルスタックには関数本体実行時には、計算の中間結果やマイクロサブルーチンへの引数などが置かれる。FPはフレームヘッドの位置を、STP はスタックトップポインタで現在のスタックの先頭を示す。PC、MPC 、FPはAm2903のレジスタファイル中にとられる。

プログラムカウンタは式の評価の際、ユーザプログラムの 2進木リストを指して順にたどってゆくときに用いる。MPC はAm2910中の真のマイクロプログラムカウンタ（μ-PC ）にロードするマイクロプログラムアドレス値を保持する

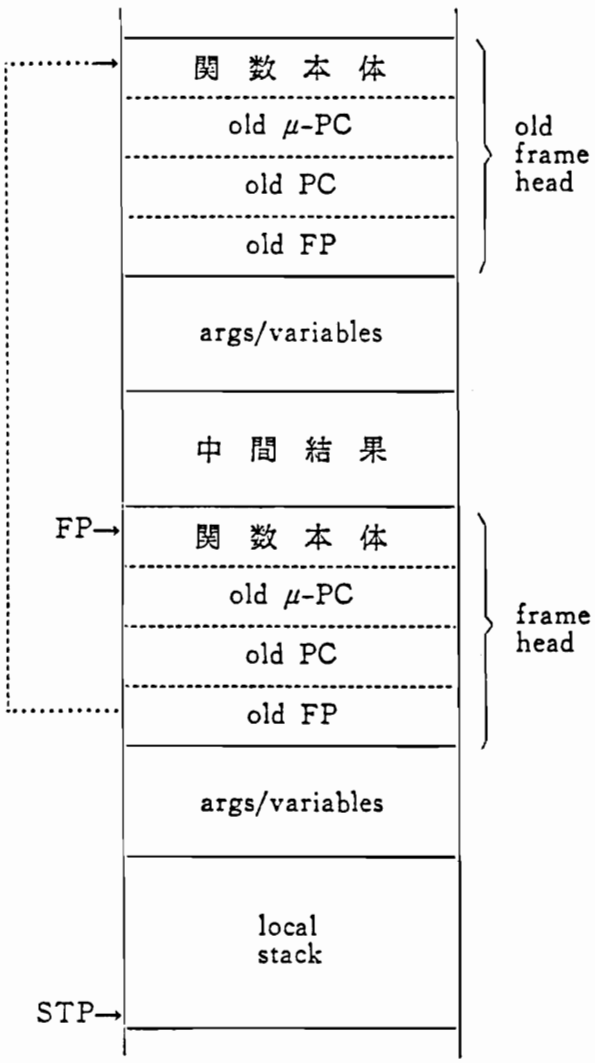

図３．５ Frame の構造

のに用いる。

一例として、式（関数 1、引数 1、引数 2、引数 3）を評価する場合を考えてみる。EVALの中で式がアトムでないことが分った時点でフレームが作られ、関数 1の性質が調べられ、引数の評価方法と評価後の関数本体の渡し方が決る。引数を評価してはローカルスタックに積んでゆき、完了後関数本体へ制御を渡す。制御の移動はマイクロプログラム化された関数の場合には関数本体へマイクロプログラムジャンプすることにより行なわれる。関数本体の中で引数を用いて計算が終了すると結果を持って呼出しもとのフレームへもどる。もし引数に再び（関数 2、引数1'）の形の式があらわれた場合には新しいフレームを作って、引数リスト（引数 2、引数 3）と引数の評価方法、関数本体への渡し方の情報を保存して、引数1'の評価に移る。関数評価の手順はおおよそ以上の通りである。

引数の評価ではEVILS 関数を使わずにスタックを利用して引数の受け渡しを行なうのでむだな自由リストの消費がなく、ガーベジコレクションも起りにくくなっている。またマイクロプログラムの戻り番地をスタック上のold-μPCに格納すること、引数リストをスタック上に展開することによりマイクロプログラムレベルでの再帰呼出が可能となっており、インタプリタ全体のマイクロプログラム化を可能とするとともに、再帰呼出制御の効率化を可能としている。また変数のバインディング(binding) 形式はshallow-binding で自由変数の値の取出しについて速度の向上が期待される。またバインディング処理では実引数をApply 処理に渡す際、リストの形でなくスタックに積んだままの形で渡すこととアトムのold value とその値セルの番地をスタックに残す手法を用いることにより処理の高速化を実現している。

３．２．５．３　ハードウェアの利用によるインタプリタの高速化

インタプリタで多様される条件判断において、If then elseタイプの処理を繰返すのを避け、マルチウェイジャンプ、間接ジャンプを多用する。マルチウェイジャンプはマッピングメモリの出力の 3ビットコード、またはフィールド抽出回路によって取り出された 4ビットコードと定数アドレスとの和の番地へ 1マイクロサイクルでジャンプするもので、データ型や文字アトムの属性の種類により多分岐するときに用いる。

さらに演算結果がnil であるか、アトムか、リストか、数値か、文字アトムであるかといった単純な判定は専用のフラグをテストすることにより、 1ステップで済ませる。さらに条件または無条件ジャンプの際にはALU 関係の処理を平行して行ない可能なかぎりマイクロプログラムのステップ数を短縮している。

前節で述べたように 2つのポインタレジスタを持つ強力なハードウェアスタックとAm2903中の16個のレジスタを併用することにより、スタックマシンとレジスタマシンの長所を合せ持たせるよう工夫している。

３．３　試作と評価

３．３．１　ハードウェアとソフトウェアの製作

本システムのハードウェアサイズはおおよそつぎのとおりである。LISPプロセッサ部分は17㎝×22㎝の万能基板15枚を用いた構成で電源を含め 3段の19インチラックに収納している。内訳は、プロセッサモジュールが 7枚、メモリモ

ジュールが 8枚で、はんだ付けとワイヤラッピングにより配線を行なっている。総IC個数は約 600個である。

LSI-11側ではタイマ割込回路、8080マイクロコンピュータインタフェース回路、バス拡張回路を約70個のICを用いて 3枚の基板上に構成し、LSI-11の筐体内に収納している。

ソフトウェアとしてはLISPプロセッサ上で働くマイクロコード、LSI-11側のモニタプログラム、マイクロアセンブラが上げられる。それぞれのサイズはおおよそ次のようになる。

(1) マイクロコード（組込関数55種実装時）

(i)　総ステップ数： 1,369ステップ

(ii)　インタプリタ： 539ステップ

(iii) READ、PRINT ： 183ステップ

(iv)　ガーベジコレクション： 118ステップ

(v)　システム関数群： 212ステップ

(2) LSI-11モニタ： 3.3k 語（16ビット語）

(3) マイクロアセンブラ： 1,300ステートメント(FORTRAN)

となっている。マイクロコードの部分がコンパクトな形で実現されているのが特徴である。

３．３．２　実行時間と動特性の測定

３．３．２．１　プログラムの実行時間

完成したシステムで第２回LISPコンテスト[Takeuchi 78] に出題されたプログラムを実行したときの処理時間を測定した[Taki 79-b] 。その結果の一部を

表３．１　LISPコンテストのテストプログラム実行時間例（msec）

	……INTERPRETER……			……COMPILER……	
	FAST-LISP KOBE UNIV.	HLISP	OLISP	HLISP	OLISP
TARAI-3	55	78	197	17	36
TARAI-4	1,013	1,443	3,727	330	670
TARAI-5	27,538	39,068	100,734	8,905	18,934
TARAI-6	1,011,732	…	…	322,347	…
TPU-1	904	4,790	2,262	1,029	658
TPU-2	3,426	13,411	10,262	2,871+	2,064
TPU-3	1,365	5,684	3,932	1,227++	850
TPU-4	1,815	8,025	5,042	1,707	1,161
TPU-5	238	866	759	181	132
TPU-6	6,728	22,849	20,241	4,893+	3,617
TPU-7	1,311	5,078	4,179	1,053	776
TPU-8	1,187	3,459	3,987	724+	596
TPU-9	819	2,663	2,716	545	424

表 3.1に示す。表記方法と他の処理系の実行時間は文献[Takeuchi 78] にしたがう。測定は出題プログラムを評価するためのEVAL時間に対して行なっており、READ関数、PRINT 関数で行なう前処理および後処理の時間は含まれていない。測定はLSI-11に装備した1msec のタイマ割込みを用い、精度は+1、-0msecとなっている。比較対象としてHITAC-8800上に構成されたHLISP とACOS-800上に構成されたOLISP を取り上げる。これらの処理系はコンテストに参加したシステムでは最も高速な結果を示したシステムである。

３．３．２．２　インタプリタの動特性の測定

表 3.2、表 3.3はTARAI-1 とTPU-6 を実行させたときの総ステップ数、関数やルーチンの呼ばれた回数(count) 、それらの実行ステップの合計が全体にしめる割合である。表 3.3ではステップ数の比率を示しているが、それはほぼそのまま、各ルーチンの実行時間が全時間に占める割合に対応する。表 3.3からARGEVAL ルーチンの比率が高いことが分るが、このルーチンは引数評価用としてEVALルーチンを改良し特に高速化を計ったもので、改良の効果は大きいと考えられる。EVAL1 はEVALルーチンの前処理部分を省略した高速化ルーチンである。　表 3.4は各ルーチンや関数に含まれるハードウェア機能の利用回数を集計し、比率を表わしたものである。ここでいう比率とは、各ハードウェアオペレーションを含むステップの総和が、全実行ステップ中にしめる割合のことで、1ステップに複数のオペレーションを含むことが多く、比率の合計は100%を超えている。

表 3.5はメモリオペレーション数の合計を100%として集計しなおし、そのうちわけを示したものである。ただしread while writeオペレーションの実行時間は、write オペレーションの中に含まれる。表 3.6は直接ジャンプ（表 3.4

表3．2　LISPプログラムの実行時間とマイクロプログラムの実行ステップ数

プログラム	実行時間 (msec)	総実行ステップ数	1ステップの平均実行時間 (nsec)
TARAI-3	55	154,763	355
TPU-6	6,726	19,457,963	346

表3．3(a) TARAI-3 実行時の各関数、ルーチンの呼ばれた回数と、全実行ステップに占める割合

関数またはルーチン(*)名		回　数	比率 (%)
interpreter functions /routines (85.2%)	*EVAL	1,347	13.9
	*EVAL 1	2,523	
	*ARGEVAL	5,047	28.0
	*APPLY	673	26.5
	*EXPR	673	7.4
	COND	673	9.2
system functions (14.8%)	GREATERP	673	9.3
	SUB1	504	5.5
defined fn.	TARAI	673	…

表3．3(b) TPU-6 実行時の各関数、ルーチンの呼ばれた回数と、全実行ステップ数に占める割合

関数またはルーチン(*)名		回　数	比率 (%)
interpreter functions /routines (43.7%)	*EVAL	39,262	
	*EVAL1	331,900	9.3
	*ARGEVAL	390,002	13.0
	*APPLY	15,706	4.2
	*EXPR	15,706	1.2
	PROG	6,311	9.7
	GO	17,049	1.4
	RETURN	6,311	0.5
	COND	42,360	4.4
system functions (56.3%)	SUBST	4,361	17.3
	EQUAL	12,974	11.3
	SETQ	55,320	6.3
	MEMBER	6,392	4.8
	CAR	43,865	3.2
	NULL	25,743	2.1
	ATOM	19,319	1.6
	CDR	22,415	1.6
	OR	12,892	1.5
	CADR	9,343	1.2
	APPEND	1,210	0.8
	LENGTH	4,224	0.7
	CADDR	3,744	0.7
	LIST	2,936	0.5
	EQ	3,569	0.4
	CONS	3,379	0.4
	others	12,310	1.9
externally defined functions	RENAME	2,291	...
	INSIDE	1,836	...
	DISAGREE	7,559	...
	UNIFICATION	2,018	...
	DELETEV	184	...
	URESOLVE	499	...
	GUNIT	4	...
	PNSORT	4	...
	FDEPTH	0	...
	FTEST	0	...
	SUBSUME	1,293	...
	STEST	8	...
	CONTRADICT	8	...
	DTREE	1	...
	TPU	1	...

の条件ジャンプと無条件ジャンプの和）オペレーションを集計しなおして、うちわけを示したものである。jump with other op. とは、ジャンプしながらその他のオペレーションを平行実行することである。

３．３．３　測定結果によるハードウェアの評価

LISP処理のため実装した各種ハードウェアの有効性の評価を実用的なプログラム例と考えられるTPU での測定結果を用いて議論する。

３．３．３．１　ハードウェアスタック

表 3.4よりスタックオペレーションの比率がたいへん高いことが分るが、以下の２点について考察する。

第１はスタックアクセス時間で、本システムではアクセス時間は完全に１つのマイクロプログラムサイクルに組込まれる。したがってスタックアクセス時間は、スタックをデスティネーションとした場合は 0、ソースとした場合はサイクル延長分の75nsecとなる。このことはスタックオペレーションの比率からみて、十分高速化に貢献しているといえる。スタックアクセスの時間を不要にしているのは、ハードウェアスタック、ALU 、バス相互のデータ経路の選び方と、スタックポインタレジスタの仕様および70nsecの高速メモリの仕様によるものである。第２はスタックオペレーションのうち自動減少、自動増加（表中（ ）－，＋（ ）で示したもの）の利用である。スタック操作にしめる割合は共に約70% で、もし自動減少、増加の機能がないとそのためよけいなステップを必要とする。

以上から、ハードウェアスタックを実装して強力な機能を持たせたことはお

おむね成功であったといえる。

３．３．３．２　ジャンプに関するハードウェア機能

表 3.4からジャンプオペレーションの比率もたいへん高いことが分るが、以下の３点について考察する。

第１は、もっとも比率の高い条件ジャンプに関するもので、そのうちわけは表 3.6から分かる。マッピングメモリ関係フラグ、NIL レジスタフラグ、スタックオーバーフローフラグ、ALU 関係フラグのいずれも10% 台の利用率であるが、これらのフラグをなくした場合には、定数と比較の後ALU フラグジャンプとなり 1ないし 2ステップのオーバーヘッドとなる。したがって多様なフラグを用意したことは成功といえる。

第２は、表 3.6のjunp with other op. に関することである。これはジャンプしながら他に、スタックオペレーションやALU オペレーションその他を平行して行なうことで、飛び先での仕事の先まわり実行などを行なうことが多く、直接ジャンプのうち70% に利用されており、マイクロ命令コードを長くとって並行実行を可能にしたことが成功であったといえる。

第３はインデックスジャンプと間接ジャンプについてで、表 3.4より両者の比率の合計は10% 、ジャンプ中に占める割合は26% である。これらは主にインタプリタ中で使用され、マイクロプログラムの再帰呼びだしを許したりマルチウェイジャンプを可能にしたりする。この二者を除くと、インタプリタの構造自体を変更せねばならず、効率の低下は大きい。したがってジャンプアドレスとしてY バスデータが利用できるというバス構造は必要と考えられる。

表３．４　ハードウェア機能利用の割合

オペレーション		比率 (%)			
		TARAI-3		TPU-6	
メモリアクセス	read	15.4	12.8	11.4	10.2
	write		2.6		1.2
ジャンプ	conditional	41.0	17.0	38.3	15.0
	un-cond.		13.0		13.4
	indexed		5.8		3.9
	indirect		5.3		6.1
スタックオペレーション	source	38.0	18.4	49.0	23.4
	destination		19.6		24.6
	stack to stack		1.1		3.4
	[( )−]		[13.0]		[18.9]
	[+( )]		[12.7]		[17.8]
直接数値演算		22.2		19.1	
ALU オペレーション		20.2		20.3	
FEX オペレーション		1.3		0.6	
マッピングメモリ直接		7.0		3.9	
IDLE		1.0		0.4	

FEX: フィールド抽出回路

表3．5　メモリオペレーションのうちわけ

オペレーション	比　率　(%)	
	TARAI-3	TPU-6
READ, car only	19.0	34.0
READ, cdr only	0.0	19.2
READ, car & cdr	64.1	36.2
WRITE CAR	16.9	7.5
WRITE CDR	0.0	2.1
WRITE BOTH	0.0	1.0
read while write	8.5	6.9

表3．6　直接ジャンプのうちわけ

直接ジャンプ オペレーション	比　率　(%)	
	TARAI-3	TPU-6
un-conditional	43.4	47.3
MAP-flag	18.5	17.3
NTL-flag	6.5	12.1
Stack over-flag	5.1	10.3
ALU-flag	26.5	13.0
jmp with other op.	77.4	70.6

３．３．３．３　メモリオペレーションに関する機能

表 3.4から、メモリアクセスのためのステップ比率は11.4% で、10ステップに１度はメモリ参照のあることが分る。本システムの特徴は第一にメモリアクセス時間が短く待ち時間が 1サイクルしか必要でないこと、第二にメモリ待ちのサイクル中に別のオペレーションが可能でありむだ時間にならないこと、第三に読み打し幅が32ビットで、car 部とcdr 部を同時に読出し、メモリアクセス回数を減らせること、第四にread while write機能があることである。

第二点については、表中のIDLEのところが何もしない純粋なメモリ待ちであり、TPU の場合には、全メモリ参照のうち96%((11.4-0.4)÷11.4＝ 0.96) までが、メモリ待ちのサイクル中に別の仕事をしていることになる。すなわち、本システムの実行速度はメモリリミテッドになっていないといえる。

第三の読出し幅については、表 3.5よりcar 、cdr の両方を利用するためのREADが、メモリアクセス中36% をしめ、比較的よく利用されていることが分る。

第四点については、read　while write 機能の利用は、メモリアクセス中の 6.9% 、WRITE オペレーションの65% をしめる。この機能をなくすると、読出してどこかに保存したあと書込みを行なう手順が必要となり、１回当り最低 3 ステップのオーバーヘッドを生じる。

以上から、メモリアクセス関係のハードウェア構成はほぼ成功したといえ、またメモリアクセスがLISP処理の速度低下を招かなくなったことは大きな進歩といえる。

３．３．３．４　定数演算機能

5ステップに１度は利用されており、マイクロ命令コードを56ビットと長くしてこの機能を持たせたことが、有効になったといえる。

３．３．３．５　マッピングメモリとフィールド抽出回路

マッピングメモリを直接読出して利用する割合は、TPU では 3.9% であるが、すべてインタプリタルーチンのインデックスジャンプに利用されるため重要である。マッピングメモリフラグの利用率を合わせると 8.8% となる。フィールド抽出回路の利用率は、TPU では1%を割るが、もし実装しなければ、シフトとマスクにかなりのステップを必要とする。コンパイラやタグマシンの場合にはより有効になると考えられる。

３．３．３．６　ALU オペレーション

ALU オペレーションの比率はTPU では20.3% で、演算もデータ転送も不要のステップはIDLEと単なる直接ジャンプだけであるから、全実行ステップ中の71% は、単にデータの移しかえだけを行っていることになる。したがって、プロセッサモジュールのバス構造を変更しALU を経由しないデータパスを設けることにより、マイクロ命令サイクルをより短縮できる可能性がある。なおビットスライスプロセッサ中のレジスタファイルをソースオペランドとして利用する比率は25% 、デスティネーションとしては35% （全ステップに対して）であった。

３．３．３．７　ビットアドレッシング回路

TPU-6 の実行時に発生した 1回のガーベジコレクションを例にとると、総ステップ数が 462,917、実行時間が157msec 。回収セル数が36,123（全セルの88.2% ）であり、マーキングに実行時間の14.3% 、回収に84.3% を費していた。このときの各オペレーションのステップ比率は、BITSETが 1.0% 、BITTEST が

9.9% 、BIT フラグジャンプが 9.9% 、メモリアクセスが 9.4% 、IDLEが 12.0% であった。すなわちビットテーブルの参照が10.9% あり、ビットテーブル操作には通常はステップ数を要することから、ビットアドレッシング回路は有効と考えられる。

３．３．４　評価結果の考察

以上から、設計段階において効率向上を期待して組込んだ各種ハードウェアはほぼ期待どおりの働きをしていることが確認できたがここでは本システムの高速性についてまとめる。

３．３．４．１　プログラム実行の高速性について

プログラムの実行時間が短い理由として、各ハードウェアの高度な機能によるステップ数の減少の他に、マイクロ命令幅を長くとったことにより、いろいろな部分での並行処理パイプライン処理が行なわれて、機械語命令では不可能な短いステップ数でインタプリタを実現できたことが考えられる。実際にインタプリタ部分のステップ数は 539ステップという小さい値である。またもう一つの見方として、本システムのメモリ構成が、インタプリタの記憶のためのWCS 、データとソースプログラムのための主記憶、インタプリタが制御動作を行なうためのハードウェアスタックというふうに大きく 3ヵ所に分散していて、従来のメモリリミテッドなシステムから脱していると考えられる点である。

３．３．４．２　コンパイラ実装についての考察

コンパイラを実装した場合に、どの程度の速度向上があるかを考察する。表3.3を見ると、システム関数の比率が56.3% （リターン処理を含んでいる）あり、その中から関数からのリターン処理にかかるステップ比率の合計を差引いても残りは約50% である。すなわちコンパイラによって、インタプリタが行なっていた処理ステップが完全になくされても速度向上は２倍にとどまる。実際には中間言語命令のフェッチとデコードのためそれより悪化するはずである。これはTPU の場合であるが、TARAI のようにインタプリタルーチンのしめる割合が極端に高いと、速度向上の割合も高いと考えられる。試みとしてマイクロプログラムでTARAI を記述したときの実行時間はTARAI-1 から 5が各々、14、245、 6,646msecであった。しかしながらTPU の方がより実際的なプログラムであり、本システムではコンパイラを実装しても大きな速度向上は望めないと予想される。このことを逆から考えると、インタプリタによる処理速度がコンパイラの出力コードの実行速度に近づいたものと言えるわけで興味深いものでである。

３．４　結　言

本章では、２大人工知能向き言語の１つであるLISP言語用の専用マシンについて、処理の高速化を目的に、ハードウェアとファームウェアの設計、試作、評価を行なった。

基本方式としてマイクロプログラムによるインタプリタ方式とバックエンド

方式をとり、LISP言語向きハードウェアとしては、

(1) 高速かつ高機能のハードウェアスタック

(2) タグの代わりにデータ型の判定に利用するマッピングメモリ、フィールド処理、ビット処理などの機能回路

(3) 多方向分岐、間接分岐および多様な条件分岐機能

(4) CAR 部とCDR 部を同時に読出し、CPU と並行動作のできるメモリ回路。

(5) １段のパイプラインと水平形マイクロ命令による並行処理

(6) インタプリタと組込関数全体のファームウェア化を目指した、大容量の書替え可能な制御メモリ

を盛り込んだ。また実装に当たっては、ビットスライスプロセッサエレメントとマイクロプログラムシーケンサのLSI 、大容量ICメモリチップなど高集積度のLSI を使用し、ハードウェア量の削減と短期完成を実現した。ファームウェアの設計に当たっては、ハードウェアスタックを活用した呼出復帰情報管理および引数受渡しの高速化、データ型にもとづく多方向分岐を活用したインタプリタ核部の小型化などを実現した。

こうして完成したシステム上で、第２回LISPコンテストの課題プログラムを走らせた結果、処理速度はコンテストに参加していたもっとも高速の汎用大型計算機と比較して、同等またはそれ以上の値を示し、予想を大きく上まわる結果となった。またLISP言語向きハードウェアの有効性をさらに詳細に調べるために、同じプログラム実行時のハードウェアの利用率を測定し、ハードウェアスタック、分岐回路などを中心に、各々のハードウェアの有効性を確認した。このような高速性が実現できた理由として、

(1) LISPインタプリタ向きの水平形で強力なマイクロ命令を用意し、並行制御とパイプライン処理を積極的に取入れた。

(2) 高速のハードウェアスタックと高機能のスタックオペレーションを準備した。

(3) 分岐回路の強化とともにデータ型判定回路その他の機能回路を準備した。

(4) 実装上可能な限り高速のICを用い、しかもボトルネックの発生しないようバランスのとれたシステム設計につとめた。

(5) インタプリタの全体のマイクロプログラム化とインタプリタ核部の小形化を図った。

などを上げることができる。

またコンパイラを実装した場合の速度向上比の予想値がを２倍程度となることを求め、インタプリタをマイクロプログラム化した場合に、インタプリタ方式の速度がコンパイル方式に近づくことを示した。

第４章

インタプリタ方式・
スタンドアロン方式による
逐次型推論マシンの試作と評価

４．１　緒　言

本章では、２大人工知能向き言語の１つであるPrologを強化拡張した言語用の専用マシンについて、アーキテクチャの検討と設計、試作、評価を行なう。本システムの第１の研究目的は、高速処理のための言語向きマシンアーキテクチャの追求である。それと同時に、マシンの実用化の観点からの方式検討と設計を合わせて行ない、実験マシンと比較した場合のアーキテクチャ上の特徴を明らかにする。

本マシンは基本方式として、第１にマイクロプログラムによるインタプリタ方式を採用した。これはLISPマシンと同じく設計の見通しの良さによるものである。第２は実用マシンの観点から、人工知能向き専用マシンへの要求である対話型入出力装置の装備とパーソナル化を実現するため、スタンドアロン方式を採用した。

Prolog系の言語向きハードウェア機能としては、

(1) 複数のスタックを高速にサポートする機能

(2) タグアーキテクチャ

(3) 大容量で高機能のレジスタファイル

(4) データタイプの判定による多方向分岐と強力な条件分岐機能

(5) 高速なメモリアクセス機能

(6) マイクロ命令レベルでの並列制御機能とパイプライン処理

を盛り込んだ。また実用化の観点からは、

(1) マルチプロセス（タスク）のサポート機能

(2) 入出力バスの制御と割込み機能

(3) ページングによる大容量メモリの管理と効率的再配置を行なう機能

(4) RAS （信頼性と保守性）に関する機能

を準備した。CPU 部分は高速の TTL LSI、MSI を中心に実現し、入出力バスにはビットマップディスプレイやマウスなどの対話型入出力装置を実装した[Taki 83][Yokota 83][Taki 84-a][Taki 84-b]。またファームウェアはモジュール化構成をとり、インタプリタ、ユニファイアの高速化とともに、割込み、プロセス切替え、メモリ管理などのオペレーティングシステムサポート機能の充実を図った[Yokota 84][Yamamoto 84]。

評価用プログラムとして第１回Prologコンテスト[Okuno 85]の課題およびいくつかの応用プログラムにより、処理性能と動特性の測定を行ない、他システムとの性能比較とハードウェア各部の評価を行なった[Taki 85][Nakashima 85][Nishikawa 85][Nakajima 86]。

またLISPマシンとの言語向きアーキテクチャの比較、実験マシンと実用マシンとのアーキテクチャおよびハードウェア構成の比較・考察を行なった。

４．２　逐次型推論マシンシステムの設計

４．２．１　方式の選択と設計方針

本システムは、Prolog系言語の高速化を目指した研究用システムであると同時に、日本の第５世代コンピュータプロジェクト[Fuchi 83][Fuchi 84]のソフトウェア研究開発用マシンとして、実用性も重視した方式選択と設計を行なった。

まず基本方式としては、マイクロプログラムによるインタプリタ方式とスタンドアロン方式を採用した。インタプリタ方式については、DEC-10 Prolog [Warren 77] の処理方式などからマイクロプログラム化の見通しが得易いのに対し、コンパイル方式ではProlog系言語向きの命令セットを決めるのに十分な処理系作成経験を必要することによっている。スタンドアロン方式については、ハードウェア量の増加はやむを得ないものとして、後述する入出力機能の充実とパーソナル化の目的から採用を決めた。

本システムは、実用マシンとするために人工知能向き専用マシンに対して要求される次の項目

(1) 高速化への要求

(2) メモリの大容量化への要求

(3) パーソナル化の要求

(4) 対話型入出力装置の強化の要求

(5) ネットワーク化の要求

のすべてを満たすよう、システム設計を行なった。高速化に当たっては、Prolog系言語の処理系の次の特徴

(1) 複数のスタックを用いる実行メカニズム

(2) バックトラック情報をスタックに積むことによる、ユニフィケーション対象の変数がスタックの先頭と深部に分散すること

(3) データ型の判定による処理の分岐の頻度の高さ

(4) テイルリカージョン（述語の尾部に出現する再帰呼出し）の最適化

に着目し、インタプリタの高速化に役立つハードウェアをマイクロプログラムから使い易い形で実装することに努めた。

メモリの大容量化については、実メモリの効率的運用のためにページング方式のメモリ管理ハードウェアを用意し、再配置機能はオペレーティングシステムに持たせることとした。また対話型入出力装置の強化に対しては、入出力バス制御と割込みのハードウェア機能を持たせ、オペレーティングシステム中の入出力装置制御ソフトウェアの設計を楽にするためにマルチプロセス（タスク）機能をハードウェアとファームウェアで実現することとした。さらにRAS（信頼性と保守性）機能としてエラー検出機能、ハードウェアレベルのデバッグ機能などを持たせた。

製作にあたっては早期完成を目指して、無理のない設計と市販の高集積度のIC、安定した実装技術を用いた。マシンの性能目標としては、DEC-10 Prologのコンパイラ版をDEC 2060上で利用した場合と比較して、速度では同等の30K LIPS(Logical Inference per Second ： 1秒当たりの推論回数）、メモリ容量は64倍の16メガ語を設定した。また対話型入出力装置には米国のLISPマシンに見られるようなビットマップディスプレイとマウスを採用した。

４．２．２　システム構成

PSI のシステム構成を図 4.1に示す。大きくCPU 部分と入出力部分に分かれ、CPU 部分は制御部、演算部を中心とし、主記憶、キャッシュメモリ、アドレス変換器を含むメモリ部、I/O バス制御部の各々が、内部バスにより相互接続された構成をとる。入出力部分はIEEE796 規格の標準バスと入出力装置群から成り、市販の装置の実装も可能である。CPU 部分には、保守、立上げ、デバッグサポート用にコンソールプロセッサ（8086マイクロプロセッサ使用）とコンソールバスを用意し、さらに強力なデバッグ用とクロスシステム（DEC 2060上のマイクロアセンブラなど）からのマイクロプログラム等のダウンラインローディング用に、ミニコンピュータ（PDP11/23PLUS）の接続を可能とした。

CPU の内部バスは 2本のソースバス、 1本のデスティネーションバス、制御信号線から成り、 8ビットのタグを含む40ビット幅のデータが流れる。キャッシュメモリの容量は8k語、主記憶は最大で16メガ語、書替え可能な制御記憶(WCS) は16k 語を実装した。また入出力バスはCPU のマイクロプログラムでコントロールされ、主記憶との間の直接のパスは持たない。

入出力装置としては、ビットマップディスプレイ（約1200× 900ドット）、光学式マウス、キーボード、固定式ディスク、フレキシブルディスク、ローカルエリアネットワーク(LAN) 、シリアルプリンタ、デバッグ用CRT ディスプレイを標準装備し、拡張入出力装置として大容量ディスク、レーザプリンタ、磁気テープなどを接続可能とした。コンソールプロセッサは、低速入出力装置のコントローラを兼ね、フレキシブルディスクまたは固定式ディスクからのイニシャルプログラムロード処理も行なう。ビットマップディスプレイは、 2メガバイトのローカルメモリを持ち、文字フォントやウィンドウイメージを蓄える。

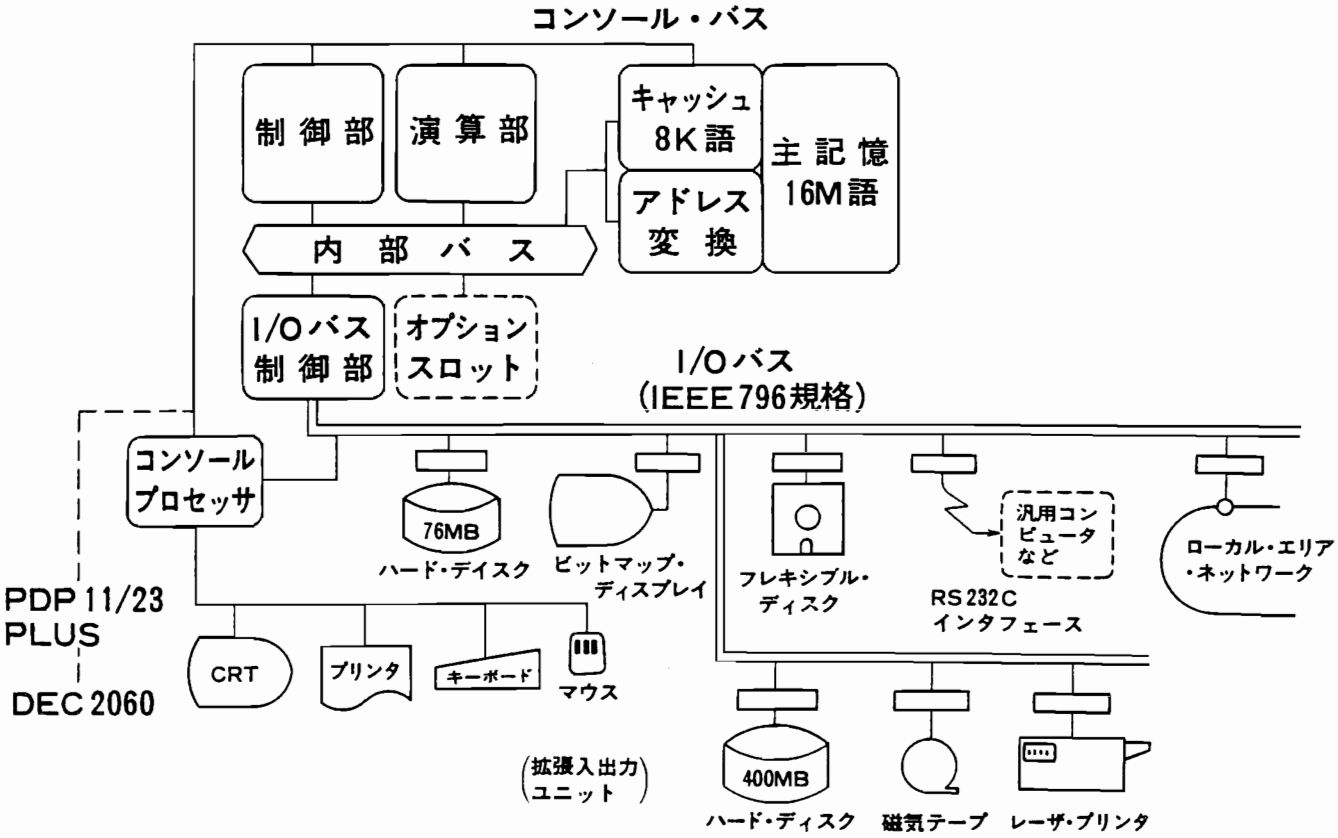

図４．１　逐次型推論マシン（ＰＳＩ）のシステム構成

またラスタオペレーションの機能も持たせた。

４．２．３　アーキテクチャと言語仕様

本節では、機械語命令のレベルから見えるマシンアーキテクチャと、機械語およびその上のシステム記述言語の言語仕様について述べる。

４．２．３．１　語形式とデータ型

PSI の 1語は40ビットから成り、そのうち上位 8ビットはタグ部、下位32ビットがデータ部である。タグ部はさらに 2つに分かれ、上位 2ビットがガーベジコレクション用のマークビット、下位 6ビットがデータ型を表わすタイプタグである。プログラムから取扱えるデータ型には、

Symbolic atom, Integer number, Floating point number,
Stack vector, Heap vector, String

などがある。Stack vectorはユニフィケーションを許す構造体データを表わすものであり、Heap vector は、破壊的代入を許すかわりにユニフィケーションを許さないデータ型でFortran の一次元配列のようなものである。Stringには16ビット、バイト、ビットの各々の種類があり、漢字列、ASCII 文字列、ビットマップディスプレイ用のデータなどに利用する。またHeap vector はシステム記述言語ESP の中でオブジェクト指向機能を実現するのに主に利用する。要素数 7までのvectorは、タグ中に要素数をエンコードして用いている。

他に命令コードに関するデータ型、実行制御情報に関するデータ型が約30種類ある。

４．２．３．２　機械語の設定

PSI ではDEC-10 Prolog のサブセットにいくつかの拡張機能[Takagi 83] を付加した述語論理型言語をマシンが直接解釈実行するレベルの言語（機械語）として設定した。この言語をKL0(Kernel Language Version 0) と呼んでいる。

KL0 の仕様は次のようにまとめられる。

(1) DEC-10 Prolog のサブセット

(2) 拡張された実行制御機能

(3) 破壊的代入を許すデータ型の導入

(4) オブジェクト指向のサポート機能

(5) ハードウェア制御機能

KL0 はPrologと同じレベルの述語論理型言語であると同時に、スタンドアロン型のマシンで必要となるあらゆること（例えばハードウェア制御）が記述できるように設計されている。

(1) の意味の第１は、assert、retract などの機能をKL0 は直接サポートしていないということで、これらはオペレーティングシステムにより実現されている。KL0 では命令コードの呼出しはアドレスで行なわれ、述語名の文字アトムと命令コードアドレスとの対応関係はソフトウェアで管理している。文字アトムもマシンの内部表現ではアトム番号に変換されて持たれており、印字名との対応関係はソフトウェアが管理する。これらの仕様は、多くのモジュールから成る大規模なプログラムを管理する立場からの要求で決められた。

(1) の第２の意味は、入出力に関する組込述語を持たないことで、この機能は(5) に含まれる低レベルの組込述語を使ってソフトウェアで実現している。

(2) はDEC-10 Prolog で実現されている実行制御メカニズム自体を拡張したもの[Takagi 83] で、多段のカットや、特別の事象による手続き呼出し、例え

ばバックトラック発生時や特定の変数への値のユニファイや引数タイプミスマッチなどの例外事象の発生による手続き呼出しなどがある。

(3) はheap vector 型のことで、オブジェクトの表現や、オペレーティングシステム中の管理テーブルなどに使用される。

(4) はシステム記述言語ESP の持つオブジェクト指向の為の、手続き呼出しや変数アクセスをサポートする組込述語のことである。

(5) は入出力装置のレジスタ操作やメモリ管理テーブルの操作、プロセス切替や割込に関する操作のための組込述語のことで、オペレーティングシステム中で使用される。

４．２．３．３　命令語表現

KL0 のマシンの内部表現形式を図 4.2に示す。大きく分けると、プロシジャヘッド部、クローズインデキシングのためのインデクステーブル部、クローズ部（述語本体）からなる。さらに１つのクローズは、クローズヘッダ部、クローズヘッドの引数部、複数のボディゴール部からなり、ボディゴールにはユーザ定義述語と組込述語が現れる。組込述語は１語中にオペレーションコードと 3個までの引数が納められ、各引数は 3ビットのタグを持つ。引数には変数番号や小さな数値を直接置くことができるが、ここに納まりきらない大きなデータのためには別に 1語がとられる。

内部表現に変換されたプログラムはヒープ領域と呼ぶ論理空間に置かれ、ファームウェアインタプリタは 4本のスタックを使ってこれを解釈実行する。

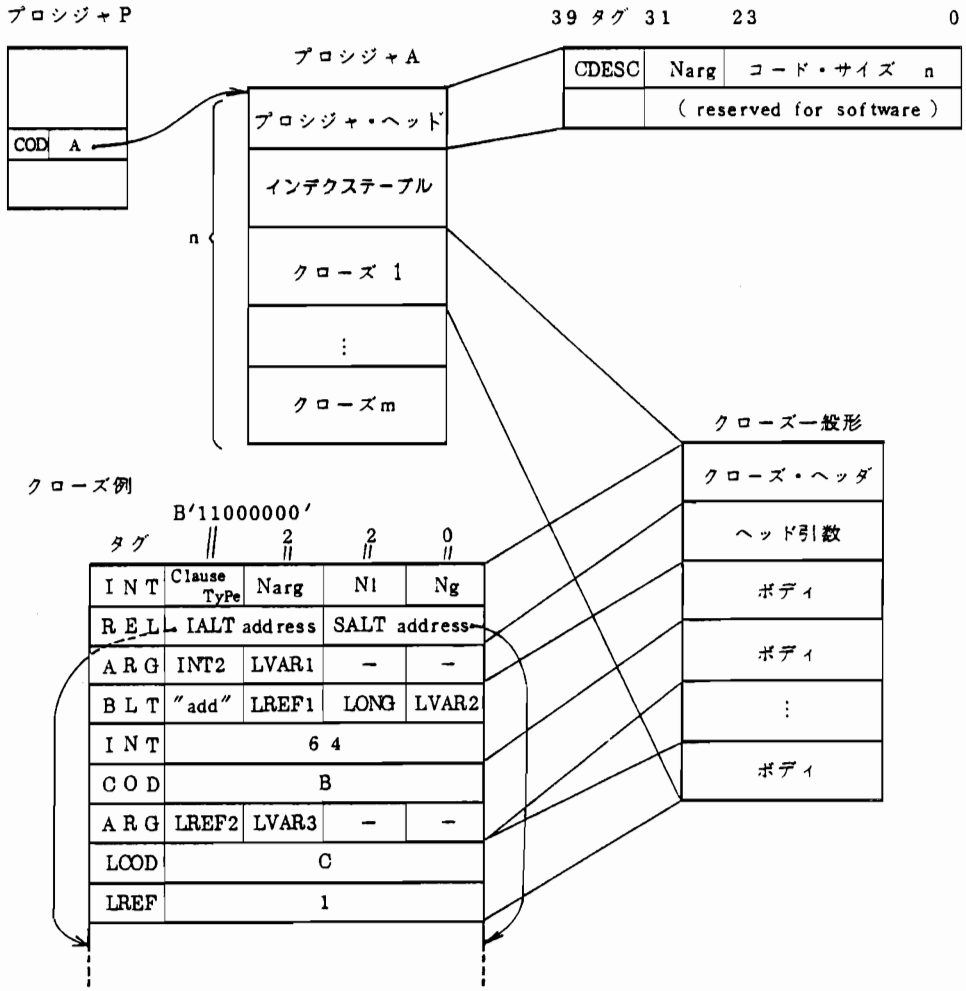

図４．２　KL0 のマシン内部表現形式

４．２．３．４　システム記述言語

システム記述言語はKLO を拡張した述語論理型言語でESP(Extended Self-contained Prolog) と呼ばれる。すべてのオペレーティングシステムとユーザプログラムはESP で記述され、KLO にコンパイルされて実行される。ESP の仕様は次のようにまとめられる。

(1) KLO

(2) マクロ機能

(3) オブジェクト指向機能

ESP の中からはKLO の機能がすべて利用できる。マクロ機能とは、述語の引数部分に数式を書くと組込述語に展開してくれるような機能である。オブジェクト指向機能は、Smalltalk-80[Goldberg 83] に見られるようなクラスシステムと、Flavors[Weinreb 80] に見られるような多重継承を合わせ持つもので、プログラムのモジュール化と階層化に効果が大きい。PSI のオペレーティングシステムとプログラミングシステム約15万行は、すべてESP のオブジェクト指向機能を用いて書かれている。

４．２．３．５　マルチプロセス

PSI 上のプログラムは、プロセスという形をとって実行される。ハードウェアとファームウェアにより63個までのプロセスが並行実行できるようサポートしている。

１つのプロセスは実行環境として、KLO の実行に必要な 4本のスタックとそれを指すポインタ類、プログラムカウンタなどとともに、割込許可レベルなどの若干のハードウェア制御情報を持ち、これらを１まとめにしてプロセスコントロールブロック(PCB) と呼ぶ。ユーザプログラム、OSプログラム、割込処理

プログラムなどは各々別のプロセスとして実行され、休止中のプロセスのPCBは定められた場所にまとめて退避されている。実行中のプロセスのPCB（カレントPCB）はCPUのレジスタ上に分散して存在し、プロセスの切替時にはカレントPCBの中身がファームウェアの制御で入れかえられる。プロセス切替えのタイミングは、割込やトラップ時と、OSのスケジューラの中でプロセス切替の組込述語を実行したときである。プロセス個数は、後述するエリア数の制限から63個であるが、OSやデバイスハンドラの記述には十分な数である。

４．２．３．６　メモリ管理

たくさん存在するプロセスのスタックの論理空間を各々独立にするためと、物理メモリを有効に利用するために、メモリ管理機能を設けた。

KLOの実行に用いる4本のスタックは、各々独立に伸縮し、さらに4本の組がプロセスの個数分存在する。これらのスタックは、1本ずつ独立の論理空間に配置したい。一方命令コードとプロセス間通信に使うデータ領域は、全プロセスで共有できるスタックとは異なる論理空間に置きたい。そのため“エリア”と呼ぶ論理アドレス空間を導入した。PSIの論理アドレスは、図4.3に示すように、8ビットのエリア番号と24ビットのエリア内アドレスからなり、16メガ語の大きさを持つエリアが256個利用できる。各エリアは常に0番地側から使用され、途中に塵がたまれば、ガーベジコレクション時にスライディングコンパクション[Morris 79]により0番地側へ詰め合わされる。

一方、PSIの物理メモリは最大16メガ語が実装可能であるが、プログラム実行時には各エリアの必要とするメモリ容量がダイナミックに変化するため、物理メモリを1k語のページに分割して管理し、ページ単位にオンデマンド方式で各エリアに割当てている。ページ割当てはファームウェアにより実行され、ペ

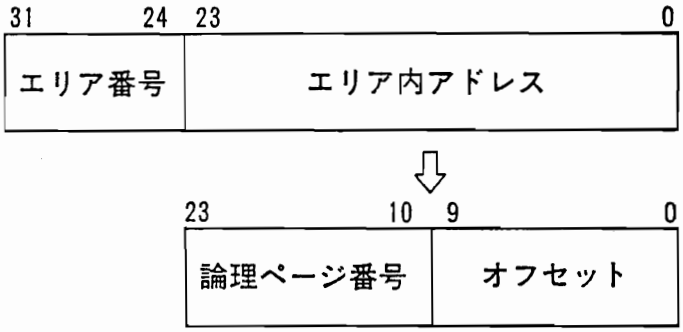

図４．３　アドレス表現

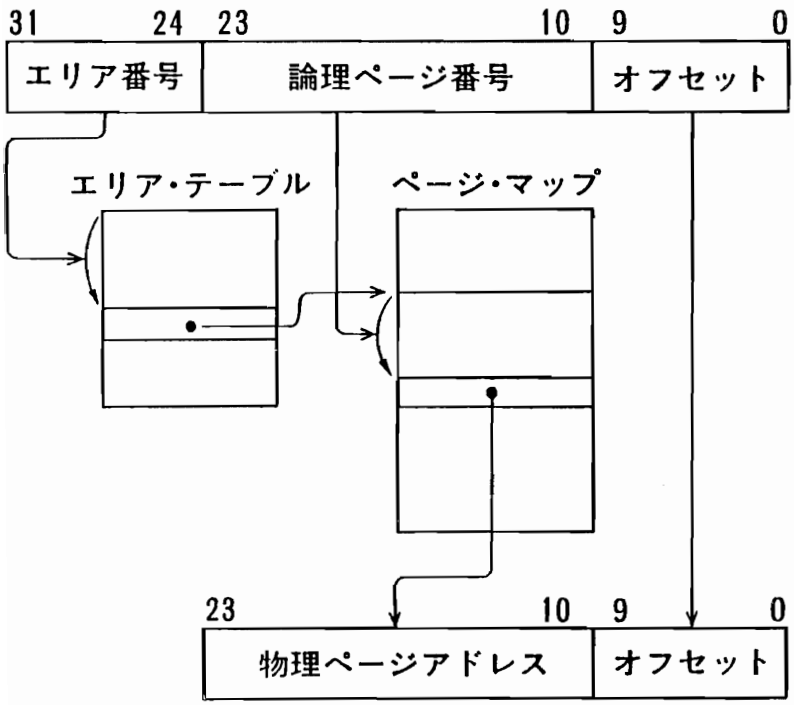

図４．４　アドレス変換機構

ージがなくなったときにはトラップ（後述）を発生してガーベジコレクタを呼出す。図 4.4はアドレス変換機構の概念図で、ページマップベースとページマップの２つのテーブルを用いて変換を行なう。これらのテーブルは、高速化と管理の簡単化のため専用のハードウェアとして用意している。

４．２．３．７　割込み

PSI の割込みは、割込み原因毎にベクタを持つ方式を採用し、ベクタにはプロセス番号を登録しておいて、割込が発生すると登録してあるプロセスへプロセス切替えが起こる。割込レベルは入出力用に 2レベル、タイマーやエラー、処理異常などのために 3レベルを用意しており、 1つのレベルに対して原因対応の複数の割込ベクタがある。入出力用の割込ベクタは16個あり16種の入出力装置がサポートできる。

割込の他にプログラム実行に同期して発生するエラーに対処するため、いつでも割込むことのできるトラップを設けている。

４．２．４　ハードウェア設計

４．２．４．１　CPU 部分の構成

Prolog系言語向きのハードウェア機能としては、

(1) 複数のスタックを高速にサポートする機能

(2) タグアーキテクチャ

(3) インタプリタの最適化のための大容量で高機能のレジスタファイル

(4) データタイプの判定による多方向分岐と強力な条件分岐機能

(5) 高速なメモリアクセス機能

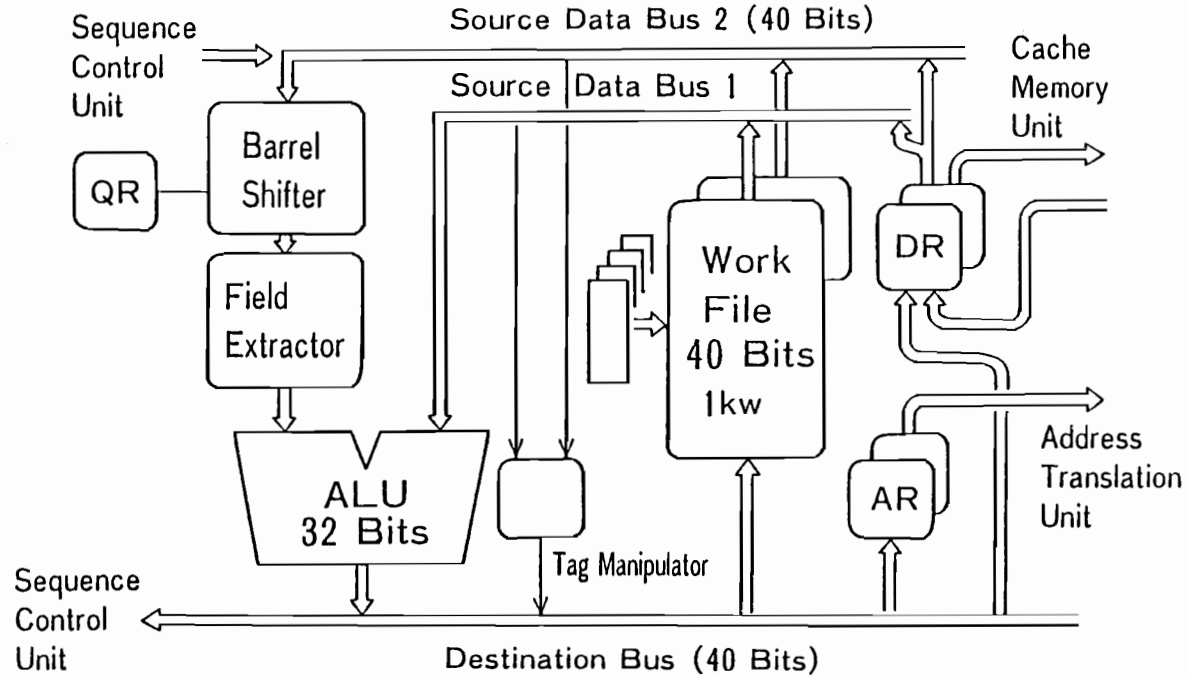
Sequence
Control
Unit
Source Data Bus 2 (40 Bits)
Source Data Bus 1
Cache
Memory
Unit
Barrel
Shifter
QR
Field
Extractor
Work
File
40 Bits
1kw
DR
ALU
32 Bits
Tag Manipulator
AR
Address
Translation
Unit
Sequence
Control
Unit
Destination Bus (40 Bits)

図４．５　演算部の構成

(6) マイクロ命令レベルでの並列制御機能とパイプライン処理

などが必要である。本マシンではこれらの機能を実現するため、スタック向きの機能を持つキャッシュメモリと２組のメモリインタフェースレジスタ、タグ操作とタグ判定による多方向分岐回路、WF（ワークファイル）と呼ぶ1k語×40ビットの多機能レジスタファイル、多様な条件分岐の回路、16k 語×64ビットのWCS(Writable Control Storage) などを実装することにした。この中で(1)の機能を専用のハードウェアスタックとしなかった理由は、Prolog系言語ではバックトラック情報をスタックに積むために述語からのリターン時にスタックが縮まず、呼出元の変数領域がスタックの奥深くに存在する場合がしばしば生じることによる。すなわちユニフィケーション時のスタックアクセスがスタックのトップと深いところに分散し、このためスタックの先頭付近だけを保持するスタックキャッシュのハードウェアでは十分な効果が出ないこと、また全体をハードウェアスタックとするには容量が大きくなりすぎたりプロセス切替え時の中身の入替えの問題が生じることなどによっている。

CPU 部分は、図 4.1に示すように演算部、制御部、メモリ部から成り、それらをコントロールするのがマイクロプログラム方式による制御部（図 4.6）である。それとは別にメモリ部（図 4.7）にも独自に動くシーケンサを持ち、キャッシュオーダが出された後、ミスヒットが起こるとキャッシュ内容の入替え制御などを制御部とは独立に行なうことができる。

CPU 部分のデータ経路は図 4.5の演算部の構成に示すように、40ビット幅の２本のソースバスと１本のデスティネーションバスから成り、ワークファイル(WF)やメモリインタフェースのデータレジスタ(DR)から読出したデータにALUで演算を施し結果をWFやレジスタに書込むまでを１マイクロ命令サイクルで実行する。また同時にバレルシフタによる32ビットまでのシフトと特定パターン

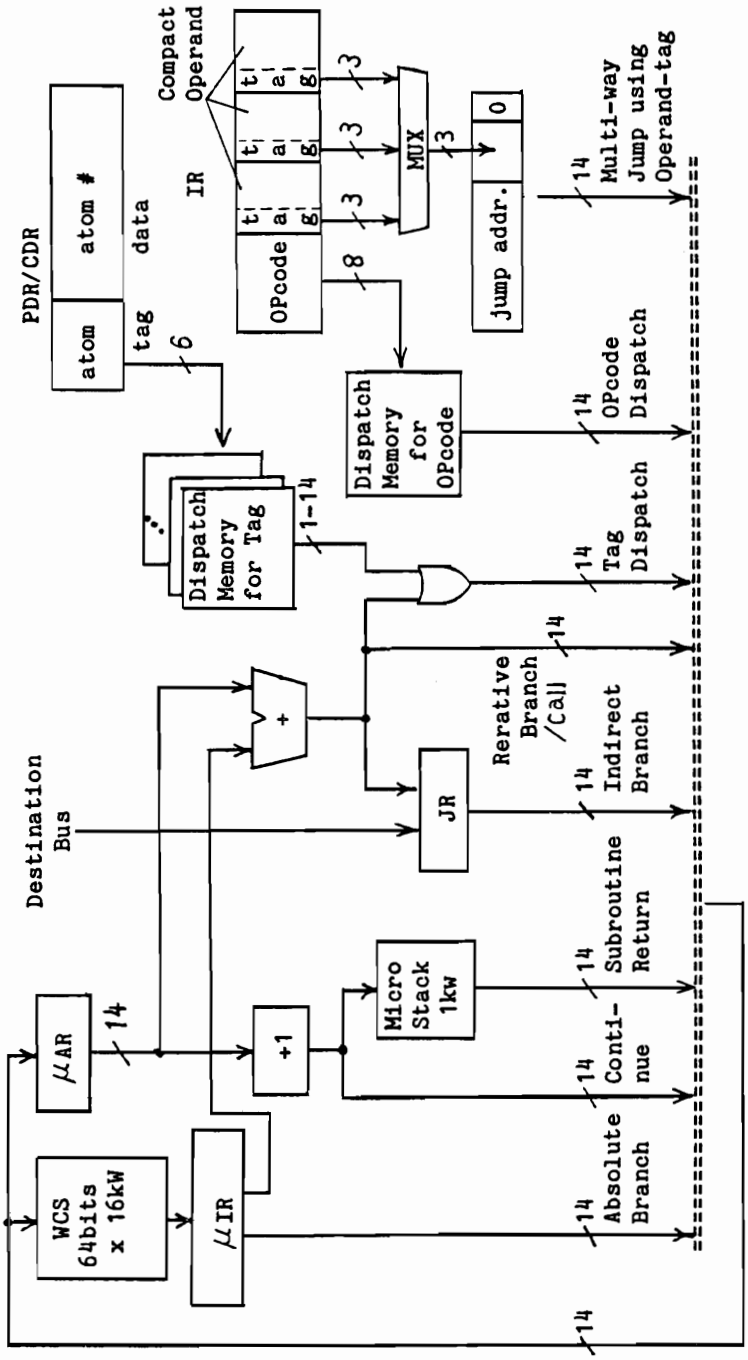
PDR/CDR
atom
atom #
data
tag
6
Compact Operand
IR
OPcode
8
MUX
3
Jump addr.
0
14
Multi-way Jump using Operand-tag
Dispatch Memory for OPcode
14
OPcode Dispatch
Dispatch Memory for Tag
1-14
14
Tag Dispatch
Destination Bus
+
Rerative Branch /Call
14
JR
14
Indirect Branch
μAR
14
+1
Micro Stack 1kw
14
Subroutine Return
14
Conti-nue
WCS 64bits x 16kW
μIR
14
Absolute Branch
14

図4．6　制御部の構成

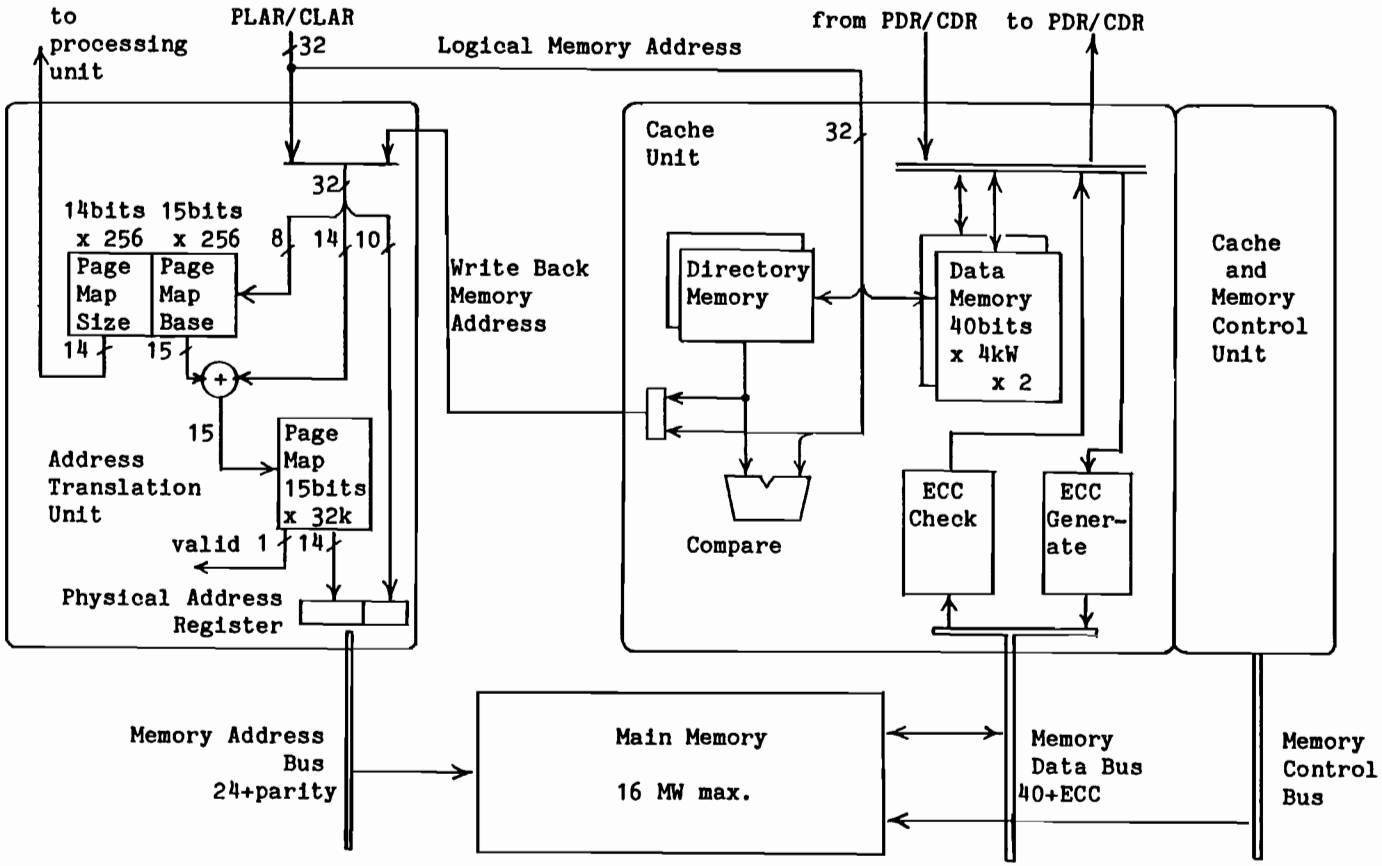
to processing unit
PLAR/CLAR
32
Logical Memory Address
from PDR/CDR
to PDR/CDR
Cache Unit
32
14bits x 256
15bits x 256
Page Map Size
Page Map Base
8
14
10
14
15
Write Back Memory Address
Directory Memory
Data Memory 40bits x 4kW x 2
Cache and Memory Control Unit
15
Address Translation Unit
Page Map 15bits x 32k
valid 1
14
Physical Address Register
Compare
ECC Check
ECC Gener-ate
Memory Address Bus 24+parity
Main Memory
16 MW max.
Memory Data Bus 40+ECC
Memory Control Bus

図4．7　メモリ部の構成

によるフィールド抽出を指定でき、ALU 出力の書込みにはWFとDRの同時書込などのマルチデスティネーション指定を許し、１マイクロ命令で多くの処理をまとめて行なうことを可能としている。また上記のデータ操作と並行してメモリアクセスを指定することにより、マイクロプログラム制御による一種のパイプライン処理が可能である。

本マシンは他に実用化の観点から、

(1) マルチプロセスのサポート機能

(2) 入出力バス制御と割込み機能

(3) ページングによる大容量メモリの管理と効率的再配置の機能

(4) RAS （信頼性と保守性）に関する機能

を実現するため、図 4.4の方式にもとづくアドレス変換回路、複数レベルの優先割込み回路、すべてのレジスタとデータ経路へのパリティビットの付加と主メモリの１ビットエラー訂正回路、コンソールプロセッサからの全レジスタの読み書きとエラーロギングの機能などを準備した。

４．２．４．２ ワークファイル(WF)

WFはメモリインタフェースのDR（２組存在し各々をPDR 、CDR と呼ぶ）とともに演算部の中心をなす多目的、多機能のレジスファファイルで、40ビット×1k語の容量を持つ。先頭部分の16語はジェネラルレジスタ用にdual-port 化されており、２本のソースバスへの読出しとデスティネーションバスからの書込みに各々異なるアドレスを１マイクロ命令中で指定できる。

WFは図 4.8に示すように大きく分けて次の４種のアドレシングモードを持つ。

(1) マイクロ命令による直接アドレス指定

(2) 間接アドレス指定

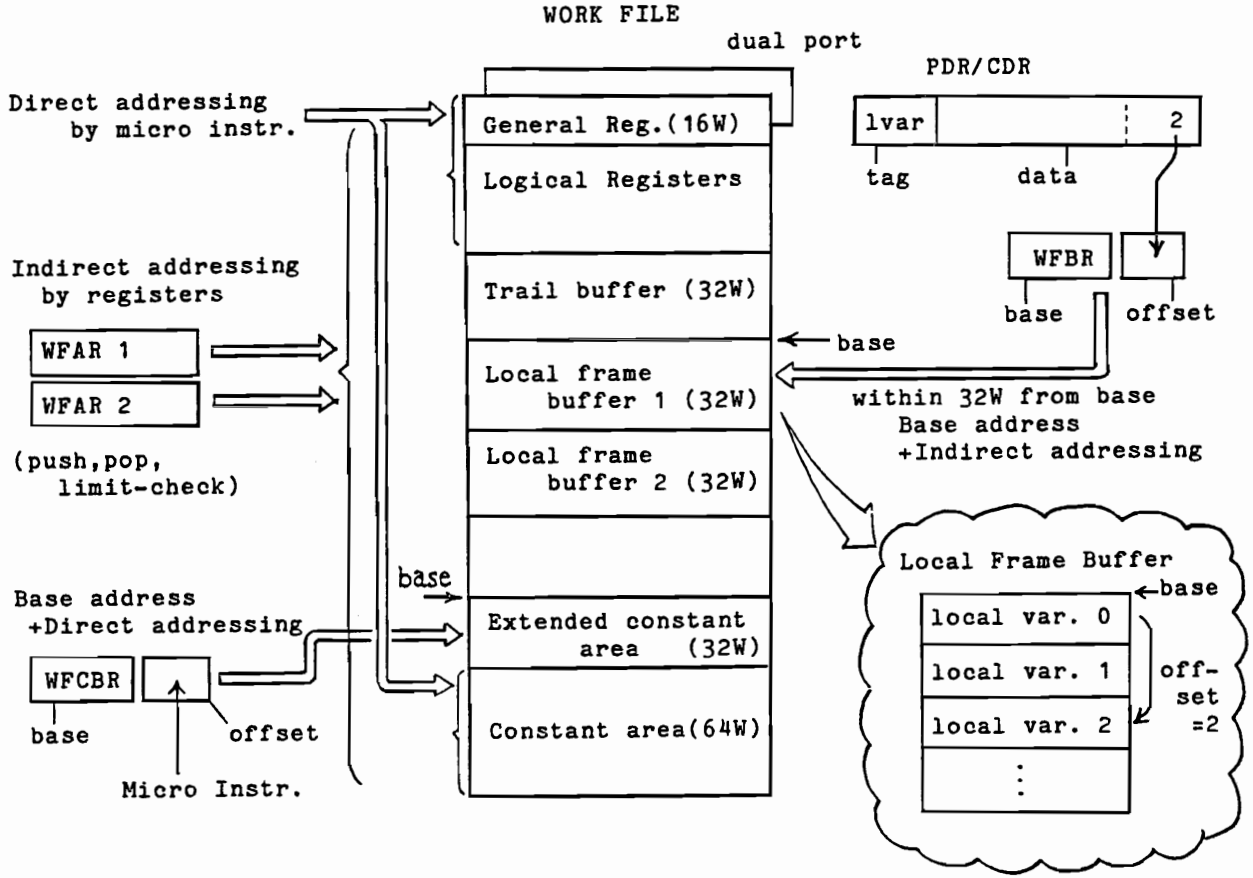

図4．8　ワークファイルのアドレシングモードの概念図

(3) PDR とCDR によるベース相対指定

(4) マイクロ命令によるベース相対指定

(1) はWFの先頭と最後の64語をマイクロプログラムから直接指定するもので、先頭部はジェネラルレジスタとKL0 の実行環境のポインタ類の格納に、最後部は32ビット長の定数格納に使用している。

(2) は 2本の間接レジスタ（WFAR1 、WFAR2 ）による間接指定で、間接後自動増加、減少の機能と32語、 256語の境界チェック機能を持ち、主にLocal frame bufferとTrail bufferのアクセスに用いる。

(3) はPDR またはCDR 中の５ビットの値をWFBRの保持するベースアドレスと結合して、Local frame bufferの中を指すのに使用する。

(4) はWFCBR の保持するベースアドレスとマイクロ命令から供給される直接アドレスを結合して拡張定数領域を指すものである。

この中で特に解釈実行処理の実行時における最適化を意図して設けたのがLocal frame buffer(LFB) と呼ぶ部分であり、もっとも最近に呼ばれた述語のローカル変数フレームを一時的に保持するところである。テイルリカージョンの最適化[Warren 80] が可能なときにはこの中身をローカルスタックに掃き出すことをせず、２面あるLFB を交互に使用してメモリアクセスの頻度を減少させている。特に前述の(3) の機能は、メモリからPDR またはCDR に読み出された変数番号を使ってLFB 上の変数を直接指定し、ステップ数の減少を図っている。

４．２．４．３　タグ操作とタグディスパッチ回路

タグに関する機能の１つめは演算部のタグ操作である。データの演算部にはソースバス 1のタグをそのまま使うかタグ定数値にそっくり置きかえるかする。

またタグ部分だけを取り出して演算し書き戻すことができ、ガーベジコレクション用のマークビットの操作などに用いている。同ビットのテストによる条件分岐や、演算系と独立にソースバス上のタグ値と定数値の比較結果による条件分岐を行なう機能も備えている。

タグ値により多方向分岐する機能をタグディスパッチと呼んでいる。制御部にはタグ用の12面のディスパッチメモリと呼ぶ変換テーブルがあり、PDR またはCDR のタグの下位 6ビットから、14ビット以内の多方向分岐パターンを生成し、マイクロ命令で指定された飛び先基底番地と論理ORを取って飛び先とする。ファームウェアインタプリタや組込述語の内部で12とおりまでの異なる多方向分岐パターンを利用することができ、ファームウェアのステップ数の短縮に効果が期待される。

４．２．４．４　キャッシュメモリとメモリインタフェースレジスタ

複数のスタックを高速にアクセスする機能とメモリアクセスを高速化する機能を合わせて実現するために、スタック向きの機能を持つキャッシュメモリを導入し、同時にインタプリタからの使い方を考慮しメモリインタフェースのアドレスレジスタとデータレジスタを 2組ずつ用意した。

キャッシュメモリは 2セットのセットアンシアティブ方式を採用し、合計で40ビット×8k語の容量を持つ。入替えの単位となるブロックのサイズは 4語であり、主記憶との間はダイナミックメモリのニブルモード機能を利用した語直列方式のブロック転送により、インタリーブ方式をとらずに高速化した。入替えにはLRU(Least Recently Used)方式を、書込み方式としてはストアイン方式を採用した。ストアイン方式とは、キャッシュメモリへの書込み時に主記憶への書込みを同時には行なわず、当該ブロックのミスヒットによる内容の入替え

時にはじめて主記憶への書込みを行なうものである。ストアイン方式はストアスルー方式に比べて、スタックへのプッシュとポップが頻繁なときの効率向上が期待され、スタックアクセス向き方式の１つである。また本キャッシュはロジカルアドレスでアクセスする方式をとっており、ヒット時のアドレス変換が不要な分高速化されている。アクセス時間は読出し、書込みともヒット時には 1マイクロサイクルであり、読出しのミスヒット時には 4マイクロサイクルである。

マイクロ命令からの主なキャッシュオーダには、read、write の他にwrite-stack を用意した。これはスタックの先頭部分への連続的なプッシュを行なう場合に用い、ミスヒットを起こしても主記憶からの値の読込みを行なわないことにより高速化を図ったものである。

メモリインタフェースレジスタには、アドレスレジスタとしてPLARとCLARが、データレジスタとしてPDR とCDR があり、任意の組合せで利用できる。P とCはParentとCurrent の意味でありLAR とはロジカルアドレスレジスタの意味である。LAR とDRを指定してキャッシュオーダを発行すると、指定したLAR の中身のアドレスに対してキャッシュアクセスが実行される。オーダ発行後は、次のread、write オーダかキャッシュとの同期オーダが出されるまでCPU とキャッシュは独立して動き、ミスヒットによるCPU の待ちを少なくするようなコーディングが可能である。またLAR には自動増加の機能を持たせ、メモリの連続領域アクセス時のステップ短縮を図っている。

parentとcurrent とは、ユニフィケーション時の呼出し元（親）と呼出先（現用）の変数領域アクセス用に各々利用するという意味あいで名付けたものである。

４．２．４．５　マイクロ命令と分岐機能

マイクロ命令のビット幅は64ビット有り、ハードウェアの各部を並列的に制御できる。命令タイプは３種類あるが、64ビット中の約３分の２が、演算対象の２つのソースレジスタ指定、デスティネーション指定、フィールド抽出とバイトローテイト指定、キャッシュオーダ、フラグ操作指定などの各タイプに共通のフィールドであり、全て並列的に指定できる。ALU 演算も全タイプで使えるが演算のバリエーションに多少の差がある。タイプによる主な違いは分岐命令であり、相対アドレスによる条件分岐をするタイプ、絶対アドレスによる無条件分岐をするタイプ、JR（ジャンプレジスタ）による間接分岐のみを許すが入出力バス制御やタグ即値の扱えるタイプに分かれる。１つめのタイプには他に、すでに述べたタグディスパッチ、組込述語引数部の 3ビットタグによる8-way 分岐、組込述語のオペレーションコードによるOPコードディスパッチ、マイクロ命令レベルのコールとリターンがあり、戻りアドレスを記憶する1k語のスタックを用意している。条件分岐用の条件としては、

(1) 演算系のフラグ10種

(2) マイクロ命令で個別セットできるフラグ

(3) タグのビットテストや即値との一致比較のフラグ

(4) 割込要求や入出力バス状態テストのためのフラグ

(5) JRをカウンタとして使用したときのゼロ検出フラグ

など全部で64種を用意し、インタプリタおよび組込述語のステップ数短縮を図っている。

４．２．５　インタプリタとファームウェア

KLO の実行メカニズムは、いくつかの実行制御機能[Takagi 83] が追加されている他は基本的にDEC-10 Prolog [Bowen 81][Warren 77] と同じ方法を用いており、構造体の表現もStructure sharing 法[Warren 77] を用いている。但しスタックにはローカル、グローバル、コントロール、トレールの４本を用い、主にクローズ内ORの効率化の為にローカルスタックとコントロールスタックを分離している。ローカルスタックにはローカル変数領域が、グローバルスタックには構造体中に現れる変数の領域がとられ、トレールスタックにはバックトラック時に変数を未束縛状態に戻す為のアドレス情報が置かれる。またコントロールスタックにはプログラムの実行制御にかかわる次の情報を格納している。

(1) 自クローズを含むプロシージャ（１つの述語名で呼ばれるクローズの集合）のベースアドレス
(2) ローカル変数領域のベースアドレス
(3) グローバルスタック上の変数領域のベースアドレス
(4) 自クローズの実行レベル番号（拡張制御機能用）
(5) トレールスタックの格納開始アドレス
(6) 呼出元のコントロールフレームのベースアドレス
(7) リターン先コントロールフレームのベースアドレス（テイルリカージョンの最適化時のみ(6) と異なる）
(8) リターン先で実行を再開すべきボディゴールアドレス
(9) バックトラック時に実行すべきボディゴールアドレス
(10)バックトラック先のコントロールフレームベースアドレス

実行中のクローズに関するこれらの情報はワークファイル中のロジカルレジ

スタの領域に置かれているが、ボディゴール中の最後のゴールを除くユーザ述語呼出し時には呼び出し元コントロールフレームとしてスタックに掃き出され、別解を持つ述語の実行開始時点ではバックトラックコントロールフレームとしてスタックに置かれる。

インタプリタは、ワークファイルを活用したテイルリカージョンの最適化、タグディスパッチによるユニフィケーションステップの短縮、多様な条件分岐によるメモリ割当て要求や割込み要求テスト等の簡単化、スタック向きキャッシュを活用したスタックへの退避、回復などにより処理の高速化を図っている。

ファームウェアは、基本制御部とユニファイヤからなるインタプリタ核部の他、図 4.9に示すように組込述語部とOSインタフェース部から成り、それらの中身はさらに細かくモジュール化を行なっている。ここではモジュール間の関係がサブルーチン関係でなくネットワーク状を成すものが多いため、ファームウェアレベルでの再帰呼出しは避けることにした。実用面での機能サポートは、ページ割当、割込処理とプロセス切替え、エラー処理、例外処理などがOSインタフェース部に集中している。図にはないが他にデバッグサポートのファームウェアが有り、またガーベジコレクタは組込述語に含めている。

４．３　試作と評価

４．３．１　ハードウェアとファームウェアの製作

本システムのハードウェアは、プリント板、カードケージ、筐体の３つの実

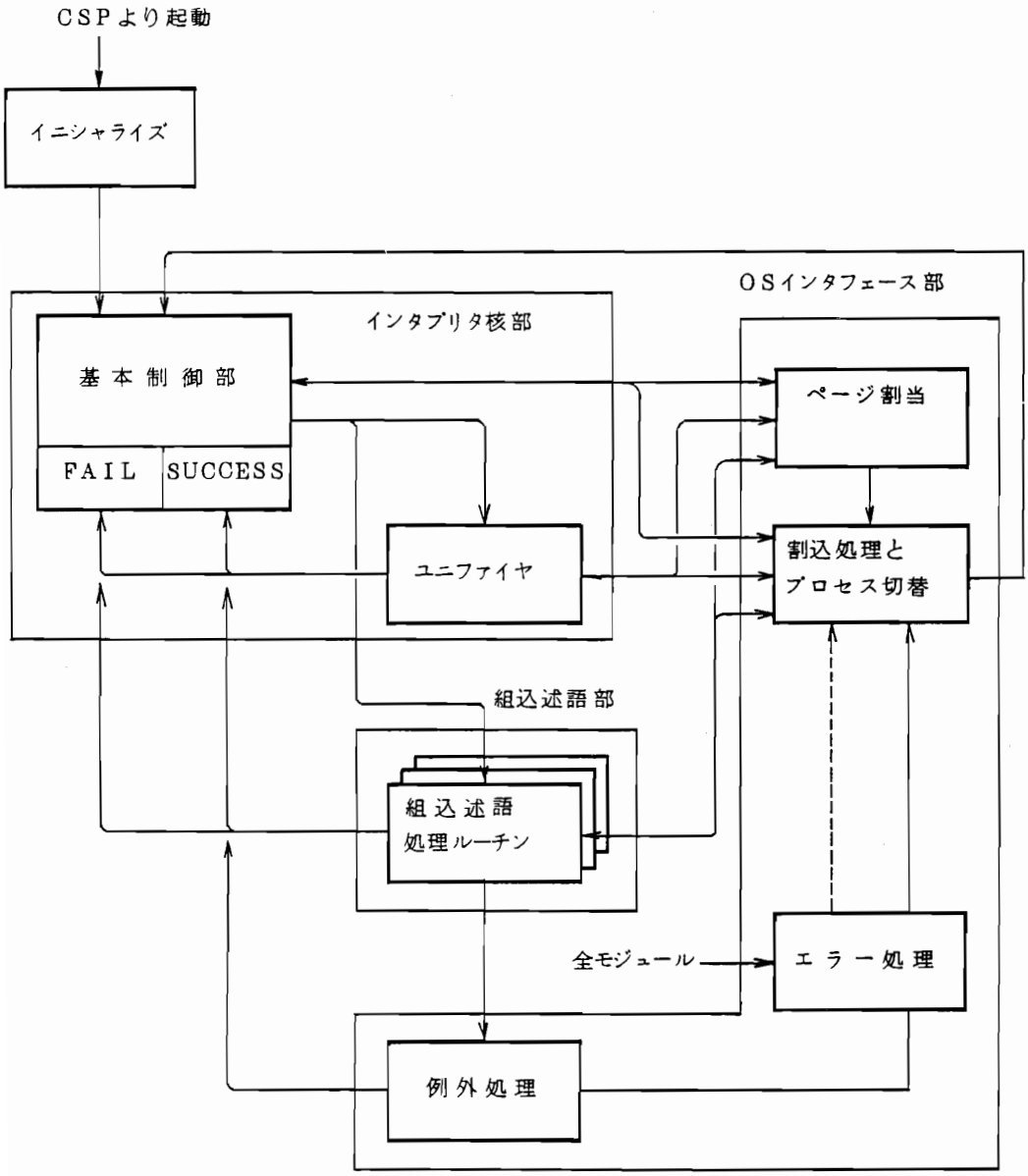

図４．９　ファームウェアのモジュール構成

装レベルから成り、たて 160㎝、横65㎝、奥行き90㎝の筐体に 4台のカードケージと電源、 2台のフレキシブルディスクと 2台の固定ディスク装置を実装した。カードケージは上から、メモリボード用（最大16枚実装）、CPU 用、入出力制御用１、入出力制御用２（IEEE796 仕様）である。CPU は、約30㎝四方でICが約 160個実装できる四層プリント基板12枚から成り、入出力制御装置には同サイズのプリント板10枚を使用している。

使用素子は高速ショットキーTTL のLSI 、MSI を中心に、CPU 部には他に高速のスタティックRAM を、メモリ部には256kビットのダイナミックRAM を使用して、マイクロ命令サイクルタイム200nS を実現している。

ファームウェアの製作に当たっては、開発量の多いことから書き易さとデバッグのし易さに注意を払い、C 言語ライクのシンタクスでマイクロ命令を記述できるマイクロ命令アセンブラと、レジスタ転送レベルのハードウェアシミュレータをそれぞれDEC-10 Prolog とPASCALで記述し使用した。ファームウェアの全ステップ数は約14k 語であり、そのうちわけは、

(1) インタプリタ核部　約 2.4k

　　（その内基本制御部約1k、ユニファイヤ約 1.4k ）

(2) 組込述語（ 158種）　約10k

　　（その内ガーベジコレクタ約 1k ）

(3) OSインタフェース部　約 0.9k

(4) デバッグサポート　約1k

である。

４．３．２　実行時間の測定と性能評価

４．３．２．１　プログラムの実行時間

評価用のプログラムを用いてPSI の実行時間とDEC-2060上のDEC-10 Prologの実行時間を測定し、両者の比とともに示したのが表 4.1である。なおDEC-10 Prologはモード宣言とfast code 指定をしてコンパイルしたものをstatisticsを用いて時間計測し数回の平均をとっている。PSI についてはCPU 内蔵の精度1msec のタイマーを用いた。

評価用のプログラムは、表中の(1) から(10)が第１回Prologコンテスト[Okuno 85]の課題の一部であり、いずれもリスト操作を多く含む小規模のプログラムである。それに対し(11)から(19)はより規模の大きい実際的なプログラムであり、BUP とLCP が自然言語処理用の各々方式の異なるパーサーであり、harmonizerは音楽知識によりメロディーにハーモニーを付加する一種の専門家システムである。(11)以下のプログラムは、次節のハードウェア評価でも使用している。

表からPSI の性能は、当初目標のDEC-10 Prolog と同等を達成していることが分かる。プログラムによりDEC とPSI の優位性が逆転するのが注目されるが、その傾向としては(1) のnreverseのように取扱うデータが単純でコンパイラによる最適化が効果的なプログラムではDEC-10 Prolog が高速なのに対し、(11)から(14)のように大きな構造体データのユニフィケーションを含み頻繁なバックトラックをともなうプログラムではPSI が速くなっている。その原因としては、PSI の処理系がDEC に比べより多くの実行管理情報をスタックに積み単純なプログラムでは基本処理のオーバーヘッドが目立つためと考えられるが、逆に複雑なユニフィケーションが出現するとすべてマイクロプログラムで処理す

表４．１　評価用プログラムの実行時間

programs	PSI(msec)	DEC(msec)	DEC/PSI
(1) nreverse (30)	14.6	9.48	0.65
(2) quick sort (50)	16.1	14.6	0.91
(3) tree traversing	52.2	61.1	1.17
(4) lisp (tarai3)	4050	4360	1.08
(5) lisp (fib10)	375	402	1.07
(6) lisp (nreverse)	174	194	1.11
(7) 8 queens (1)	105	97.5	0.93
(8) 8 queens (all)	1710	1580	0.92
(9) reverse function	38.9	41.7	1.06
(10) slow reverse (4)	6.13	5.42	0.88
(11) BUP-1	43	52	1.21
(12) BUP-2	139	194	1.40
(13) BUP-3	309	424	1.37
(14) harmonizer-1	657	1040	1.58
(15) harmonizer-2	1879	2670	1.42
(16) harmonizer-3	24119	31390	1.30
(17) LCP-1	379	295	0.78
(18) LCP-2	1387	1071	0.77
(19) LCP-3	2130	1656	0.78

表４．２　ファームウェアインタプリタ各部の実行ステップ比率

program	control	unify	trail	get_arg	cut	built	others
(1) window	31.1%	17.1%	2.0%	13.6%	10.0%	20.0%	6.2%
(2) 8 puzzle	27.5	11.0	7.5	22.7	0	30.7	0.6
(3) BUP	26.3	43.0	4.7	5.2	5.6	12.4	2.8
(4) harmonizer	25.5	47.0	5.4	7.0	4.1	8.3	2.7

るPSI が相対的に高速になるものと考えられる。

全体的に見てPSI は実用レベルの応用プログラムに強い傾向を持つといえる。ただLCP だけは構造体データを（ある形態で）扱うにもかかわらずDEC が速い。これはLCP の作者がDEC-10 Prolog 処理系の製作者の１人であるF.Pereira 氏で処理系の長所欠点を知りぬいたコーディングをしているためでありプログラムの書き方により処理効率が異なることを示すおもしろい例ともいえる。

４．３．２．２　インタプリタの動特性

実用規模の応用プログラムにおけるPSI の総合性能を評価するため、評価用プログラムを実行したときのインタプリタ各機能の動特性を測定した。表 4.2 は各機能モジュールの実行ステップ数比率を示したものである。評価用プログラムとしてはPSI のオペレーティングシステムの一部をなすwindowシステムと8 puzzleというゲームプログラムを追加した。

述語の呼出し回数は、表中にはないがユーザ述語より組込述語が多くwindowで80% 、よりProlog的なプログラムであるBUP で60% となった。しかしながら組込述語(built) の実行ステップ数の比率を表より見ると、各々20.0% 、12.4% となり回数の比率に比べてずっと小さく、むしろ実行制御(control) に多くの時間を要していることが分かる。これはすでに述べたようにKL0 では実行管理情報が多く、処理にステップ数を要することの影響と考えられる。また組込述語のための引数読出し(get_arg)にも時間がかかっていることが読取られる。これらは、処理系改良時の対象として考慮すべきものであろう。一方unify を見るとBUP とharmonizerで著しく比率が高く、これは構造体の複雑なユニフィケーションの多さを裏付けるデータである。ユニフィケーション単体の性能は、PSI はDEC に比べて悪くなく、unify 部分のスップ数短縮にはコン

パイラでユニフィケーション自体の回数を減らすような最適化が必要と思われる。

４．３．３　動特性の測定とハードウェア評価

４．３．３．１　動特性の測定

本節では主にハードウェアの改良の要否を判定する観点から、評価用プログラム実行時のハードウェアの動特性を測定し評価を加える。

評価対象は、キャッシュメモリ、ワークファイル、分岐機能であり、Prolog系言語向き機能のうちの

(1) 複数のスタックを高速にサポートする機能

(2) インタプリタの最適化のための大容量で高機能のレジスタファイル

(3) データタイプの判定による多方向分岐と強力な条件分岐機能

(4) 高速なメモリアクセス機能

について、各々のハードウェアの効果を評価し、改良の要否、あるいは可能性について検討を加える。

評価用プログラムには前節と同じものを使用し、またデータ収集と評価のためにいくつかのツールを作成した。データ収集用にはコンソールプロセッサ上にCOLLECT と呼ぶ簡単なインタプリタシステムを実装し、ステップ実行やブレークポイントまでの実行をくり返し行なわせ、CPU 停止するたびにレジスタやメモリ内容をフレキシブルディスクに貯めこむようにした。ワークファイルと分岐回路の評価には、MAP と呼ぶマイクロ命令パターンの解析ツールを用意し、COLLECT によるマイクロ命令アドレスのトレース結果を利用して、特定のマイクロ命令フィールドの特定パターンの出現度数をカウントした。キャッシュメ

モリの動特性解析には、PMMSと呼ぶキャッシュメモリシミュレータを使用し、同じくCOLLECT により収集したアドレスとキャッシュコマンドパターンからキャッシュのヒット率などを求めた。

４．３．３．２　キャッシュメモリの評価

スタックアクセスの高速化とヒープ領域アクセスの高速化の効果を評価するため、キャッシュメモリのアクセス頻度とキャッシュヒット率に関するデータを収集した。

表 4.3にキャッシュコマンド別の出現頻度を示す。全マイクロ命令ステップ中16% から23.1% はキャッシュコマンドを含んでおり、マイクロ命令 5ステップに対し 1ステップ程度の割合でメモリ参照要求のあることが分かる。readとwrite 系コマンドの比率は約 3対 1でreadコマンドが多くなっている。またwrite 系コマンド中の 5割から 7割半が write_stack コマンドであり、スタック用に導入したコマンドがよく利用されている。

表 4.4はエリア別のアクセス頻度である。はじめに、heapのアクセス頻度と 4本のスタックのアクセス頻度に分けて見ると、ヒープ領域へのアクセスが全体の 3割から 5割半を占めるのが分かる。これによりキャッシュのヒット率が高ければ、ヒープとスタックの両方への高速アクセスに効果を発揮しているといえる。ヒープ領域へのアクセスは基本的には命令コードのフェッチであるがwindowのプログラムでは他にオブジェクト中の変数セルが置かれており、その分アクセス頻度が上がっている。またプログラムにより命令のアクセス頻度が異なるのが分かる。

グローバル、ローカル、コントロールの各スタックへのアクセス頻度もプログラムに依存している。構造体データが多いとグローバルスタックが、引数中

表４．３　キャッシュコマンドの出現頻度

programs	read	write_ stack	write	write total	total
(1) window-1	15.2 %	3.5 %	1.2 %	4.7 %	19.9 %
(2) window-2	15.2	3.0	1.1	4.1	19.7
(3) window-3	17.6	3.9	1.4	5.3	22.8
(4) 8 puzzle	9.9	3.2	2.8	6.1	16.0
(5) BUP	15.6	3.5	2.2	5.7	21.3
(6) harmonizer	15.3	4.6	2.2	6.8	22.1
(7) LCP	17.0	3.9	2.2	6.1	23.1

表４．４　エリア別アクセス頻度

programs	heap	global stack	local stack	control stack	trail stack
(1) window-1	49.6 %	4.6 %	16.5 %	26.7 %	2.6 %
(2) window-2	56.6	4.4	12.7	26.3	0.1
(3) window-3	52.7	6.2	12.1	28.2	0.8
(4) 8 puzzle	31.3	14.3	33.9	14.1	6.4
(5) BUP	39.0	29.9	17.3	12.0	1.8
(6) harmonizer	35.2	17.7	30.3	12.8	3.8
(7) LCP	44.7	22.3	14.1	17.4	1.4

の（構造体に含まれない）変数が多いとローカルスタックが、引数個数に比べ述語呼出しが多いとコントロールスタックがそれぞれ頻繁にアクセスされている。トレールスタックへのアクセスは最大でも 6.4% と低くなっている。

表 4.5はエリア別のキャッシュヒット率である。平均のヒット率は、windowで90% のものがあるが、他はいずれも96% 以上で高い値を示している。特に実用規模の応用プログラムでかつバックトラックやユニフィケーションを十分に利用している(5) から(7) のプログラムで高いヒット率を得ていることから、Prolog系の言語でもメモリアクセスのローカリティは高く、PSI で採用した規模のキャッシュメモリが十分に有効であるということができる。なおwindow-1、-2のヒット率が低いのは、実行中にプロセスッスイッチが起こっていることと、多くのESP のクラスをまたがった呼びだしのために命令コードのローカリティが低下しているためと考えられ、オブジェクト指向によるプログラミングがキャッシュのヒット率を低下させるか否かについては、別途検討の余地のあることを示している。

次にキャッシュメモリの容量とセット数に関する評価結果を示す。これらは、windowプログラムのトレースデータを用いて、キャッシュシミュレータにより測定したものである。

図 4.10 は、キャッシュメモリ容量を8wから8kw まで変化させたときの性能向上率である。但し、

性能向上率＝｛（Tnc/Tc）-1｝× 100

Tnc ：キャッシュメモリが無いときの実行時間

Tc ：キャッシュメモリがあるときの実行時間

とした。またセット数はすべて 2である。図から性能向上率が512w付近で飽和しつつあることが読み取られ、PSI のキャッシュメモリ容量8kw は、幾分削減

表４．５　エリア別キャッシュヒット率

programs	heap	global stack	local stack	control stack	trail stack	total
(1) window-1	94.1 %	92.8 %	98.9 %	99.4 %	99.6 %	96.4 %
(2) window-2	87.2	90.0	98.5	99.3	95.2	91.9
(3) window-3	84.5	92.8	97.4	98.6	98.7	90.7
(4) 8 puzzle	99.2	99.4	99.6	99.2	97.7	99.3
(5) BUP	98.2	96.8	99.0	98.2	99.7	98.0
(6) harmonizer	97.5	98.4	99.4	98.2	97.9	98.4
(7) LCP	96.6	93.8	99.2	99.1	98.6	96.2

表４．６　ワークファイル各機能の利用頻度

Type	Source 1	Source 2	Destination
(1) WF00-0F	12.2/ 6.9	100/29.1	33.0/12.1
(2) WF10-3F	58.5/33.0	---	63.6/23.3
(3) Constant	23.0/13.0	---	---
(4) @PDR/CDR	1.3/ 0.8	---	0.3/ 0.1
(5) @WFAR1	4.6/ 2.6	---	2.8/ 1.0
(6) @WFAR2	0.07/0.04	---	0.3/ 0.1
(7) @WFCBR	0.3/ 0.2	---	0.0/ 0.0
Total	100/56.4	100/29.1	100/36.6

a/b----a: WFアクセス命令中の頻度

b: 全命令中の頻度

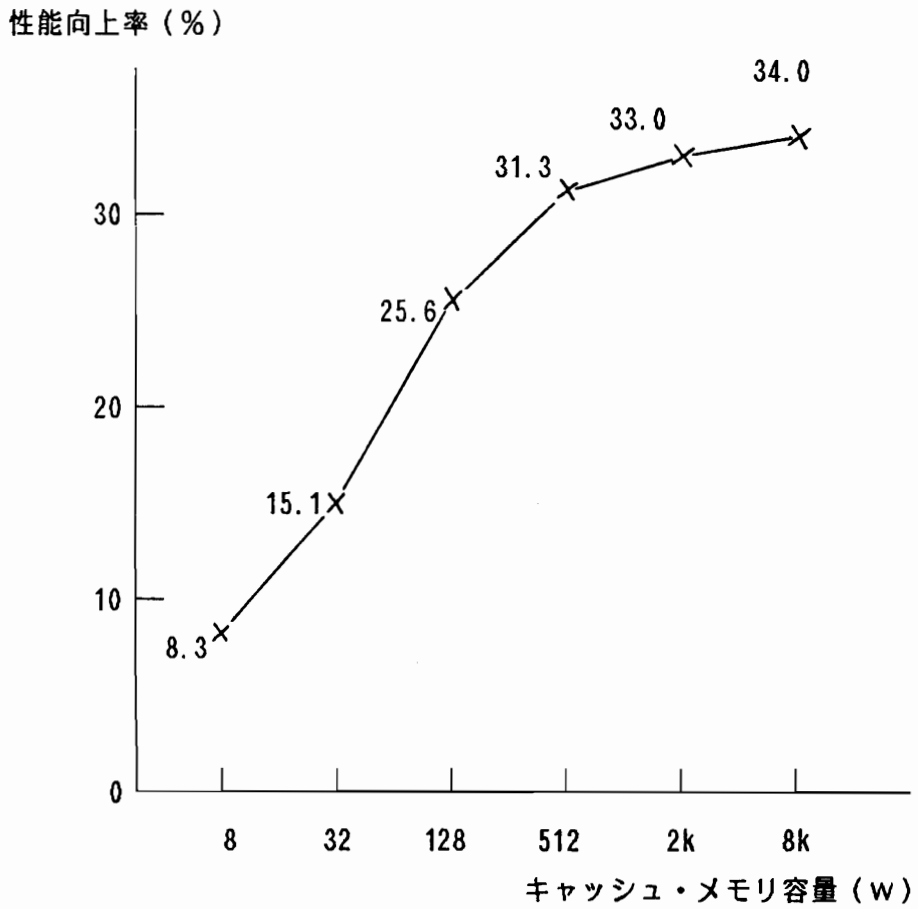

図４．１０　キャッシュメモリ容量と性能

できる可能性のあることを示している。

一方実装面からはセット数を 1にし容量を半減した方がより効果的である。そこで4kw 2 セットの場合と4kw 1 セットの場合の性能向上率の差を求めてみたところ、window、8 puzzle、BUP のいずれにおいても 1セットの方が約3%低いだけであった。これは設計・実装コストと性能の比から考えて利用可能な値ということができよう。

またストアイン方式の評価については、ストアスルー方式との性能向上率の差を求めた。その結果ストアイン方式の方が約8%高い値を示し、有効性を確認することができた。

４．３．３．３ ワークファイルの評価

マイクロ命令パターン解析ツールにより、評価用プログラムを実行したときのワークファイル(WF)各機能の利用頻度を求めた。その中からBUP に関する測定結果を表 4.6に示す。他のプログラムについても同様の傾向が見られる。

WFのアクセス頻度は表中のtotal の欄より、Source 1指定が56.4% 、Source 2指定が29.1% 、Destination 指定が36.6% である。またWFを含むレジスタアクセスの命令だけを集計した場合には、WFのアクセス頻度は50から80% となり、利用頻度の高さが確認された。

直接アドレス指定を行なうのは、表中の(1) から(3) であるが、これらのアドレッシングモードの利用率がwfアクセス全体の90% を超えていることが分かる。直接アドレス指定を可能とするため、レジスタアクセス用のマイクロ命令フィールドをマイクロ命令語長全体の 3分の 1も費しているが、この効果が十分であることが確認された。

次にベース相対指定(4) 、(7) と間接指定(5) 、(6) の評価を行なう。(4)

のPDR とCDR によるベース相対指定は期待していたより利用頻度が低く、もっとも高いwindowでも 3.3/1.7という値であった。PDR/CDR 以外に組込述語に付随するコンパクト引数からのベース相対指定が扱えれば、利用率はもう少し高まると考えられる。(5) のWFAR1 による間接指定は(4) とともにLocal frame bufferのアクセスに用いられるが、 9割以上が間接アクセス時の自動増加機能を利用しており、その有効性が確かめられた。(6) はTrail bufferのアクセスに、(7) は汎用的に利用されるがいずれも利用率が低く、トレールスタックのバッファリングについては再検討の余地があるといえる。

WFへのアクセスは全体の99% 以上が、直接アドレス指定域の128wとLocal frame buffer 2面の64w へのアクセスであり、このことから容量を1kw から削減しても性能への影響は少ないということができる。

４．３．３．４　分岐機能の評価

マイクロ命令パターン解析ツールを用いて分岐命令の出現頻度を調べた結果を表 4.7に示す。operation 欄には、マイクロ命令タイプ別の分岐命令の種類を示してある。すべての分岐命令を合わせた出現頻度は、100%からno operation の比率を差し引いて求められ、約77% から83% と非常に高い値を示している。Destination のno operation比率が30% 前後であることから、全マイクロ命令ステップ中の約 3割がデータ操作をともなわない分岐、約 5割がデータ操作を行ないながらの分岐ということになる

表中の(2)(3)が64種ある条件分岐を合わせたものであり、(4) がタグ即値との比較による条件分岐である。これらの合計は、全ステップの35% から39% となり、条件分岐機能の強化の有効性が確められた。

(5) はタグディスパッチ機能、(6) は組込述語のコンパクトオペランドに付

表４．７　分岐命令の種類別出現頻度

operation	BUP	window	8 puzzle
Type1	73.0 %	73.5 %	72.8 %
(1) no operation	7.2	6.7	4.8
(2) if (cond) then	16.0	16.5	12.1
(3) if (not(cond)) then	19.2	17.0	20.3
(4) if tag(src2) then	2.7	5.2	3.1
(5) case (tag(n,P/CDR))	10.9	8.6	9.1
(6) case (irn)	2.8	4.6	4.9
(7) case (ir_opcode)	0.5	1.4	1.5
(8) goto	3.7	1.4	2.7
(9) gosub	4.0	5.7	6.5
(10) return	3.8	5.4	6.5
(11) load_jr	0.8	0.4	0.7
(12) goto @jr	1.4	0.6	0.7
Type2	20.5 %	19.4 %	22.9 %
(13) no operation	9.6	7.8	7.7
(14) goto	10.9	11.7	15.2
Type3	6.5 %	7.1 %	4.2 %
(15) no operation	6.5	7.0	4.2
(16) goto @jr	0.0	0.04	0.05

随するタグを用いた多方向分岐機能であり、これらの合計の比率は約13から14% となった。すなわち 7から 8ステップに 1度はタグによる多方向分岐を行なっていることになり、これらの機能も非常に有効であることが示された。

(12)と(16)のJRによる間接分岐は頻度が低いが、JRはむしろループカウンタとして使用されることが多く、その頻度を求めると全ステップ中の約 9から12% もあり、専用のループカウンタを置くように再設計する方が望ましいと考えられる。

４．４　LISPマシンとの比較による考察

４．４．１　アーキテクチャに関する考察

本節では、前章で取りあげたLISPマシンシステムとの比較をとおし、LISP向きアーキテクチャとProlog向きアーキテクチャの共通点、相異点を明確にする。またシステムの実用化の観点から取り入れた機能が、他の部分の設計にどのような影響を与えたかについて考察する。

４．４．１．１　LISP向きアーキテクチャとProlog向きアーキテクチャ

LISP言語の実行を高速化するアーキテクチャとして効果の大きかったものは、

a.　高速のハードウェアスタック

b.　データ型判定用のマッピングメモリ

c.　多方向分岐と多様な条件分岐機能

d. CAR 部とCDR 部を同時に読出せるメモリ回路

e. 水平形に近いマイクロ命令

f. ガーベジコレクション用のビットテーブル

であった。一方Prolog系言語向きのアーキテクチャとして効果が大きかったのは、

① スタックアクセスとメモリアクセスの両方を高速化するキャッシュメモリと２組のメモリインタフェースレジスタ

② タグ値にもとづく多方向分岐

③ 多様な条件分岐機能

④ 高機能のレジスタファイル

⑤ 水平形に近いマイクロ命令

であった。これらを比較すると、b.は②と機能的に同じものであるし、e.は逐次型推論マシンではタグ部分に専用ビットを持っており、④を除いて両方のアーキテクチャはきわめてよく対応がつく。そこでLISPとPrologに共通する高速実行向きアーキテクチャをまとめると、

(1) スタックアクセスの高速化機能

(2) ガーベジコレクション用のビットを含むタグアーキテクチャとタグにもとづく多方向分岐機能

(3) 多様な条件分岐機能

(4) スタックを除くメモリアクセスの高速化機能

(5) マイクロ命令レベルでの並列性が生かせるマイクロ命令仕様

の５つとなる。またLISP向きとProlog向きで異なるのは、

(1) LISPではハードウェアスタックが効果的であるが、Prologではスタック上でのデータアクセスの分散とスタック本数の多さからキャッシュメモ

リの方が適すること。

(2) Prologでは実行管理情報がLISPより多いことからレジスタファイルを多く用いる傾向にあること。またフレームバッファでもレジスタファイルを利用すること。

の２点といえる。

４．４．１．２　実用化のためのアーキテクチャに関する考察

人工知能向き言語用の専用マシンに対する要求のうち実用性に関するものは、メモリの大容量化、パーソナル化と対話型入出力装置の強化への要求であることは第２章で述べた。さらに通常利用する機能以外に、RAS （信頼性、保守性、可用性）に関する要求がある。これらを実現するためのアーキテクチャとして、

(1) ページングによる実メモリ管理をサポートするハードウェアと高速なメモリ割当ておよび効率的再配置を行なうファームウェア

(2) スタンドアロンマシンのオペレーティングシステムを記述するため、マルチプロセスをサポートするファームウェアと独立な論理アドレス空間を提供するハードウェア

(3) 入出力バス制御および多重優先レベル割込みのハードウェアとそれらを制御するファームウェア

(4) 主記憶の 1ビットエラー訂正とCPU のデータパリティ回路、および保守、立上げ、デバッグ、エラーログをサポートするコンソールプロセッサのハードウェアとソフトウェア

が必要となった。これらは単にハードウェアの追加となるものと、ハードウェアを追加するために実行速度の低下を招くものがある。ファームウェアについても同様のことがいえる。

実験マシンを実用マシン向きに再設計する場合や両者の速度を比較する場合には、ハードウェアの増加と実行速度の低下の度合を知っておくことがきわめて重要となる。以下ではハードウェア、ファームウェアについて単なる増加で済むものと速度の低下を招くものを分類整理する。

まず増加だけで済むものには、

a. 入出力バスの制御のハードウェア、ファームウェアと割込のハードウェア

b. コンソールプロセッサのハードウェアとソフトウェア

c. メモリの再配置をするファームウェア

d. マルチプロセスをサポートするファームウェア

がある。次にキャッシュメモリがヒットしなかったときだけ実行時間に影響が出るものをまとめると、

e. (1) と(2) にまたがるメモリ管理ハードウェアはアドレス変換に 1サイクル(200nsec) を要する。

f. エラー訂正回路のために主記憶の読出し時間が 1サイクル分長くなっている（ 2サイクルが 3サイクルに）。

となる。最後に常に実行時間の低下を招いているものをまとめると、

g. CPU のデータパリティ回路はパリティのジェネレートとチェックの時間を必要とし、これを付加したためにCPU のサイクルタイムを166nsec から200nsec に設計変更した。

h. メモリの割り当てを行なうファームウェアは、インタプリタ中でスタックやヒープを伸ばすときに常にページ境界を越えないかどうかの判定を行っており、数％の実行オーバヘッドを生じている。

i. 割込み処理ファームウェアは命令の切れめ毎に 1サイクルをかけて割込

の有無をチェックしている。

となる。但しf.は時間短縮の可能性があり、h.とi.は方式の変更やハードウェアサポートにより軽減できる可能性もある。

これから総合して、実用化のための諸機能の導入によりCPU の実行時間が遅くなっている割合は20% 余りであると考えられる。

４．４．２　ハードウェアコストに関する考察

ハードウェアのサイズについて考察する。LISPマシンのCPU 部分（メモリを除く）のIC個数は約 400個であるのに対し、逐次型推論マシンのCPU 部分のIC個数は約1800個である。この差は主に次の点によると考えられる。

(1) LISPマシンが16ビットマシンであるのに対し、逐次型推論マシンはタグも含めると40ビットマシンである。

(2) LISPマシンではビットスライスのRALUと 1チップのマイクロプログラムシーケンサを使用しているが、逐次型推論マシンでは主にMSI を使用している。

(3) 逐次型推論マシンでは前節に述べた実用化のための多くの機能を付加しており、WCS の容量も大きい。

(4) 逐次型推論マシンはキャッシュメモリ部分とCPU 内のシフト回路や分岐回路の一部がLISPマシンに比べて重い（ワークファイルとハードウェアスタックの規模には差がない）。

これらの中でIC個数の大きな差を生み出しているのは(1) と(3) と考えられる。(1) は容易に推測できるため、(3) によるハードウェアサイズの増加について検討する。逐次型推論マシンのボード構成は12枚でほぼ次のようになって

いる。

a. 演算部分　3枚

b. シーケンサ部分　3枚

c. キャッシュメモリ

d. キャッシュコントロール（デレクトリ含む）

e. アドレス変換

f. 主記憶エラー訂正とメモリインタフェース

g. クロックとシステムコントロール

h. 入出力バス制御とコンソールプロセッサ

この中から実用化のために付加した機能を取り出してみると、a.とb.の中の1枚ずつ、およびe.、f.、h の 5枚となる。a.とb.はもともと 2枚ずつで済んだものが、パリティ回路の付加により 3枚ずつに増加した経緯がある。

これらにより、逐次型推論マシンの中でマシンの実用化向けに導入したハードウェアのサイズは全体の約42%（12分の 5）もあり、実験マシンに比べて大きなハードウェアサイズの増加となることを示している。

またLISPマシンに比べて機能的に高くなっているキャッシュメモリ部分のハードウェアサイズは全体の17%（12分の 2）と大きく、導入時にはセット数や容量の検討が重要と考えられる。

４．５ 結　言

本章ではPrologを拡張した述語論理型言語用の専用マシンについて、処理の

高速化と実用化の観点からアーキテクチャの検討と設計、試作を行ない、処理性能と採用したハードウェアについて評価を加えた。また実験マシンと比較した場合のアーキテクチャ上の特徴とハードウェアコストについて考察した。

基本方式としてマイクロプログラムによるインタプリタ方式とスタンドアロン方式をとり、Prolog系言語の高速化向きハードウェアとしては、

(1) スタックアクセスとメモリアクセスを高速化し特にスタックアクセス向き機能を持つキャッシュメモリ

(2) タグアーキテクチャとタグ値にもとづく多方向分岐機能

(3) インタプリタの最適化のための高機能レジスタファイル

(4) 多様な条件分岐機能

(5) メモリアクセスの高速化と最適化を図る 2組のメモリインタフェースレジスタ

(6) マイクロ命令レベルでの並列処理

(7) インタプリタと組込述語全体のファームウェア化を可能とする大容量のWCS

を盛りこんだ。また実用化の観点からは、

(1) ページングによる大容量メモリの管理と効率的再配置を行なう機能

(2) マルチプロセスのサポート機能

(3) 入出力バスの制御と割込み機能

(4) RAS （信頼性と保守性）に関する機能

を準備した。CPU 部分は高速の TTL LSI、 MSIを主に使用し、入出力バスにはビットマップディスプレイやマウスなどの対話型入出力装置を準備した。インタプリタは 4本のスタックを用いる実行方式をとり、主に実行制御に関してPrologの機能を拡張した。ファームウェアはモジュール化構成をとり、ユニフ

ァイアの高速化の外に、割込み、プロセス切り替え、メモリ管理などのオペレーティングシステムサポート機能の充実を図った。

完成したシステム上で第１回Prologコンテストの課題プログラムといくつかの応用プログラムを走らせた結果、目標としていたDEC-2060上でのコンパイルコードと同等の実行性能が得られた。注目されるのは最適化のよくかかるコンテストのプログラムではDEC が速いのに対し、動的処理の大きい応用プログラムではPSI の方が速いことであり、ファームウェアインタプリタの 1つの特徴を示している。

また同じプログラム実行時のハードウェアの利用特性を測定した結果、キャッシュメモリのヒット率は応用プログラムで96% であり、Prologに対してキャッシュメモリが十分有効であること、ストアイン方式が効率的なこと、キャッシュメモリの容量が削減可能なことが判明した。ワークファイルについては直接アドレス指定が有効であるがフレームバッファの効果が予想より小さいこと、容量削減が可能であることが示された。また分岐回路についてはタグ値にもとづく多方向分岐、条件分岐とも十分に有効であることが確認された。

LISPマシンとの比較からは、LISPとPrologの高速実行向きアーキテクチャはほとんど同じであり、違いは

(1) LISPではハードウェアスタックが有効であるがPrologではキャッシュメモリの方が適すること。

(2) Prologの方がレジスタファイルをよく使う傾向にあること

の 2点であることが示された。またマシンの実用化のために導入されたハードウェアのサイズは、マシン全体の約40% を占め、そのことによりCPU の処理性能は20% 余り低下していることが示された。

第５章

コンパイル方式の導入と評価

５．１　緒　言

第３章、第４章ではファームウェアインタプリタ方式によるLISPマシンと逐次型推論マシンの試作と評価を行なったが、本章ではこれら２つのシステムに対してコンパイラを試作し処理速度を中心に評価を行なう。

コンパイラの製作に当たっては、オブジェクトコードとなるべき機械語命令としてそれぞれLISP向きおよびProlog系言語向きの中間言語命令を設計し、その中間言語命令をターゲットとして簡単な最適化を行なうコンパイラを設計した。評価用には第２回LISPコンテスト [Takeuchi 78]、第１回Prologコンテスト[Okuno 85]の課題プログラムなどをそれぞれ使用し、実行時間と動特性の測定を行なうとともに第３章の考察で予測されたコンパイル方式導入による速度向上率が大きくないことに対する検証と分析を行なった[Kaneda 81][Nakajima 86] 。またLISPマシンについては直接マイクロコードを生成するコンパイラとそのときの速度向上についての検討を行なった[Kaneda 82] 。

最後にLISPマシンとPrologマシンのコンパイル方式、速度向上率についての比較・考察を行なった。

５．２　LISPマシンへのコンパイル方式の導入と評価

５．２．１　コンパイラの設計

本節ではLISPマシンのために設計した中間言語命令の仕様と、コンパイラの中での最適化方式について述べる。

５．２．１．１　中間言語命令

本システムはマイクロプログラムにより制御されているため、機械語命令を本質的には持たない。そのためコンパイラを構成する場合機械語命令に対応した中間言語命令を設定し、それをオブジェクトコードとして生成する。コンパイルされた関数の実行時にはマイクロプログラムで記述されたエミュレータ（命令のフェッチと実行の制御をする）により中間言語命令が順にフェッチされ実行される。

これら中間言語命令の設定の際、メモリ参照があまり負担にならないこと、命令語長を１語（32ビット）にするなどに留意して命令コードを構成した。

(1) CALL命令：オペランドで示された関数に対し、その属性に応じた実行の予備操作（引数のバインディングとかリターンの前準備）を行ない、関数を呼出す。

(2) PSH 命令：オペランドで示されたデータをスタックの先頭に積む命令であり、オペランドとしては(i) 実行する関数の実引数、(ii)現在の自由変数値、(iii) データそのもの、の３種類指定でき以下の命令も同様。

(3) POP 、STORE 命令：現在のスタックの内容をオペランドで示されるフィ

ールドにコピーする命令でSET 、SETQに対応している。POP はコピー後スタックをポップアップする。

(4) MKF 命令：関数の実行に先だちフレームヘッドを作る命令で、PSH 命令をかねることも可能。

(5) JMP 命令：条件によりラベル先へ分岐するか否か制御する命令でありスタックをポップアップすることも可能である。条件として(i) 無条件分岐、(ii)NIL 分岐、(iii) T 分岐の３種類が指定できる。

(6) RET 命令：現在のスタック先頭値を関数の評価値として呼び出し元へリターンする。あわせてフレームの消滅処理を行なう。

(7) BIND命令：コンパイル時に宣言した自由変数が束縛を受ける際にバインド処理をする。

以上７種の命令はオペランドの指定により26個の中間言語命令となる。

５．２．１．２　最適化方式

本システムのコンパイラにおける最適化処理は、生成された中間言語オブジェクトコードの圧縮という形で行なっている。これは実行時における命令コードのフェッチとデコードによるオーバヘッドを軽減させるもので以下に示すことが行なわれる。

(1) 複数個のフレームを作るときは可能なかぎり１つのMKF 命令を用い個数はオペランドで指定する。MKF 命令の直後のPSH 命令はオペランドで指定。(MKF 1 NIL)(MKF 1 NIL)(PSHA-5)→(MKF 2-5)

(2) RET 命令直前の命令がPSH 命令であった場合はRET 命令のオペランドでPSH 命令を指定する。T やNIL を評価値としてRETURNする場合、T 、NIL 専用のRET 命令を用いる。

(PSHA-4)(RET) →(RET-4)

(PSH T)(RET)→(RET T)

(3) CONDの条件クローズにNULL、NOT が用いられたときはそれらの関数は呼出さずに引数によりその判断を行なう。

(4) ジャンプ命令が２つ並び前のものが条件ジャンプである場合、条件ジャンプの条件を変えることにより無条件ジャンプが削除できる場合には削除する。

…(JPN GO)(JMP B1)GO…→…(JPT B1)…

５．２．２　性能評価と分析

５．２．２．１　性能評価

第２回LISPコンテスト[Takeuchi 78] に出題されたプログラムを用いて実行時間を測定した結果を表 5.1に示す。インタプリタⅠは表 3.1で示した値であり、インタプリタⅡはその後改良して高速化したインタプリタである。コンパイラはコンパイルされたコードの実行時間で、インタプリタⅡの 1.74 ～ 2.86 倍の速度向上となっている。

TPU-6 のインタプリタⅠに対するコンパイラの速度向上率は 2.17 倍であり、第３章で予測した速度向上率よりはやや良い値となっている。

５．２．２．２　TARAI-3 を用いた動特性の分析

本LISPマシンにおいてコンパイラがインタプリタに対していかなる形で高速化をもたらしたかを分析するため、最も簡単でかつインタプリタの動的機能をよく利用している。TARAI-3 を例に上げて分析を試みる。図 5.2にTARAI-3 と

表5．1　テストプログラムの実行時間

プログラム名	インタプリタⅠ	インタプリタⅡ	コンパイラ
SORT-20	73	68	25
SORT-60	415	387	135
SORT-100	804	750	262
TARAI-3	55	52	23
TARAI-4	1,013	954	429
TARAI-5	27,538	25,938	11,663
TARAI-6	1,011,732	952,988	428,487
TPU-4	1,815	1,674	960
TPU-5	238	222	106
TPU-6	6,728	6,237	3,105

表5．2　TPU-6 の実行時間とマイクロプログラムの実行ステップ数

	実行時間 (msec)	マイクロステップ数	平均実行時間
インタプリタⅡ	6,237	18,621,434	335
コンパイラ	3,105	9,714,310	320

そのコンパイルコードを示してある。TARAI-3 は

(TARAI 6 3 0)

を実行することになる。なおTARAI-3 を実行したときのプログラムの各部分の実行回数もあわせて図 5.2に示してある。この実行回数のデータを基にして以後の分析を進めている。

コンパイルコード実行をするエミュレータの流れ図を図 5.1に示す。〈〉内に示された数は各ブロック内で実行されるマイクロプログラムステップ数を示したものである。

図 5.3はTARAI-3 をインタプリタで実行するときの実行の流れと各ブロックの実行回数、各ブロックの処理において実行されたマイクロプログラムステップ数を示している。各ルーチン群の実行ステップの内訳を示すと以下のようになる。

evalおよびeval①	12.6%
apply ②	20.1
argeval	25.9
cond	9.6
return	7.9
rebind	7.2
expr	1.0
sub 1	7.5
greaterp	7.6

またコンパイルコードを実行したときのマイクロプログラムステップ数を図 5.2にコンパイルコードとともに示してある。各ルーチンの内分けを示すと以下のようになっている。

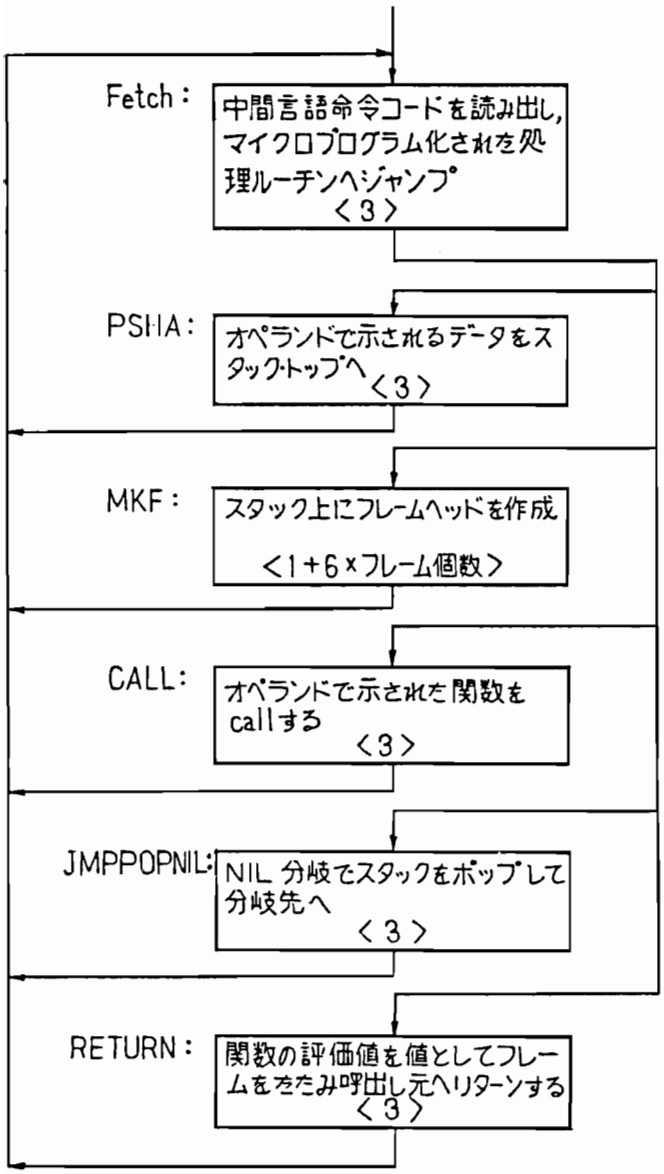

図５．１　コンパイルコードの実行の流れ図

Fetch	26.1%
PSHA	9.5
MKF	16.8
CALL（直接実行関数）	4.8
CALL（コンパイルされた関数）	2.8
JMPPOPNIL	2.7
RETURN	2.7
SUB 1	15.5
GREATERP	19.4

図 5.2のコンパイルコードと図 5.3のインタプリタの実行フローは機能上の対応付けが可能であり、それぞれ対応する部分をA:からD:の記号で示してある。それぞれの部分について、コンパイルコードの実行ステップ数に対するインタプリタのステップ数の比を求めると、

A:	2.86 倍
B:, C:	1.02 倍
D:	2.25 倍

となり、A:とD:の部分でインタプリタの処理ステップ数が大きくなっていることが分かる。この原因となる処理を特定するために前述のインタプリタ各ルーチンのステップ比率と図 5.3のA:、D:における処理内容を分析する。まずインタプリタに特有の処理としてA:中のapply ②とD:中のrebind→returnで表わされるバインディングに関する処理がインタプリタ中の31.4% を占めることが分かる。コンパイルされるとローカル変数はスタック上にとられるのでバインディングは不要となりもっとも大きな効率の改善となる。それに続くものとしてはCONDの処理に 9.6% のステップを要しているが、コンパイルされればオープ

```
(DE TARAI (X Y Z)                                    } 673 回
 (COND
  ((GREATERP X Y)
   (TARAI (TARAI (SUB1 X) Y Z)                       } 168 回
          (TARAI (SUB1 Y) Z X)
          (TARAI (SUB1 Z) X Y)))                     }
   (T Y) ))                                          } 505 回

   (LAP SUBR TARAI 3
        (MKF  1 -6)                                  } A:
        (PSHA   -6)                                  } 673 回
        (CALL 2 GREATERP)                            } <32034 ステップ>
        (JMPPOPNIL GO)
        (MKF  3 -14)                                 }
        (CALL 2 SUB1)                                }
        (PSHA   -10)                                 }
        (PSHA   -10)                                 }
        (CALL 3 TARAI)                               }
        (MKF  2 -14)                                 }
        (CALL 2 SUB1)                                } B:, C:
        (PSHA   -10)                                 } 168 回
        (PSHA   -13)                                 } <34776 ステップ>
        (CALL 3 TARAI)                               }
        (MKF  2 -14)                                 }
        (CALL 2 SUB1)                                }
        (PSHA   -13)                                 }
        (PSHA   -13)                                 }
        (CALL 3 TARAI)                               }
        (CALL 3 TARAI)                               }
        (RET   )                                     } D:
 GO     (PSHA   -2)                                  } 505 回
        (RET   )                                     } <6060 ステップ>
```

図5．2　TARAI とそのコンパイルコード

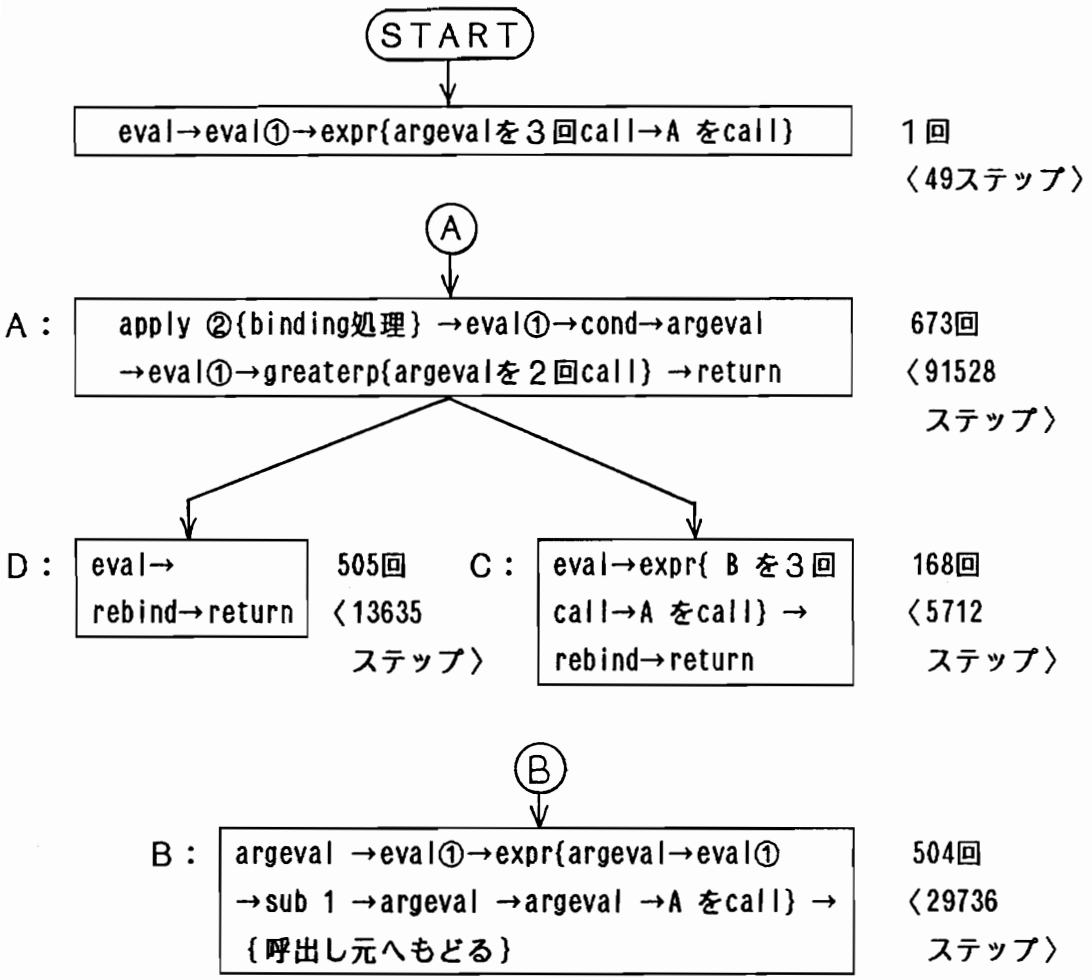

図５．３　ＥＶＡＬ（ＴＡＲＡＩ　６　３　０）の実行フロー

ン展開されるのでこのステップもほぼなくなる。

他にコンパイラに比べてインタプリタで必要となっている処理として、SUBR関数を呼ぶためのARGEVAL 中でのフレームの作成と、SUBR関数から戻るときのフレームの解放処理が考えられる。これらはA:とB:の中でargeval →eval①→greaterp、argeval →eval①→sub1として出現するが、B:、C:部のインタプリタとコンパイラのステップ比が小さいことから、インタプリタ中でのこれらの処理が、コンパイラ中ではPUSHやCALL命令のFetch のオーバーヘッドの中にある程度埋もれていると考えることができる。中間言語命令フェッチのステップ数は、コンパイルコード実行ステップ数の26% と大きな値を占めるが、この問題は 5.2.3節で述べる。

５．２．２．３　TPU-6 を用いた動特性の分析

次にプログラムサイズの最も大きな測定用プログラムであるTPU を用いて行なったインタプリタⅡとコンパイルコードの性能の測定結果について論じる。

実行時の総ステップ数とルーチンの総ステップ数を求めた結果が表 5.2、表 5.3である。まず表 5.3より、インタプリタにおけるシステム関数部の実行ステップ数が約 823万ステップであったのに対し、コンパイルした場合のシミュレーション部の実行ステップ数が約 300万と63.4% も短縮されていることが判る。またCAR 、CDR 等のマイクロプログラム化された直接実行関数では引数評価ルーチンのcallおよびRETURN処理の省略により大幅に実行時間が短縮されている。またNULLに関しては 5.2.1.2-(3)で述べたように関数を呼び出さないで引数のみでNULL機能を行なうためステップ数が大幅に減少した。

第３章でコンパイラ導入による速度向上は約 2倍であり実際にはFetch のオーバーヘッドによりそれより悪くなると予測したところが、実際には 2.17 倍

表5．3　TPU-6 実行時の各ルーチンの実行ステップ数と割合

インタプリタII				コンパイラ			
	ルーチン	ステップ数	割合(%)		ルーチン	ステップ数	割合(%)
システム関数	EVAL	1,818,316	9.8	エミュレーション部	FETCH	1,345,009	13.8
	ARGEVL	2,526,304	13.6		PSHA	480,633	4.9
	PROG	1,886,867	10.1		MKF-A	309,643	3.2
	その他	2,112,731	11.4		その他	878,929	9.0
SUBR関数	SUBST	3,363,520	18.1	SUBR関数	SUBST	3,293,744	33.9
	EQUAL	2,371,520	12.7		EQUAL	2,195,884	22.6
	CAR	570,245	3.1		CAR	131,595	1.4
	NULL	338,776	2.1		NULL	525	0.0
	CDR	291,395	1.6		CDR	67,245	0.7
	その他	3,001,725	17.7		その他	1,011,143	10.4

表5．4　中間言語命令の先取りをした場合の推定実行時間と改善率

	実行時間	推定実行時間	改善率(%)
TARAI-6	428,487	315,989	26.2
SORT-100	262	175	33.3
TPU-6	3,105	2,625	15.5

の向上となった主な理由は、上述のようにSUBR関数の引数評価部分のステップ数短縮とNULL関数の省略によるステップ数短縮が、Fetch のオーバヘッド分より大きかったことによっている。

５．２．３　命令先取りの効果予測

表 5.1の実行時間でインタプリタとコンパイラの場合を比較してみると、コンパイルすることによっても 1.5～ 3倍程度の処理速度の向上しか得られていない。これはインタプリタが高速に動作しているためでもあるが、中間言語命令のフェッチおよびデコードをマイクロプログラムで実現しているためと考えられる。特に命令コードのフェッチを行なう部分は関数の実行とは直接関係のない部分である。そこで、このフェッチによるオーバーヘッドをなくすために、ハードウェアによる命令コードの先取りを行なった場合の実行時間を推定してみた。つまり、プログラム領域の命令コード専用のPC（プログラムカウンタ）を用意し、あらかじめ命令コードを読み出しておき、その命令コードの実行が始まると、PCにより次の命令コードを読み出すという機能をハードウェアで実現した場合の実行時間である。表 5.4はこのようなハードウェアを設けることによりフェッチのオーバーヘッドが完全にとり除かれた場合の推定実行時間である。表 5.4よりTARAI 、SORTのような組み込み関数をあまり使用しないプログラムでは25～30% の改善率が期待できるが、TPU のように組込関数を多用するプログラムでは15% 程度の改善率となることがわかった。

５．２．４　マイクロコードへのコンパイルによる高速化

５．２．４．１　直接マイクロコード生成形コンパイラ

中間言語目的コード実行においては、目的コードのフェッチなど実際のLISPプログラム実行に直接結びつかない操作が必要となり冗長となっている。したがってLISPプログラムを直接マイクロコード化できればより高速な処理が実現できると考えられる。ここではA 、B 2つのバージョンのマイクロコード生成形コンパイラを想定し若干のベンチマークプログラムのマイクロコードの生成をマニュアルで行ない、LISPマシンのWCS （マイクロプログラムコード用メモリ）に格納し実行させ性能評価を行なった。

LISPプログラムのマイクロプログラム化は、現コンパイラによって生成された中間言語目的コードを利用し各目的コードの実行マイクロコード部分を連結して 1つのマイクロプログラムにすることにより実現している。この際に行なう最適化の程度によりＡ、Ｂのバージョンが想定されている。

バージョンＡ：中間言語命令コードのフェッチと再帰関数以外の関数呼出におけるフレーム作成の省略。

バージョンＢ：Ａに加え以下の最適化をしている。

(1) なるべく引数を内部レジスタに残しておき、スタックを介さずに引き渡すとか、スタック中の引数を取り出して再びスタック先頭に積む操作をやめ直接レジスタに読み込むなどの内部レジスタの有効利用。

(2) 演算結果がT かNIL の条件で、リターンするか否かが決る部分では結果のT 、NIL をスタックに積む操作を省略し条件分岐命令JPN でリターン。

(3) 並行実行可能な連続した分岐命令と演算命令は一命令に統合してマイク

表5．5　ベンチマークプログラムの実行時間

	INTERPRETER	COMPILER	VERSION-A	VERSION-B
TARAI-5	25,900	11,800	4,646	3,902
TARAI-6	952,400	429,300	170,712	143,367
BITA-7	178	80	40	31
BITA-8	618	276	136	108
BITB-8	113	73	30	23
BITB-9	377	244	101	75
SORT-80	572	209	95	75
SORT-100	749	277	123	98

表5．6　TARAI-5 実行時の各ルーチンの実行ステップ数と割合

COMPILER			VERSION-A			VERSION-B		
ルーチン	ステップ数	割合(%)	ルーチン	ステップ数	割合(%)	ルーチン	ステップ数	割合(%)
FETCH	9,262,959	28.3	PSH	2,229,968	16.9	PSH	1,029,216	9.0
PSH	2,573,043	7.9	MKF	1,886,900	14.3	MKF	1,972,664	17.2
MKF	7,118,753	21.7	CALL	257,304	1.9	RET	2,573,048	22.4
CALL	2,830,347	8.6	JP	686,146	5.2	EVAL	36	0.0
JP	686,146	2.1	RET	2,573,054	19.5			
RET	5,403,399	16.5	EVAL	36	0.0			
EVAL	27	0.0						
GREATERP	3,087,657	9.4	GREATERP	3,773,803	28.6	GREATERP	4,116,876	35.8
SUB 1	1,801,128	5.5	SUB 1	1,801,128	13.6	SUB 1	1,801,128	15.7
合　計	32,763,459		合　計	13,208,339		合　計	11,492,968	

ロコードの圧縮をする

(4) MAPCAR、MAPCONルーチンと引数関数とを直接統合することにより関数呼出やパラメータの受け渡しを簡略化する。

５．２．４．２　性能評価と考察

性能評価に用いたプログラムは第２回LISPコンテストにおいて出題されたプログラムからTARAI 、BITA、BITB、SORTを選んだ。実行時間の測定結果を表5.5に示す。プログラム中で最大実行時間のものを例にするとバージョンA 、B はそれぞれ現コンパイラに比して1/2.0 ～1/2.5 、1/2.8 ～1/3.2 に、インタプリタに比し1/3.7 ～1/6.1 、1/5.0 ～1/7.7 に実行時間を短縮している。

また表 5.6には動特性の測定結果の一部を示す。FETCH ルーチンの省略、MKF とCALL命令の減少または省略により大幅にステップ数を短縮していることがわかる。最適化の種類に応じ、コンパイルコードに比して実行ステップが、

(1) 目的コードフェッチルーチン(FETCH) の省略により20% ～31% （SORT-100）、

(2) 不要なフレーム作成の省略により10% ～25% （TARAI-6 ）、

(3) 引数の受け渡しをスタックのかわりに内部レジスタを用いて行なうなどの工夫により5%～15% （BITB-9）、

(4) 関数呼出の工夫、BINDの際のスタック上の値の有効利用、並列演算を生かしたマイクロコードの圧縮により7%～14% （BITA-8）

減少している。

またインタプリタからバージョンB のコンパイラに移るにつれて当然のことながら組込みのSUBR関数の実行時間比が増大しており、より高速化を目指すには高速演算回路の付加などのSUBR関数機能のハードウェア化が必要となってく

る。

５．３　逐次型推論マシンへのコンパイル方式の導入と評価

５．３．１　コンパイラの設計と試作

５．３．１．１　コンパイラの方式

基本的な考え方としては、LISPマシンと同じく中間言語命令を設定し、それをオブジェクトコードとして生成する方式をとった。中間言語には、D.H.D. Warrenにより提案された仮想マシン用の命令セット[Warren 83] に拡張を施したものを使用した。その特徴としては、

(1) ボディゴール中の引数、ヘッド中の引数は、各々引数１個が１個のユニフィケーション系の命令に変換される。

(2) レジスタをユニフィケーション系命令で取扱えるようにし、レジスタへの変数割付けをコンパイラで最適化することによりユニフィケーション回数自体を削減する。

(3) 述語呼出時の引数をレジスタ渡しとして(2) の機能を活用し、テイルリカージョンの最適化にも利用する。

(4) ストラクチャコピー法を用いる。

(5) 述語呼出しから復帰後の実行継続に必要な情報environment とバックトラック時の再試行に必要な情報choice pointを分離してスタックに積むようにし、それを行なう命令を各々必要なときにだけコンパイラで生成する。

(6) カットの機能とPSI 用の拡張実行制御機能を包含する。

などがあげられる。

またPSI ハードウェアの活用と一部方式の見直しにより次のような高速化を図った。

(1) マイクロプログラムによる命令の先読み制御を行ない、中間言語命令サイクルの始まりはいきなりディスパッチから開始してステップ数を短縮する。

(2) メモリ割当てが可能か否かのファームウェアでのテストを省略して割当て処理を先行させ、不都合があったときは割込みで正規のメモリ割当て処理を行なわせステップ数を短縮する。

５．３．１．２　中間言語命令

中間言語命令には、Prolog系言語用のユニフィケーション系の命令、インデキシング命令、カット命令と、組込述語命令があり、ほとんどの命令コードをタグ部分に割付けた。これにより命令先読みでメモリインタフェースのデータレジスタへ読込まれた命令を用いてただちにタグディスパッチをかけることを可能とし命令フェッチのオーバーヘッドを低減させた。またロジカルアドレスレジスタの１本をプログラムカウンタとして固定的に使用し、先読み時には同レジスタの自動増加機能を活用した。このように先読みを命令実行ルーチンの中で平行して行なうため、命令フェッチとディスパッチの専用ルーチンというものは使用していない。中間言語は次のような命令セットから成り、１語中にほとんどの場合１語か２語のオペランドを含む。

(1) get 系命令：クローズヘッドの引数に対応する命令で、引数レジスタに与えられた引数とのユニフィケーションを行なう。次のような命令がある。

get_variable　　get_builtin _variable

get_value　　get_atom　　get_integer

get_constant　　get_vector　　get_list

(2) put 系命令：ボディゴール中の引数に対応し、引数を引数レジスタにロードする。次のような命令がある。

put_variable　　put_value　　put_local _value

put_unsafe_value　　put_builtin _value　　put_constant

put_nil　　put_vector　　put_list

(3) unify 系命令：構造体あるいはリスト中の変数に対応し、既存の構造体要素とのユニフィケーションまたは新しい構造体要素の作成を行なう。次の命令がある。

unify_void　　unify_variable

unify_local _variable　　unify_value

unify_local _value　　unify_atom

unify_integer　　unify_constant

(4) procedural命令：述語の呼出し、復帰、environment の割付け、除去の制御を行なう。次のような命令がある。

allocate　　call

execute　　execute _with_deallocate

proceed　　proceed _with_deallocate

(5) indexing命令：述語を作り上げている複数のクローズをつなぎ合わせ呼出し順を与える命令とクローズインデキシングを行なう命令がある。choice pointを作る命令も含まれる。次のような命令がある。

try _me_else　　try

retry _me_else　　retry

trust _me_else_fail　　trust

switch_on_term　　switch_on_constant

switch_on_hash_constant　　switch_on_structure

switch_on_hash_structure

(6) cut 命令：KL0 のカットに対応する命令で、最適化のために次のような種類を用意した。

cut _me　　cut _allocated _me

cut _me_and _proceed　　cut _normal

cut _and _proceed　　cut _me_and _fail

cut _and _fail　　cut _method

(7) builtin 命令：組込述語はオペランドにレジスタが指定できるように作りなおし、評価用プログラムが必要とするもの約35種を用意した。

５．３．１．３　最適化方式

本システムのコンパイラでは主に次の３種類の最適化を実施した。

(1) 永久変数と一時変数を判別して一時変数はレジスタに割付け、永久変数の存在するときだけenvironment を作る。

(2) 一時変数と引数用のレジスタ割付けを最適化し、 get_variable、 put _value の削除により不要なユニフィケーションを除く。

(3) 第１引数によるクローズインデキシング処理を行ない成功時にはchoice point 作成命令をスキップするようコード作成する。

図 5.4にコンパイルコードの例を示す。

```
nreverse([X|L0],L) :- nreverse(L0,L1), append(L1,[X],L).
nreverse([],[]).

append([W|X],Y,[W|Z]) :- !, append(X,Y,Z).
append([],Y,Y).
```

```
        nrev:   switch_on_term N1a,N2,N1,fail

        N1a:    try_me_else N2a
        N1:     allocate 3
                get_list R0
                unify_variable_parm Y1
                unify_variable_temp R0
                get_variable_parm R1,Y0
                put_variable_parm R1,Y2
                call nrev,2
                put_unsafe_value R0,Y2
                put_list R1
                unify_value_parm Y1
                unify_nil
                put_value_parm R2,Y0
                execute_with_deallocate appe,3

        N2a:    trust_me_else fail
        N2:     get_nil R0
                get_nil R1
                proceed


        appe:   switch_on_term A1a,A2,A1,fail

        A1a:    try_me_else A2a
        A1:     get_list R0
                unify_variable_temp R3
                unify_variable_temp R0
                get_list R2
                unify_value_temp R3
                unify_variable_temp R2
                cut_me
                execute appe,3

        A2a:    trust_me_else fail
        A2:     get_nil R0
                get_value_temp R2,R1
                proceed
```

図5．4　nreverseのコンパイルコードの例

５．３．２　性能評価

第４章で評価に使用したPrologコンテストの課題の一部と、応用プログラムの8-puzzle、harmonizerを用いて実行時間を測定した。その結果を表 5.7に示す。

コンパイラはインタプリタに比べ 1.74 倍から 3.36 倍高速であり、最高の推論速度は 114.5KLIPS を達成していることがわかる。プログラムによる速度比のばらつきは、(1) から(6) の単純なプログラムについて見ると、表 4.1のインタプリタとDEC-10 Prolog を比べたときのばらつきと同じ傾向を示している。すなわち、インタプリタに比べDEC が速くなる傾向を持つプログラム（最適化の可能なプログラム）では、インタプリタとコンパイラの速度比も同様に大きくなっており、本コンパイラにおいてDEC と同じ傾向の最適化が実現されていることが分かる。また絶対的な実行速度で比較すると本システムはDEC の約２倍から３倍の性能であることが分かる。

インタプリタに対する本コンパイラシステムの高速性は、前節で述べたコンパイラにおける最適化で得られている他に、次の事項によっている。

(1) マイクロプログラム制御による命令先取り

(2) ページ割当ての要否判定を通常の実行フローから外し割込で知らせるように変更したこと

(3) 使用するスタックを３本に減らすとともにchoice pointとenvironmentを分離したことにより、environment としてスタックへ積む情報量が減少したこと

したがって、コンパイルにより得られた高速化比率の 1.74 から 3.36 倍という値のうち、純粋にコンパイラによる最適化で得られている部分は、もう少

表５．７　評価用プログラムの実行時間

programs	compiler (msec)	KLIPS on compiler	interpreter (msec)	interpreter /compiler
(1) nreverse (30)	4.35	114.5	14.6	3.36
(2) quick sort (50)	6.73	90.8	16.1	2.39
(3) tree traversing	28.95	107.9	52.2	1.80
(4) 8 queens (1)	48.10	---	105	2.18
(5) reverse function	22.31	59.5	38.9	1.74
(6) slow reverse (4)	2.58	82.5	6.13	2.38
(7) 8-puzzle	2442	---	6544	2.68
(8) harmonizer-1	276	---	657	2.34
(9) harmonizer-2	784	---	1879	2.39
(10) harmonizer-3	9846	---	24119	2.48

し低い値になると考えられる。

５．３．３　LISPマシンとの比較による考察

５．３．３．１　性能比較表の準備

表 5.8にはLISPプログラムの実行時間比を示す。LISPマシン上でのコンパイルコードの実行時間を１としたときのインタプリタの実行時間比、および第３回LISPコンテスト[Okuno 85]に参加した３つの処理系の実行時間比を示す。実行時間そのものは、示された比の値にコンパイラ欄のカッコ内に示した実行時間の基準値をかければ求まる。MACLISP はMIT で作られ最適化が良く行なわれる処理系として知られており、DEC-2060で実行させている。VAX LISPはDEC 社で最近作られたCommon LISP の一種で最適化についてはいま一歩の可能性がある。VAX11/785 上で実行した値である。SYMBOLICS COMMON LISP は米国のLISPマシンとして知られたSYMBOLICS 3600上の処理系である。

表 5.9にはPrologプログラムの実行時間比を示す。PSI 上でのコンパイルコードの実行時間を１とした時のインタプリタの実行時間比、その他ICOTで計測した２つの処理系の実行時間比を示す。DEC-10 Prolog は第４章で比較の対象としたものであり最適化のよくかかる処理系として知られている。Quintus PrologはDEC-10 Prolog の製作者であるD.H.D. Warren 氏が最近完成させた処理系で、コンパイラにより文献[Warren 83] に示された中間言語命令列を生成し、それを機械語で書かれたエミュレータ（一種のインタプリタ）で実行する。最適化は十分であるが、中間言語実行のための多少のオーバーヘッドが存在すると考えられる。

表 5.8の中で、VAX LISPとSYMBOLICS はプログラムにより実行時間比に大き

表５．８　LISPプログラムの実行時間比

programs	compiler on LISP Machine (time=msec)	interpreter -II LISPMachine	MACLISP DEC-2060	VAXLISP VAX-785	SYMBOLICS COMMONLISP 3600+IFU
(1) tarai-4	1 (429)	2.22	1.49	1.43	0.20
(2) tarai-5	1 (11663)	2.22	1.61	1.43	0.20
(3) tarai-6	1 (428487)	2.22	1.51	--	0.20
(4) TPU-4	1 (960)	1.74	1.58	4.29	1.52
(5) TPU-5	1 (106)	2.09	1.06	4.24	1.52
(6) TPU-6	1 (3105)	2.01	1.57	4.00	1.32
	CPU cycle time = 300nsec		around 62.5nsec	133nsec	around 200nsec

表５．９　Prologプログラムの実行時間比

programs	compiler on PSI (time=msec)	interpreter on PSI	DEC-10 Prolog DEC-2060	Quintus Prolog VAX-785
(1) reverse (30)	1 (4.35)	3.36	2.18	3.37
(2) quick sort (50)	1 (6.73)	2.39	2.17	4.34
(3) tree traversing	1 (28.95)	1.80	2.11	4.01
(4) 8 queens (1)	1 (48.10)	2.18	2.03	4.74
(5) reverse function	1 (22.31)	1.74	1.87	--
(6) slow reverse (4)	1 (2.58)	2.38	2.10	4.46
(7) harmonizer-3	1 (9846.)	2.45	3.19	--
	CPU cycle time = 200nsec		around 62.5nsec	133nsec

なばらつきのあるのがわかる。これは小さいtarai プログラムに対して特に効果のある最適化が実施されているためであり、性能比較には規模が大きくより実際的なTPU プログラムを用いることにする。表よりLISPマシンのコンパイルコードは、DEC-10 Prolog に比べて約 2倍、Quintus Prologに比べて約 4倍高速であるのが分かる。

５．３．３．２　コンパイラとインタプリタの性能比

表 5.8と表 5.9より、LISPマシンのインタプリタとコンパイラの性能比は約２倍、PSI の場合には 2倍よりやや大きめであることが分かる。但しPSI の場合には、 5.3.2で述べたようにコンパイルしたこと以外に方式変更による高速化が図られているため、コンパイラによる高速化のゲインは、LISPマシンとPSI でともに 2倍程度かあるいはPSI の方が少し低いと考えられる。

このことから、インタプリタの高速化を目指して設計した専用マシン上のインタプリタと、その上にあとから実装したコンパイラシステムの性能比は、プログラムによりばらつきはあるものの、LISP用の専用マシンでもProlog系言語用の専用マシンでもともに 2倍前後であるということができる。

但しコンパイルコードの中間言語実行用にマシンを再検討するならばさらに高速化は可能である。高速化のポイントは、まず命令の先取り機能を備えること、次に命令のオペランド部からLISPマシンであればスタックのフレーム上の変数セルを効率よくアドレッシングできること、逐次型推論マシンであればレジスタを効率よく指定できることが必要である。これらによってさらに、 1.5倍程度までの高速化が可能と予想している。このことから総合して、インタプリタ向きに設計した専用マシンとコンパイラ向きに設計した専用マシンの性能比はおよそ 3倍前後、あるいはマイクロプログラムとコンパイラの最適化を十

分に頑張るならばさらにもう少しの性能差が出ると予想する。

以上から処理速度を追求するならばコンパイラ向きの専用マシンを設計すべきであると結論される。しかしながら汎用計算機上の処理系に比べるとインタプリタとコンパイラシステムの速度差が十分に縮められることから、どうしてもインタプリティブな実行を必要とする応用がある場合には、コンパイラ向き専用マシンに比べて３倍程度の遅さのインタプリタ向き専用マシンが実現可能であるといえる。

５．３．３．３　専用マシンと汎用マシンの性能比

ハードウェア規模が同等の専用マインと汎用マシンとの間にどの程度の性能比があるかについては、興味が持たれながらもこれまで比較のための適当な材料がそろわなかった。本節ではこの点について、LISPとProlog用マシンに限定し、多少議論の精度は荒いながらも検討を加える。

まず表 5.8のLISPマシンのコンパイラとMACLISP （コンパイラ）の性能比および表 5.9のPSI のコンパイラとDEC-10 Prolog （コンパイラ）の性能比を比較する。ここではMACLISP もDEC-10 Prolog も十分な最適化が実現されていると仮定している。MACLISP はLSIPマシンに比べると約 1.5倍遅く、DEC-10 PrologはPSI に比べ約 2倍遅い。今LISPマシンの使用素子を古いICから最近のICに置きかえるとサイクルタイムは200nsec に短縮でき、実行時間は 0.67 倍になる。この新しいLISPマシンとMACLISP の性能比は約 2.2倍となる。PSI のサイクルタイムも200nsec であるから、200nsec のサイクルタイムを持つ専用マシン（PSI と新しいLISPマシン）と、MACLISP やDEC-10 Prolog が走っているDEC-2060の性能比は、LISPでもPrologでも約 2倍ということができる。

DEC-2060はECL で構成されたマシンでCPU サイクルタイムは62.5nsec程度と

推定され（レジスタファイルアクセスが125nsec であることから）、CPU の構成はPSI やLISPマシンと同じく実行段の多段パイプラインを使用しないシンプルなものである。そこでDEC-2060のアーキテクチャを変えずにサイクルタイムを 3.2倍の200nsec に変えると、実行時間も約 3倍に延びるはずである（厳密にはキャッシュミスヒット時の主メモリアクセス待ちのサイクル数が減るため、3.2倍よりも延び方は多少小さい）。これを仮りに改変DEC-20と呼ぶと、先に議論したサイクルタイム200nsec の専用マシンとこの改変DEC-20との性能比は約 6倍になる。但しDEC-2060はCPU サイクルタイムとレジスタファイルのアクセスタイムに２倍ぐらい開きのある古い設計であるため、議論の厳密性は乏しい。

そこで設計のより新しいVAX11/785 について同様な検討を行なってみる。同マシンはVAX11/780 （サイクルタイム200nsec ）とまったく同じ設計でサイクルタイムだけを３分の２にしたマシンである。VAX11/780 は実行段の多段パイプラインがない点でPSI などと同じシンプルな構成であり、ハードウェア規模もPSI に近いと考えられる。そこでVAX11/785 での実行時間を 2分の 3倍にすると、ほぼVAX11/780 の実行時間となり、サイクルタイム200nsec の専用マシンのきわめてよい比較対象となる。

Quintus Prologの実行時間を 2分の 3倍にしてPSI のコンパイラとの性能比を求めると約 6倍となる。一方VAX LISPの実行時間を 2分の 3倍してLISPマシンのコンパイラと比べると約 6倍となり、さらにLISPマシン側もサイクルタイム200nsec の新しいLISPマシンに換算すると性能比は約 9倍となる。すなわち、サイクルタイム200nsec の専用マシンと、サイクルタイムが同じの汎用マシンVAX11/780 との性能比は、Prologで約 6倍、LISPで約 9倍ということになる。但しVAX LISPはCommon LISP 仕様で処理系が多少重いことと最適化の程度が十

分でない可能性があるため、同仕様のLISPで同程度の最適化がされていれば性能差は 9倍より小さいと考えられる。実際に PSI、 DEC-10 Prolog、Quintus Prologの実行時間比 1： 2： 4と、新しいLISPマシン、 MACLISP、VAX LISPの実行時間比 0.67 ： 1.5： 4＝ 1： 2.25 ：6 を見てもVAX LISPが遅そうであることが読み取れる。したがってサイクルタイムが200nsec の専用マシンと、同サイクルタイムのVAX11/780 の性能比は、処理系の重さと最適化の程度が同じならば、PrologでもLISPでも 6倍に近い値であると考えられる。

このようにDEC-2060とVAX という２つの例から、LISPまたはProlog用の専用マシンと、サイクルタイムとハードウェア規模が同程度の汎用マシン上の処理系との性能比は、 6倍前後であると結論する。

ここでの議論はインタプリタ向きに設計した専用マシン上でのコンパイルコードの実行時間と、汎用マシン上での現存する処理系の実行時間の比較であった。したがって、専用マシンはコンパイルコードの実行向きに再設計するとさらに 1.5倍ぐらい速くなる可能性を有しており、またQuintus Prologなどは中間言語のエミュレータによる実行のかわりにコードをオープン展開するなどの最適化でさらに高速化できることに注意する必要がある。

またここでの議論の対象とした汎用マシンは、実行段の多段パイプラインを持たない中・小型機種であり、 5段以上の多段パイプラインを持つ汎用超大型計算機の実現方式を考える場合には、同様な実現方式の専用マシンとの性能差が 6倍も出ないことに注意する必要がある。その根拠は次のとおりである。実行段パイプラインを持たない実現方式で６倍の性能差を生み出しているものは

(1) 言語の実行メカニズムに適したハードウェア、具体的には

タグ判定機構

条件分岐の多様性

スタックアクセスの高速化のサポート

(2) 適切な中間言語命令の設定によるマイクロプログラムレベルでの処理の最適化

の２点である。これらを実行段の多段パイプライン方式の中で実現するためには、パイプの各段を流れている複数個の命令の各々について(1) の機構を独立にサポートする必要があり、同時にパイプの流れの乱れ方を極力押さえる必要があること、またマイクロプログラムで行なっていた最適化を主にパイプの切り方と各段での実行制御というハードウェアに置きかえねばならないことで、これらを同時にかつ十分に実現することの困難さは想像に難くないと考えられるためである。

５．４　結　　言

本章では第３章と第４章でインタプリタ方式により試作したLISPマシンと逐次型推論マシンに対してコンパイラを試作し、主に処理速度の向上に関する評価を行なった。また両者の比較をとおして、インタプリタとコンパイルコードの速度比、専用マシンと同規模の汎用マシンとの速度比について考察した。

コンパイラの製作に当たっては、両処理系ともオブジェクトコードとなるべきLISP向きおよびProlog向きの中間言語命令を設計し、それらをターゲットとしてあまり重くない程度の最適化を行なうコンパイラを設計した。LISPマシン用のコンパイラはスタック上で引数の受渡しをするのに対し、逐次型推論マシン上のコンパイラはレジスタ上に引数を最適配置するところがもっとも異なっ

ている。

第２回LISPコンテストと第１回Prologコンテストの課題プログラムにより実行時間を測定した結果、インタプリタに比較してLISPマシンでは 1.74 から 2.86 倍、逐次型推論マシンでは 1.74 から 3.36 倍の高速化が達成された。

動特性の分析からLISPマシンで高速化の効果の大きいのは、バインディング処理の省略、つづいてCOND処理の省略と判明した。また命令先取りの効果の推定値はTPU プログラムで15% 程度となった。逐次型推論マシンでは、レジスタ割付の最適化などの他に、マイクロプログラム制御による命令先取りなどが高速化に貢献している。

LISPマシンではさらにマイクロコードへのコンパイラによる高速化を検討し、中間言語へのコンパイラに対しさらに 2.8倍から 3.2倍高速化されることを確認した。

両処理系の比較から、代表的な評価用プログラムで比較した場合、インタプリタとコンパイルコードの実行時間比はLISPでもPrologでもともに約２倍となった。コンパイルコード実行向きにマシンを再設計することによりこの比率は３倍程度まで向上しそうであるが、汎用マシン上のインタプリタとコンパイルコードの実行時間差に比べ、この比率はきわめて小さいといえる。

また両専用マシンのコンパイルコードの処理速度をCPU の実現方式が（多段実行パイプラインを持たない点で）近いVAX11/785 とDEC-2060の２つのシステム上の処理系と比較した。各々の処理時間を各マシンのCPU サイクルタイムで正規化したときの専用マシンと汎用マシンの性能差は、LISPとProlog、VAX とDEC-2060によらず 6倍前後という値が得られた。すなわちCPU の、サイクルタイムと実現方式が同じような専用マシンと汎用マシンを比較した場合、LISPやPrologに関する性能差は 6倍前後であるという目安を得た。この値は専用マシ

ンのハードウェアをコンパイルコード実行用にさらに最適化すれば開く傾向にあり、汎用マシン上のコンパイラにおける最適化を頑張れば縮む傾向にあるが、どちらも 1.5倍以内と考えている。

但し多段実行パイプラインを備えた汎用超大型計算機のような実現方式をとる場合には、高速化への貢献度の高い言語実行メカニズムのハードウェアサポート、マイクロプログラムレベルでの最適化の２つが多段パイプラインハードウェア中に十分取込むことが容易でなく、 6倍の高速化を達成することは難しいことを述べた。

第６章　結　論

本論文では人工知能向き言語であるLISPおよびPrologを効率良く実行する２つの専用マシンの試作と評価をとおして、

(1) LISP言語およびProlog言語の高速実行向きアーキテクチャとはいかなるものか。また両者に違いはあるか。

(2) 実用マシンを指向したときのアーキテクチャ上の特徴は何か。また実験マシンに比べてのハードウェア量の増加はいかなる程度か。

(3) インタプリタ向きに設計した専用マシンとコンパイルコードの実行向きに設計した専用マシンの性能差はどの程度か。

(4) 同規模のハードウェアで構成された専用マシンと汎用マシンの性能差は、LISPやPrologに限った場合にいかなる程度となるか。

の４点に関する検討と考察を行ない、人工知能向き高級言語専用マシンを設計するに当たっての採用すべきアーキテクチャ、実現できる速度、ハードウェアコストに関する指針を与え、また汎用計算機上の処理系との性能差に関する目安を提示した。

人工知能向き高級言語専用マシンに対する要求は、高速化、メモリの大容量化、パーソナル化、対話型入出力装置の強化であり、実用化に当たってはこれらのすべてを満たす必要がある。またこれらはLISPマシンとPrologマシンに共通の要求である。高速化を実現するためには、言語とマシンハードウェアの間のセマンティックギャップを縮める必要があり、その方法として、

・言語向きハードウェアアーキテクチャの設計

・マイクロプログラムによる最適化と適切な命令レベルの設定

・最適化コンパイラの設計

の３種類の組合せが必要である。マシン実現のための基本方式のうち、言語の実行メカニズムとしては、設計の容易なマイクロプログラムによるインタプリタ方式と、最適化により高速化が図れるコンパイラ方式がある。またハードウェアの実現形態としてはパーソナル形の実用マシンに適するスタンドアロン方式と、設計コストが低いため実験マシンに適するバックエンド方式があり、目的に応じて使い分けることが必要である。

最初に試作したLISPマシンでは、LISPの高速実行向きアーキテクチャの研究を主目的として、設計工数の低いバックエンド方式を使用した。２台目の逐次型推論マシンでは、Prologを拡張した言語の高速実行向きアーキテクチャを追求するとともに、マシンの実用性も重視してスタンドアロン方式を採用した。また両マシンとも最初の言語実行メカニズムには、設計の容易なマイクロプログラムによるインタプリタ方式を採用した。完成したマシン上で第２回LISPコンテスト、第１回Prologコンテストの課題プログラムと若干の応用プログラムを実行させ、性能とアーキテクチャの評価を行なうとともに、LISP向きとProlog向きのアーキテクチャの比較検討と、実験マシンと実用マシンの相異点に関する考察を行なった。

性能に関する結果は、両マシンともインタプリタ方式でありながら、汎用大型マシン上のコンパイルコードの実行時間と同等かそれ以上の性能を示した。高速性を解明するため、言語実行向きに導入したハードウェアの動特性を測定したところ、効果の大きかったハードウェア機能はLISPマシンでも逐次型推論マシンでもほぼ共通しており、

・スタックアクセスの高速化機能

・ガーベジコレクション用のビットを含むタグアーキテクチャとタグ値にもとづく多方向分岐機能（LISPマシンではマッピングメモリで実現）

・多様な条件分岐機能

・スタック以外のメモリアクセスの高速化機能

・マイクロ命令レベルでの並列制御の機能

の５つであった。またLISP向きとProlog向きで異なるのは

・LISPではハードウェアスタックが効果的なのに対し、Prologではスタック上でのデータアクセスの分散とスタック本数の多さからキャッシュメモリの方が適すること。

・Prologでは実行管理情報がLISPより多いことからレジスタファイルを多く用いる傾向にあること。またフレームバッファの実現やコンパイルコードの実行でもレジスタファイルを多用すること。

の２点であった。この中でPrologの応用プログラムの実行でキャッシュヒット率が96% 以上あったことや分岐命令の頻度がLISPマシンで40% 、逐次型推論マシンで70% もあるなどの興味深いデータが得られた。また逐次型推論マシンで実用化のために導入された機能としては、

・ページングによる大容量メモリの管理と効率的再配置の機能

・マルチプロセスのサポート機能

・入出力バスの制御と割込み機能

・メモリエラー訂正とエラー検出および保守とデバッグの機能

があり、これらを実現するためのハードウェアはマシン全体の約40% を占め、ハードウェアの追加とファームウェアの複雑化による性能低下は20% 余りとなった。

次にそれぞれのマシンに対して言語向きの中間言語命令とそれをターゲット

とするコンパイラを設計した。LISPマシン上のコンパイラはスタック上で、逐次型推論マシン上のコンパイラはレジスタ上でデータの受渡しを行なう方式とし、各々最適化処理を行なわせた。先に述べたものと同じ評価用プログラムにより実行時間を測定した結果、インタプリタに比べLISPマシンでは 1.74 から 2.86 倍、逐次型推論マシンでは 1.74 から 3.36 倍の高速化が達成された。またLISPマシンではマイクロコードへのコンパイラを検討し、中間言語へのコンパイラに比べてさらに 2.8倍から 3.2倍高速化されることを確認した。 両処理系とも、もっとも実用レベルに近い評価用プログラムでは、インタプリタとコンパイルコードの実行時間比が約 2倍となった。コンパイルコード実行のために命令の先読み回路やオペランドを切出してスタックやレジスタファイルの指定に使う機能を導入すると、さらに 1.5倍程度の高速化が可能と考えられた。したがってインタプリタの高速実行向きに設計した専用マシンとコンパイルコード実行向きに設計した専用マシンの性能差は、LISPでもPrologでも約 3倍程度になると考えられる。この比率は汎用マシン上のインタプリタとコンパイルコードの実行時間比に比べて十分小さいものであり、もしもインタプリティブな実行を必要とする応用がある場合には、マイクロプログラム化インタプリタ方式による専用マシンでも速度的に見て十分に利用可能な範囲にあるといえる。

最後に両専用マシンのコンパイルコードの処理速度をCPU の実現方式が（多段実行パイプラインを持たない点で）近いVAX11/785 とDEC-2060の２つのシステム上の処理系と比較した。各々の処理時間を各マシンのCPU サイクルタイムで正規化したときの専用マシンと汎用マシンの性能差は、LISPとProlog、VAXとDEC-2060によらず 6倍前後という値が得られた。すなわちCPU のサイクルタイムと実現方式が同じような、専用マシンと汎用マシンを比較した場合、LISP

やPrologに関する性能差は 6倍前後であるという目安を得た。この値は専用マシンのハードウェアをコンパイルコード実行用にさらに最適化すれば開く傾向にあり、汎用マシン上のコンパイラにおける最適化を頑張れば縮む傾向にあるが、どちらも 1.5倍以内と考えている。但し、多段実行パイプラインを備えた汎用超大型計算機のような実現方式をとる場合には、本研究で有効性の確かめられた言語実行メカニズムのハードウェアサポート、マイクロプログラムレベルでの最適化の２つが多段パイプラインハードウェア中に十分取込むことが容易でなく、６倍の高速化を達成することは難しいと考えるものである。

謝　辞

本研究全体をまとめるにあたって暖かい御理解と御指導を賜わり、また逐次型推論マシンの試作と評価の機会を与えて下さった（財）新世代コンピュータ技術開発機構（ＩＣＯＴ）の渕一博所長に心から感謝の意を表する。また直接の御指導と数多くの有益な御助言を賜わった同研究所の内田俊一室長に心から感謝する。

LISPマシンシステムの試作と評価ではその機会を与えてくださり、暖かい御指導とともに研究機材の整備に御尽力を賜わった神戸大学工学部システム工学科の前川禎男教授に心から感謝の意を表する。また同学科の金田悠紀夫助教授には直接の御指導、御助言を賜わったとともに、論文投稿の折り筆者にかわり多大の御努力をいただいた。特に記して感謝の意を表する。また元システム工学科第４講座の小林康博氏には、ハードウェアの製作とコンパイラの試作・評価に多大の努力を払っていただいた。心から感謝する。また同じく元第４講座の滝本博道氏、多田光弘氏、三上昭弘氏、高岸英之氏には、ハードウェア、ソフトウェアの製作とともに評価データの収集に御努力いただいた。深く感謝する。また和田耕一氏をはじめとする第４講座諸氏と小畑正貴氏をはじめとする神戸大学電子計算機研究会の諸氏には、ハードウェアの製作と図面作成などに協力いただいた。感謝の意を表する。また本システムを企画する段階において有益な御討論をいただいた通産省工業技術院電子技術総合研究所の島田俊夫氏、山口喜教氏に深く感謝する。

逐次型推論マシンシステムの試作と評価では絶えず励ましと御指導をいただ

いたＩＣＯＴの横井俊夫室長に心から感謝の意を表する。また方式設計とファームウェアの設計・試作に多大なる努力を払っていただいたＩＣＯＴ第４研究室の横田実氏に心から感謝する。同じく元第４研究室の西川宏氏、山本明氏に深く感謝する。また第４研究室の近山隆氏には数多くの有益な御助言をいただいたことを感謝する。詳細設計、試作と評価にあたっては、三菱電機株式会社情報電子研究所の中島浩氏、同じく中島克人氏（現ＩＣＯＴ）にひとかたならぬ御協力をいただいた。深く感謝の意を表する。またファームウェアの製作と評価で御努力いただいた沖電気工業株式会社の諸氏に感謝する。また終始励ましと有益な御討論をいただいたＩＣＯＴのオペレーティングシステム開発グループの諸氏、古川康一室長をはじめとするＩＣＯＴの関連研究室の諸氏に感謝する。David H.D. Warren 氏（現在英国マンチェスタ大学教授）には方式検討段階において有益な御助言、御討論をいただいたことを深く感謝する。

また本論文をまとめるにあたり御指導と御助言を賜わった神戸大学工学部計測工学科の松本治彌教授、同じく電子工学科の平野浩太郎教授、同じくシステム工学科の藤井進教授に深く感謝する。

最後に本研究をまとめるにあたり、家庭を守り心身両面における支えとなってくれた妻明美に心から感謝する。

参 考 文 献

[AMD 85] Bipolar Microprocessor Logic and Interface Data Book, Advanced Micro Devices, Inc. (1985).

[Bawden 77] Bawden, Aian et al.:LISP Machine Progress Report, MIT AI Lab. Memo No.444(Aug. 1977).

[Bobrow 73] Bobrow, D.G.:A Model and Stack Implementation of Mutiple Environments. C. ACM Vol.16, No.10 (1973).

[Bowen 81] Bowen, D.L.:DEC system-10 PROLOG USER'S MANUAL, Dept. of Artificial Intelligence Univ. of Edinburgh (1981).

[Boyar 72] Boyer, R.S., Moore, J.S.:The Sharing of Structure in Theorem Proving Programs, Machine Intelligence Vol.1 -7, Edinburgh Up (1972)

[Bruynooghe 82] Bruynooghe, M.:The Memory management of PROLOG impulementations, Logic Programming (eds. K.L.Clark and S.A.Taernlund), pp.83-98, Academic Press, New York (1982)

[Chikayama 83] ESP-Extended Self-contained Prolog-as a Preliminary Kernel Language of Fifth Generation Computers, New Generation Computing Vol.1, No.1, Ohmusha, Ltd. (1983)

[Chikayama 84] Chikayama, T.:ESP Reference Manual ICOT Technikal Report TR-044, Feb.3 (1984)

[Cohen 81] Cohen, J.:Garbage Collection of Linked Data Structures, Computing Serveys, 13-3 (1981)

[Computer 77] STACK MACHINES, IEEE, COMPUTER Vol.10 No.5, (May. 1977) .

[Deutsch 73] A LISP machine with very compact programs, Proc.of 3rd

IJCAI. p.697 (1973).

[Deutsch 79] Deutsch, L.P.:Experience with a Microprogrammed Interlisp System, IEEE Trans. Comp., Vol.C-28, No.10, (Oct. 1979).

[ETL 78] LISP 1.9 User's Mannual, EPICS-5-ON-2, ETL.

[Fuch 83] 渕一博、廣瀬健：第五世代コンピュータの計画、MONAD BOOKS 27, 海鳴社

[Fuchi 84] 渕一博、赤木照夫：第５世代コンピュータを作る、日本放送出版協会(1984)。

[Furukawa 84] 古川康一：Prolog総論、情報処理、Vol.25, No.12, pp.1313-1318 (1984).

[Goldberg 83] Goldberg, A., Robson, D.:Smalltalk-80 －The Language and Its Implementation, Addison-Wesley Publishing Co. (1983).

[Greenblatt 74] Greenblatt, R.:The LISP Machine, MIT AI Lab. Working Paper 79 (Nov. 1974).

[Hagiwara 77] 萩原宏：マイクロプログラミング、産業図書、コンピュータサイエンス・ライブラリー(Apr. 1977).

[Hattori 83] Hattori, T., et al.:Basic Construct of SIM Operating System, New Generation Computing Vol.1, No.1, 1983

[Hishinuma 76] 菱沼、山下、中西：LISPインタプリタにおけるスタック技法とａリストの抑制法、情報処理Vol.17, No.11, p.p.1002-1008, (Nov. 1976).

[Kaneda 80] 金田悠紀夫、小林康博、前川禎男、他：試作LISPマシンの高速化について、電子通信学会電子計算機研究会、EC79-58, pp. 71-79 (Jan. 1980).

[Kaneda 81] 金田悠紀夫、小林康博、前川禎男、瀧和男：コンパイラ導入による試作LISPマシンの効率改善について、情報処理学会論

文誌Vol. 22, No.2, pp.114-120(1981).

[Kaneda 82] 金田悠紀夫、前川禎男、瀧和男：直接マイクロコード生成形コンパイラによるLISPマシンの高速化、情報処理学会論文誌、Vol. 23, No.1, pp.96-99(1982).

[Kaneda 84-a] Kaneda, Y., Tamura, N., Wada, K. and Maekawa, S.: Sequential PROLOG Machine PEK Architecture and Software System, Proc. of International Workshop on High-Level Computer Architecture, 4.1, Los Angels (1984).

[Kaneda 84-b] 金田悠紀夫：Prologマシン、情報処理、Vol.25, No.12, pp.1345-1352 (1984).

[Knight 74] Knight, Tom:CONS, MIT AI Lab. Working Paper 80 (Nov. 1974.)

[Knight 81] Knight, Tom, et al.:CADR, MIT AI Memo 528, (Mar, 1981).

[Kowalski 74] Kowalski, R.:Predicate Logic as Programming Language, Information Processing 74, pp.569-574, North-Holland (1974).

[Kurokawa 76-a] 黒川利明：LISPのデータ表現、情報処理、Vol.17, No.2 (1976).

[Kurokawa 76-b] 黒川利明：LISP 1.9プログラミングシステム、情報処理 Vol.17, No.11, pp.1056-1063, (Nov. 1976).

[McCarthy 66] McCarthy, J. et al.:LISP 1.5 Programmer's Manual, MIT Press (1966).

[McCarthy 78] McCarthy, J.:History of LISP, ACM SIGPLAN Notices, Vol.13, No.8, pp.217-223(1978).

[Mizoguchi 83] Mizoguchi Fumio :Prolog Based Expert System, New Generation Computing, Vol.1, No.1, pp.99-104 (1983).

[Mizoguchi 84-a]溝口文雄、片山佳則、武田正之：論理型言語Prologの比較検討、Proc. of the Logic Programming Conference '84, 5-4, Tokyo (Mar.1984).

[Mizoguchi 84-b]溝口文雄：Prologによるエキスパートシステム、情報処理、Vol.25, No.12, pp.1386-1395 (1984).

[Moon 74] Moon, D.A.,;MACLISP Reference Manual, Project MAC, MIT, (Apr.1974).

[Morris 79] Morris, F.L.:A Time-and Space-Efficient Garbage Compaction Algorithm, CACM 20-10 (1979)

[Nakajima 86] Nakajima Katsuto, et al.:Evaluation of PSI micro-interpreter, Proc. of Compcon '86, To appear.

[Nakamura 84] 中村克彦：Prolog処理系、情報処理、Vol.25, No.12, pp.1329-1335, (1984).

[Nakanishi 77] 中西正和：LISP入門、近代科学社(1977).

[Nakashima 85] 中島浩、三石彰純，瀧和男：PSI の性能評価(2) 、情報処理学会第30回全国大会、No.1C-3 (Mar. 1985)

[Nishikawa 85] 西川宏、山本明、横田実、中島克人、三井正樹、吉田裕之：PSI の性能評価(1) 、情報処理学会第30回全国大会、No,1C-2 (Mar. 1985)

[Okuno 85] Okuno Hiroshi :The Report of The Third Lisp Contest and The First Prolog Contest, 情報処理学会研究会資料、記号処理33-4 (Sep.1985).

[Robinson 65] Robinson, J.A.:A machine-oriented logic based on the resolution principle, J.ACM. Vol.12, No.1, pp.23-44 (1965).

[Robinson 83] Robinson, J.A.:Logic Programming－Past, Present and Future, New Generation Computing, Vol.1, No.2 (1983).

[Shimada 76] 島田俊夫、山口喜教、坂村健：LISPマシンとその評価、電子

通信学会論文誌D.J59-D,6 (1976).

[Siewiorex 82] Siewiorex, D.P., Bell, C.G., Newell, A:Computer Structures－Principles and Examples, McGraw-Hill, pp.549-572 (1982).

[Takagi 83] 高木茂行、近山隆、横田実、服部隆：拡張制御構造のPrologへの導入、情報処理学会第26回全国大会、No.4D-11 (Mar. 1983)

[Takeuchi 78] 竹内郁雄：LISP処理系コンテストの結果、情報処理学会研究会資料、記号処理5-3 (Aug. 1978).

[Takeuchi 79] 竹内郁雄：第２回LISPコンテスト、情報処理、Vol.20, No.3, pp.192-199 (1979).

[Taki 78-a] 瀧和男、金田悠紀夫、前川禎男：LISPマシンの試作、情報処理学会研究会資料、計算機アーキテクチャ32-3 (Sep.1978).

[Taki 78-b] 瀧和男、金田悠紀夫、前川禎男：試作LISPマシンのインタプリータのマイクロプログラム化について、情報処理学会研究会資料、計算機アーキテクチャ33-2 (Nov. 1978).

[Taki 79-a] 瀧和男：LISPマシンシステムの研究、神戸大学工学部、システム工学科修士論文(Feb. 1979).

[Taki 79-b] 瀧和男、金田悠紀夫、前川禎男：試作LISPマシンとその評価、情報処理学会研究会資料、記号処理 7-3 (Mar. 1979).

[Taki 79-c] Taki, K., Kaneda, Y. and Maekawa, S.:The Experimental LISP Machine, IJCAI proceedings, pp.865-867 (Aug. 1979).

[Taki 79-d] 瀧和男、金田悠紀夫、前川禎男：LISPマシンの試作－アーキテクチャとLISP言語の仕様－、情報処理学会論文誌、Vol.20, No.6., pp.481-486 (1979).

[Taki 79-e] 瀧和男、金田悠紀夫、前川禎男：LISPマシンの試作－インタプリタの構造とシステムの評価－、情報処理学会論文誌、

Vol.20, No.6., pp.487-493 (1979).

[Taki 83] 瀧和男、西川宏、横田実、山本明、内田俊一：パーソナル逐次型推論マシンψ－そのアーキテクチャ、情報処理学会第26回全国大会、No,4N-2 (Mar. 1983)

[Taki 84-a] 瀧和男、横田実、山本明、西川宏、内田俊一：パーソナル逐次型推論マシンPSI のハードウェア設計、Proc. of The Logic Programming Conference '84, 8.1, Tokyo (Mar. 1984).

[Taki 84-b] Taki, K., Yokota, M., Yamamoto, A., Nishikawa, H., Uchida, S., Nakashima, H. and Mitsuishi, A.:Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI), Proc. of the International Conf. on FGCS 1984, pp.398-409, Tokyo (Nov. 1984).

[Taki 85] 瀧和男、他：PSI のハードウェア評価、ICOT Technical Report, To appear.

[Tamura 84-a] 田村直之、和田耕一、松田秀雄、金田悠紀夫、前川禎男：PROLOGマシンPEK について、Proc. of the Logic Programming Conference '84, 8-2, Tokyo (Mar. 1984).

[Tamura 84-b] Tamura, N., Wada, K., Matsuda, H., Kaneda, Y. and Maekawa, S.:Sequential Prolog Machine PEK, Proc. of The International Conf. on FGCS '84, Tokyo (Nov. 1984).

[Tanaka 84] 田中穂積、松本裕治：自然言語処理におけるProlog、情報処理、Vol.25, No.12, pp.1396-1403 (1984).

[Teitelman 74] Teitelman, W.:INTERLISP Reference Manual, Xerox (Feb. 1974).

[Terashima 85] 寺島元章：LISP－その発展の方向、情報処理、Vol.26, No.7, pp.711-720 (1985).

[Tick 84] Tick, E., Warren, D.H.D.:Towards a Pipelined Prolog Processor, Proc. of 1984 International Symposium on Logic Programming, Atlantic City (Feb. 1984).

[Uchida 83] Uchida, S., Yokota, M., Yamamoto, A., Taki, K., and Nishikawa, H.:Outline of the Personal Sequential Inference Machine PSI, New Generation Computing Vol.1, No.1, Ohmusha, Ltd. (1983)

[Uehara 74] 上原、松崎：マイクロプログラミングとその応用、産報、電子科学シリーズ54 (May. 1974).

[Uehara 84] 上原貴夫：CAD におけるProlog、情報処理、Vol.25, No.12, pp.1373-1379 (1984).

[Warren 77] Warren, D.H.D:Implementing Prolog －Compiling Predicate Logic Programs Vol.1,2, D.A.I . Research Report No.39,40, Dept. of Artigicial Intelligence, Univ. of Edinburgh (1977).

[Warren 80] Warren, D.H.D:An Improved Prolog Impremention which optimises Tail Recursion, Proc. of the Logic Programming Workshop, Hungary (July 1980).

[Warren 83] Warren, D.H.D:An Abstruct Prolog Instruction Set, SRI Technical Note 309, (Oct. 1983).

[Weinreb 80] Weinreb, D., Moon, D.:Flavors －Message Passing in the LISP Machine, MIT A.I. Memo No.602 (Nov. 1980)

[Yamaguchi 78] 山口喜教、島田俊夫：仮想計算機によるLISPプログラムの動的特性、電子通信学会論文誌D,J61-D, (1978).

[Yamamoto 84] 山本明、横田実、西川宏、瀧和男、内田俊一：逐次型推論マシンψのマイクロインタプリタ、情報処理学会研究会資料、記号処理29-7 (Oct. 1984)

[Yasui 82] 安井裕：LISPマシン、情報処理、Vol.23, No.8, pp.757-772

(1982).

[Yokoi 76-a] 横井俊夫：人工知能向言語（Ⅰ）、情報処理、Vol.17, No.7, pp.577-586 (1976).

[Yokoi 76-b] 横井俊夫：人工知能向言語（Ⅱ）、情報処理、Vol.17, No.8, pp.760-768 (1976).

[Yokota 83] Yokota, M., Yamamoto, A., Taki, K., Nishikawa, H. and Uchida, S.:The Design and Implementation of a Personal Sequential Inference Machine PSI, New Generation Computing, Vol.1, No.2, (1983).

[Yokota 84] Yokota, M., Yamamoto, A., Taki, K., Nishikawa, H., Uchida, S., Nakajima, K. and Mitsui, M.:A Microprogrammed Interpreter for the Personal Sequential Inference Machine, Proc. of the International Conf. on FGCS 1984 pp.410-418, Tokyo. (Nov. 1984).