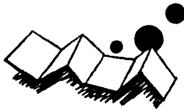
解 説



数式処理プロセッサ†

井 田 哲 雄†† 後 藤 英 一††,†††

1. はじめに

複雑な数式の整理，展開，微・積分等のできるシステムを数式処理システム*と呼ぶ．計算機による数式処理の試みは古く，1960年代より行われている．現在良く知られている数式処理ソフトウェアシステム，ALTRAN[1]，CAMAL[2]，FORMAC[3]，MACSYMA[4]，REDUCE[5]，SCRATCHPAD[6]等は，1970年代の前半に，その稼動が学術誌に報告されている．

数式処理システムの機能の向上につれ**，数式処理システムの対象とする分野も，プラズマ物理学，量子電気力学，電子幾何光学等の多岐にわたり，現行の計算機アーキテクチャでは，数式処理システム利用者からの，「より大規模な問題をより高速に」という要求に十分対処できなくなりつつある．

高度な数式処理の要求を満たすには，より良い言語翻訳，解釈処理系の製作，高速数式処理アルゴリズムの開発とともに，数式処理専用ハードウェアの製作が不可欠であると考えられる．

本稿では，前半で，数式処理向きシステムのアーキテクチャについて，現時点で試作の行われた数式処理向きプロセッサに関するいくつかの話題を含めて，論じる．後半では，筆者らが開発中の数式処理プロセッサ FLATS に言及する．

2. 数式処理ソフトウェアシステムの現状とその限界

前述の数式処理ソフトウェアシステム作成の動機をたどってみると，物理学，数学の研究者が，各々の研究分野で，限られた応用問題を解くのを目的としたものと，情報科学の専門家が，人工知能や記号処理の一つの課題として追求したものとがある．いずれも，システムの機能増強につれ，汎用数式処理システムへと発展してきたが，各々得意とする応用分野があるようである．これらのシステムの機能および性能の比較については文献8）が詳しい．

REDUCE の作成者ユタ大学の Hearn は，1978年秋に，理化学研究所において開催された「計算機による数式処理」シンポジウムで，利用可能性(availability)と維持能力 (maintanability)，システム作成者以外の使用実績の三つの見地から，現存する数式処理システムを評価し，前述システムのうち，一般の研究者に開放され，しかも，将来にわたって改良活動の行われるものは，MACSYMA と REDUCE であろうと述べている．

MACSYMA は，MIT の PDP 10 を ARPA ネットで利用するのが，主な使用形態である。REDUCE は，現在の代表的大型計算機で稼動するよう，移植性 (portability) を考慮して作られている．両者とも，我が国においても，使用できる状況にあるが，使用実績から見る限りでは，REDUCE のほうが，我が国においては，より広く普及しているようである．

システムインプリメンタにとって興味ある点は，両者ともに LISP をホスト言語にしたシステムである事である (MACSYMA は MACLISP を，REDUCE は Standard LISP (SLISP と略記する) をホスト言語とする)．数式処理においては，数式を表現するデータ構造が動的に変化する．データ構造の生成・消去を数式処理システムの一つのルーチンとして用意するかあるいはホスト言語の処理系に任せてしまうか，システム設計上，選択の余地がある．MACSYMA, REDUCE では LISP システムにデータ構造の生成・消去をゆだねている．

計算機アーキテクチャに要求される仕様は，両者と

† Architecture of Machines for Symbolic and Algebraic Manipulation by Tetsuo IDA and Eiichi GOTO (Institute of Physical and Chemical Research)††. (Department of Information Science University of Tokyo)†††.

†† 理化学研究所

††† 東京大学理学部

* 英語では，'formura manipulation' あるいは 'symbolic and algebraic manipulation' と通常呼ばれるが，後者の用語が，ACM の専門部会 (SIG) の名称でも用いられており，その使用が定着している．

** 計算機による数式処理の現状については参考文献7）で報告している．興味ある読者はそれを参照されたい．

も似かよっていると考えられるので，本章の以下の考察は，著者らが日頃使用しているREDUCEを念頭に置いて，進める．MACSYMA については，1978年より，隔年に，MACSYMAユーザ会議が開催されており，その現状については会議報告集より知ることができる．REDUCE の場合もユタ大学より REDUCE newsletter が1978年より年4回刊行されており，REDUCE の現状を知ることができる．

変数や関数を名前のまま扱うという意味では，数式処理は，記号処理である．さらに，数式処理の場合，ユーザにとっての最終的な解は，数値解であることが多く，数式処理によって得られた正確な数式解から，必要な精度にまで数値解を求める必要がある*．数値計算機としての機能を，数式処理プロセッサはあわせ持たなければならない．

数式処理システム使用者からの「大規模かつ高速処理能力を備えた計算機」の要求は，計算機に対する要求として，数式処理に限られるものではないが，この要求は，次の意味では，数式処理に特有である．

（1） 大きな論理アドレス空間と，問題の規模に応じた実主記憶量が必要である．（LISP プログラムは，一般に大きなワーキングセットを必要とする．数式処理では仮想記憶方式は，仮想／実・記憶比がある程度以上になると，実行速度が著しく低下し，その効力を失なう．）

（2） 基本操作が現行の計算機機械語命令と一対一に対応しないため，実行に時間がかかる．（たとえば，リストをたぐる操作であるCDRの実行には，現行計算機では数ステップから10ステップ程度の機械語命令を必要とすると思われるが，専用計算機を作れば，データ型の検査を含めて一命令（実行速度は2メモリサイクル）で実行可能である）．

したがって，

- ミニコンピュータ（16ビット語長）では，実用的な問題を解けない．
- 数式処理に要求される計算機との対話性が，TSS環境下では過大実記憶占有のため著しく阻害される．
- 大型計算機を用いても多くの CPU 資源の割り当てなくしては，有意義な数式処理はできない．

* REDUCE では，プログラムで ON FORT と指定すれば，FORTRAN のシンタックスで，数式を出力できる．しかし，FORTRAN に可変精度浮動小数点演算や任意多倍長固定小数点演算の諸機能が備わっていないと，せっかくのこの機能も有効に生かし得ないことになろう．

といった事態に我々は直面している．

3. 数式処理の基本操作とそのハードウェア化

数式処理プロセッサの実現は，既存ソフトウェアのハードウェア化がその一歩である．なかんずく，LISP の高速化が重要課題である．したがって，高級言語マシン開発の一連の研究の一つとして，位置付けることができよう．現在までのところ，バローズなどの一部計算機を除いては，高級言語（除く FORTRAN）向き計算機の研究は，価格対性能比（cost performance）の向上の追求が主である．数式処理の分野においても，LISP 向き計算機の研究が進んでいるが，絶対的な速度を，既存の大型計算機と競える実用規模の数式処理プロセッサの製作にまでは至っていない．しかしながら，あとで述べるような，研究目的の LISP マシンの製作も報告され，新しい LISP 向き計算機アーキテクチャの提案もあり[9),10)]，今後ともに，この方向での盛んな研究活動が行われていくものと思われる．

数式処理の高速化に本質的であると思われる処理の主なものに，機能的にみて，リスト処理，索表処理，任意多倍長，可変精度数値演算などがあり，実現手法からみると，実行時データ型検査，ハッシング，ガーベッジコレクションなどがあげられる．

3.1 リスト処理

ここでリスト処理と呼んでいるのは，リスト構造の生成，参照，変更の操作の総称である．より具体的にはリスト処理は記憶セルの読み書き，およびリスト構造を表現するタグのチェックの複合操作である．LISP における基本関数であるCAR, CDR, CONS, ATOM, RPLACA, RPLACD 等を数式処理プロセッサの機械命令として装備することが数式処理の高速化につながることは言うまでもないが，基本的ハードウェア機能要求としては，次の形で現れてくる．①主記憶の実効バンド幅を大きくすること．②リスト処理に使用されるタグの検出機構を設けること．ソフトウェア側からリスト処理を高速化するには，③リストセルの消費を抑える，④cdr-リンクを省略するリスト表現を考える，ことなどが考えられる．

①については，高速アクセスの記憶素子を用いること，記憶装置の読み出し幅を大きくすることである．主記憶の容量は，実用的問題を解くには最低数メガバイト程度は必要であるから，すべての主記憶にバイポーラの高速記憶素子を使用することは価格対性能比

の点から実際的でない．したがって，記憶システムに階層をもたせて，実効バンド幅を増大させるのが良いであろう．読み出し幅は，最低，一リストセル分あることが望ましい．後述するような並列ハッシングを行おうとするならば，その読み出し幅を広くとればそれだけハッシング探索の性能を向上させることができる．タグの検出機構は3.3で他のタグと合せて論じる．③は，より上位レベルのソフトアルゴリズムの問題であるのでここでは論じない．④については，Half-word LISP (HISP) として，知られている手法があり[11]，後述する MIT の CONS マシンも採用している．HISP では RPLACD のアルゴリズムが複雑になる．HISP のハードウェア化は，実行速度，記憶消費量，アルゴリズムの複雑化等を比較検討して，その採否を決定する必要がある．cdr リンクの省略によるリストセル用記憶量の節約は最大50％であり，HISP によるハードウェアおよびソフトウェアの複雑化に見合うものであるかは議論の余地がある．cdr リンクの省略と同一の効果は，ベクトル構造（一次元配列）[12] を利用することによっても得られる．前述の SLISP は，ベクトルを導入することにより，REDUCE の高速化を計画している．

3.2 ガーベッジコレクション

ここで言うガーベッジコレクション（以下 GBC と略記する）とは，一ユーザの使用する論理記憶空間上に，動的にとられたデータ構造（例えば，リスト，ベクトル）のうち，使用済になったものを回収し，使われていた領域を後の計算に再利用可能な状態にすることを言う．GBC のアルゴリズムの研究は，既に多くの研究者によってなされている．とりわけ，Deutsch の所要ハードウェアの考察を含めた既存の GBC アルゴリズムの比較検討が興味深い[14]．

標準的な方法のマーク・トレース・コピー法を，例にとって考察してみよう．まず，マークの仕方であるが，各セルにマーク用のビットを設けておく方法と，別にビット表を GBC 時に主記憶にとる方法がある．LISP で扱う基本データ構造（例えばリスト，ベクトル，ハッシュ表等）をどのように設計するかにもよるが，新アーキテクチャの計算機を考えるのであれば，ビット表法のほうが，次の理由により有利であろうと考えられる．

- マークビットは最低2回は走査されるが，表の形でビット列を保持しておいたほうが，アクセスの局所性がよい．
- コピー時にポインタ再調整を行う場合にもビット表を有効に使用できる．

ただし，ビット表法を実現するには，ビット表を効率良く用いる命令が必要になる．

並列処理 GBC (concurrent garbage collection) は，多くの研究者によって追求されている課題であるが，数式処理の場合．

- 実時間処理の必要性は，当面は，ない．
- 主記憶アクセスがシステムの隘路になっている場合，別個のプロセッサによって，GBC を行っても，単一のプロセッサによって，GBC を行っても，全体的な処理能力（スループット）はほとんど変化しない．

等の理由により，目下緊急の課題とはなっていない．

リストのみならず，任意長のデータ構造を扱う場合，空領域をポインタでつなぐ方法によるメモリ管理では，平均して，50％のメモリが遊んでしまうため[15]，コピー時のデータ構造の圧縮 (compactify) が不可欠である．この時ポインタ値の再調整が必要であるが，これはハードウェアを用いれば高速に行える[13]．

3.3 データ型の実行時検査

コンパイル時に，データ型を定めておくことは，実行時の能率向上に大きく寄与するが，数式処理の場合，コンパイル時の，完全なデータ型決定は不可能である．

LISP に関して言うと，SLISP を例にとっても基本的データ型は，LISP システムインプリメンタにとって，int（整数），flt（浮動小数点），str（ストリング）等7種類ある[12]．さらに，後述するハッシングを用いた連想処理ができるように拡張すれば，表-1 のように多くの型が存在することになる．ハードウェアレ

表-1 LISP におけるデータ型の一例（HLISP[20] の場合）

atom	constant	int (integer)	
		flt (floating number)	
		str (string)	str
			hstring (hashed string)
		funcp (function pointer)	
		vector	vector
			hvector
	id (identifier)	AMT	AMT (Associative Membership Table)
			HAMT (Hashed AMT)
		CAT	CAT (Content Addressed Table)
			HCAT (Hashed CAT)
pairs			

表-2 データ型と記憶内での標識法

方法	長所	短所	使途の例
語内タグビット	高速	語内ビット損	pairs 終端マーク
語内タグフィールド	高速	語内フィールド損	データ型標識
アドレス領域分割	高速	領域再配置が複雑	
別タグビット表	ビット群演算の高速化可能，語内ビット損なし	1語に対するタグビット検査が遅い	ガーベッジのマッキングとデータの再配置
データブロックヘッド	語内ビット損なし，データ領域がメモリを見るだけで分る	語外ワード損，間接アドレスによる速度低下	ページより小さな単位でデータの再配置を行う
ページ・マップテーブル	語内ビット損，語外ワード損なし	間接アドレスによる速度低下，ページ単位でのみ可能	仮想記憶の管理

ベルでは，さらに，通常プログラマから見えないデータ型（例えば int はハードウェア内部表現では，sint（通常の整数）と bint（大きな整数）とに分れる）が存在する．

データ型を計算機内で表現する代表的方法を列挙すると表-2 のようになる．実用的数式処理システムを実現するには，これらの方法はすべてハードウェアでサポートする必要があろう．

なおタグを付けた計算機アーキテクチャは古くから提唱があるが[16),17)]，現在の計算機アーキテクチャの主流とはなっていない．タグ使用に対する反論として，速度の低下，ビット損失をあげられるが，前者については，表を利用した簡単なハードウェアで実現でき（一例としては文献 18）を見よ），後者についても，浮動小数点の指数の一部の値を，タグフィールド用に使用するなどの方法[19)]をとれば，損失は小さい．後者の方法は，現在米国ですすめられている浮動小数点の標準化の一案ともあいいれるものとなっている[31)]．

3.4 ハッシング

従来，ハッシングはアセンブラやコンパイラの記号表の高速参照法として利用されてきた．リスト処理においても，同様に，アトムを生成する過程で，ハッシングは古くから使用されてきている．これらの応用に加えて，数式処理の高速化に貢献する手法として，ハッシングによる連想処理用データ構造を利用した多項式演算法がある[20)]．

この他にも，筆者らの研究グループにより，連想計算法[21)]，スパース行列演算，表を用いた高速多分岐ジャンプ法，属性リストの参照，集合演算[22)]へのハッシングの応用が挙げられている．筆者らの他にも，同様分野のハッシングの応用として，ハッシュ行列[23)]，ハッシュリンキング[24)]，高速パターンマッチング法[25)]などがあげられる．

このような事情から，ハッシングの高速化は，数式処理，記号処理の高速化に著しく寄与するものと思われる．ハッシングのハードウェアによる高速化法として，メモリ・バンクの並列動作性を利用した並列ハッシングが著者らにより提案されている[26)]．

3.5 任意多倍長・可変精度演算機能

任意多倍長演算とは，整数を自動的に必要な桁数まで求めることを言い，可変精度演算とは浮動小数点数を必要な精度まで求めることを言う．

任意多倍長演算機能の欠けた数式処理システムは存在意義がうすい．可変精度演算機能なくしては，数式処理システムの数値演算能力を大きく減ずることになる．いずれの演算の場合も，「非常に大きな数」に至るまで同一速度で高速に実行するハードウェアの実現は困難である．実際には単精度（あるいは倍精度）演算機構，データ型検出機構，リスト処理機構を併用して，演算を行うことになる．まず，データ型をチェックし，被演算数が単精度ならば，標準装備の演算回路により，高速に行う．結果はデータ型検出機構で再び検査し，適当なタグを付す．もし，被演算数が単精度でないならば，割出し機構を経由して，マイクロプログラムあるいはソフトウェアにより演算を行う．一語で，納まらない結果はリスト（あるいはベクトル）構造で表現する．

通常は単精度数の出現頻度が高いため，単精度数の検出⇒単精度数どおしの演算のパスが高速であれば，全般的な，演算速度の低下はほとんどない．

3.6 その他の数式処理高速化の方策

以上述べた他に，ハードウェア化が LISP 高速化をもたらすものに，スタック操作，引数の受け渡し・チェックの合理化，配列のアクセス範囲のチェック等がある．

スタック操作の高速化は LISP や ALGOL 等の再帰的呼出しを許す言語の共通の課題である．コンパイルされた変数の割当てや，計算の中間結果の格納にふつうは，スタックをフレームと呼ぶブロックに分割して用いる．人工知能のように複雑な制御構造が必要とされる場合は，スパゲティスタックと呼ばれるようなスタック[27)]が広く用いられているが，数式処理の場合，LIFO (Last In First Out) フレームスタックで十分である．更に LISP コンパイラ (SLISP の場合) では，変数のスコープ規則を簡単化し，テキスト上でのブロックの多重化を許さず，純局所的変数か，さも

なくば大局的（あるいは流動（fluid*））変数かに分けてしまえば，フレーム構造が簡単になる．一サブルーチン内では，現在のフレーム以外の局所変数の参照を許さないことになるため，スタック管理のために，現在のフレームの先頭を示すポインタレジスタのみあればよい．スタックはフレーム単位にみれば，ストレジの確保，放出は LIFO であり，参照の局所性も高いため，ポインタ管理のハードウェア化に加え，高速スタック用メモリを用いれば，スタック操作が高速化される．今までにインプリメントされた LISP マシンも，このような考え方を取り入れたものがある[28),29)]．

今まで論じてきたスタックは値の一時格納用ストレジとして，使用されたものであるが，この他にも制御情報（例えば，サブルーチンの戻り番地，前述スタックフレームポインタ値）や，流動変数の値束縛（binding）にともなう旧値の保存にもスタックは利用される．これらの利用法を一つのスタックで実現することもできるが，後者の利用は純 LIFO であることや，エラー時のトレースの容易さなどを考えると，後者の使用については，別のスタックで行い二つのスタックを併存させることも考えられる．

引数の受渡しは LISP に限って言えば call by value であるため，前述のフレームスタックを用いれば，簡易に実現でき，余分のハードウェアは不要である．FORTRAN については，call by address であるため，間接参照（indirection）を指示するようなビットを語内に設けることにより，統一的に引数の束縛を行うことができよう．間接参照ビットの検出機能は，前述のデータ型検出機構に持たせれば良いであろう．

配列の範囲のチェックは現在の計算機では，インデックスチェック用のコードを生成することによって行っている．このチェックは本来ならば，配列の参照と並行して行えるものである．FORTRAN のプログラマの通常行っている方法はデバック時には，範囲チェックコードを生成し，プロダクションランでは，速度を上げるためこれらのコードをはずしている．本来，正しい答を得るべき時に，保護をはずすというのは危険極まりないことである．また，現在のプログラムの正当性証明システムが対処できるのは，限られた問題である．完全なアクセス範囲のチェックは原理的に不可能である．LISP システムのように，ポインタを含め，システム全体の保護機構が完全でないと，GBC，リストの割当等がうまくいかない．したがって，配列のアクセス範囲チェックは不可欠である．速度の低下をもたらさずに，アクセス範囲チェックを行うには，アドレス形成の回路とともにアクセス範囲比較を行う回路を付加する必要がある．

* 名前と値を格納するストレジとの対応がプログラムテキストのスコープ規則のみによらず，動的に定まる変数を SLISP では流動変数と呼んでいる．

4. 実用的数式処理プロセッサに向けての研究—その動向—

ハードウェアを含めた数式処理システムの研究は，米国では MIT（人工知能研究所），ユタ大学，ハワイ大学，日本では筆者らのグループが理化学研究所で行っているものが主なものである．数式処理プロセッサの開発は LISP マシンの開発と重複するところが多いので，LISP マシン研究を含めると米国の BBN (Bolt Beranek and Newman, Inc.)，ゼロックス PARC (Palo Alto Research Center) の Interlisp 用 LISP マシン，我が国でも，京都大学[29)]，神戸大学[18)]のものなどがある．この他にも，現時点では，数式処理システム作成の動きと直接結びついていないが，カリフォルニア大学 (Berkeley) のグループによる浮動小数点表示，演算の合理化，標準化の動きがある[30),31)]．

本誌においても，別に LISP マシンの編集計画があるとのことであるので，本稿では，MIT，ユタ大学，BBN, PARC の計画の概要を本節以下にまとめる．個々のマシンの詳しい内容は参考文献を参照されたい．

4.1 MIT CONS[28)]

経緯：このシステムは，MIT 人工知能研究所の Greenblatt, Knight らが開発したもので 1973 年より開発が始まり，1979 年現在複数台が稼動している．人工知能・数式処理システムのプログラム開発，教育に利用されている．CONS 製作の経験を基に，現在は CADR[32)] と呼ぶ次機種の設計および，超 LSI による LISP マシンの実現計画が MIT においてすすめられている[40)]．

目標：MACLISP や人工知能研究に用いられる新しい実験的諸機能をもった LISP の稼動するパーソナル LISP マシンを実現する．

動機：既存の計算機のアーキテクチャは LISP には不向きである．

- 実記憶を多用する大きな LISP プログラムは TSS の環境下では，満足な応答性が得られない．
- 記憶空間が不足

実現法：・パーソナル計算機として普及しうる値段で，完全マイクロプログラム計算機を製作する．

・主記憶は他のプロセッサと共有しない．ただし，ファイルは共有する．(CONS プロセッサは Ethernet[33] により PDP 10 システムに接続している．)

4.2 BBN Interlisp-11[34]

目標：Interlisp-10 (PDP-10 で稼動する Interlisp) と完全にコンパチブルな，デスクトップパーソナル Interlisp マシンを実現する．

動機：・PDP 10 の仮想アドレス空間では人工知能の多くの応用には不十分．

・Interlisp実行のほとんどは限定された命令シーケンスによっており，マイクロプログラム化によって，高能率化（価格対性能比の向上）が計れるはずである．

・TSS の環境下では，満足な応答性が得られない．(CONS と同じ)

実現法：・PDP 11/40 に WCS を付加し，マイクロプログラムにより，LISP 向きマイクロ命令を実行する．

・メモリマップ (22 bit 仮想アドレス→18 ビット実アドレス）を装備し，広い仮想アドレス空間を実現する．

・テレタイプインタフェースで PDP 10 と接続しファイルを共有する．

4.3 PARC ALTO[14]

目標・動機：ByteLISP プロジェクトの一環として，製作された．製作の動機は前述二つのプロジェクトとほぼ一致する．

実現法：実験的なマイクロプログラムミニコンピュータを製作し，Interlisp を稼動させる．なお，このシステムに関しては，性能の評価が細く行われており，今後の LISP マシン設計に有意義なデータを提供している．

4.4 ユタ大学 REDUCE/1700[35]

目標：大規模 LISP システム (特に REDUCE) を小さな計算機においてインプリメントする上でのデータ表現，コード表現をコンパクトにかつ効率良くするための方策を探る．REDUCE のためのポータブルな抽象的 LISP マシンを定義する．

実現法：バローズ B1726 を用い，いくつかのソフトウェア（ファームウェア）レベルでの抽象的機械を定義し，ソフトウェアの階層構造により，上記の目標を実現している．特別なハードウェアの付加はない．

全者に共通していることは，いずれも，LISP 向きの新たなアーキテクチャを追求する実験的なシステムであることである．LISP 言語自身もまだ発展しつつあり，新しい機能を導入しつつ，システムを発展させるため，ソフト（ファーム）ウェアに多くを依存したシステムになっている．つまり，順次処理的なアルゴリズムの主要部をより早い論理素子で実行することにより，また，高速レジスタファイル活用により，記憶実効アクセス時間を減少させ，従来より高速化をはかる手法である．並列処理による高速化は，データ型のチェックなどごく一部分である．

しかしながら，いずれのシステムも速度の向上にすぐれた結果をもたらしており，現在の主流計算機アーキテクチャと LISP/数式処理マシンのアーキテクチャとの不整合度がきわめて大きいことを如実に示している．

5. FLATS マシン

5.1 FLATS 概説[10]

FLATS は理化学研究所において，設計 (54 年度 1 部製作) 中の数式処理プロセッサである．数値計算，記号処理およびハッシングを利用した連想処理を高速に行うことを目ざした専用プロセッサである．高級言語マシンの観点から言えば，BFORT, HLISP マシンである．ここで BFORT とは，通常の数値演算用言語としての FORTRAN に加えて，3.5 で述べた拡張算術演算機能を付加した言語である．HLISP はハッシングによる索表・連想機能を付加した LISP で筆者の一人（後藤）らが開発したものである．FLATS マシンのため HLISP は現在改良中（詳細仕様については文献 20) 参照）であるが，新 HLISP は Hearn らの SLISP をサブセットに含み，REDUCE のホスト言語となる．数値計算機能をサポートすることは，前に述べた理由で不可欠である．

FLATS は，中・大型計算機のバックエンドプロセッサとなりうるよう設計を進めている．したがって，FLATS 自体は，アレイプロセッサにみられるように，単一課題をできる限り外部からの割込みのない状態で高速に実行することを目標とする．

計算の高速化には，論理素子の高速化は言うまでもないが，最終的には並列処理を導入することによるしかない．しかしながら，数式処理で扱う多くのアルゴリズムは並列処理を活用することは困難である．例え

ば，因数分解，積分等で頻繁に使用する，ユークリッド互除法のアルゴリズムでは，常に商の剰余を順次に計算しなくてはならず，現在のアルゴリズムでは並列処理を導入することはできない．リスト処理，任意多倍長計算，可変精度の浮動小数点計算にしても並列処理を導入することは困難であり，命令実行サイクルの高速化が全体の高速化のきめ手となる．では，まったく，並列処理を活用できないかと言うとそうではない．リスト処理では，最低限一リストセルを読み出す記憶アクセスの並列処理は必要である．さらに，ハッシングでは鍵を探す操作に簡単な比較回路を内蔵した記憶バンクを用い，バンクの並列読み出し動作を利用すると，索表操作の能率が大幅に向上することが知られている．FLATS は，本質的には SISD (Single Instruction stream Single data stream)[36]の計算機であるが，記憶読み出しの並列性は最大限活用する予定である．

FLATS においては BFORT および REDUCE がユーザとのインタフェース言語であるが，索表および連想機能を次のような形で提供する．

索表計算： ①頻繁に使用される関数の代表点における関数値を高速記憶に格納しておき，補間ハードウェアを使用して，高速に関数計算を行う．例えば $\sin x$, $\log x$, $1/x$, $\sqrt{x}$ 等の計算はこの手法を用いる．②変数値が記号，整数あるいは複雑なリスト構造等の時は①の方法は利用できない．そこで，変数値と関数名をハッシングの鍵に，関数値を鍵に付けた値としてハッシュ表に登録しておき，同一引数での繰り返し関数評価を避ける．

この機能はユーザにとって直接の効果はスピードアップとしてしか現われないが，②の索表計算法については，ユーザからのモード指示を受けて行うことも考えられる．

連想機能： ハッシング鍵がシステムにユニークに存在することを活用して，同一構造の高速同定に用いる場合と，ハッシュ表を単一ヒットの連想メモリとみなして，連想読み出しを行う場合と二とおりが，ここで言う連想機能である．多項式の演算はこの機能を用いると，高速に行えることがわかっているが，ユーザインタフェースにどのような形で連想機能を提供するかは今後のソフトウェア設計上問題である．

5.2 FLATS アーキテクチャ上の諸選択

1. 記憶階層

CPU の速度と主記憶とのサイクル時間に大きなひらきがある場合キャッシュメモリが必要となる．CPU の速度にみあう記憶素子で主記憶全体を構成するのも一法であるが，現在広く使われている低速 MOSRAM と高速バイポーラ RAM との価格比が 1：50 程度もありしかも，数式処理では最低数メガバイトのメモリ容量が必要であることから，キャッシュを用いた記憶階層構造を設けることとする．FLATS では4種のキャッシュメモリを使用する．命令（インストラクション）用空間，リスト，ベクトル，データバッファ等のデータ用空間，フレームスタック空間，コントロールスタック空間に対するキャッシュである．4種類のキャッシュメモリのアクセス法に違いがあるため，キャッシュメモリ管理の方法は異なる．

2. 命令形式とオペランド指定法

命令形式としては 0, 1, 2, 3 アドレス法が比較検討の対象である．さまざまな制約条件下でどの命令方式が良いか，断定的結論はなく，現在に至るまで，計算機アーキテクチャ専門家の間で議論が続いている[37]．FLATS では，まず，一つの制約条件である．オペレーションコード (OP) 圧縮は特に追求しないこととし，さらに一つのフレーム内アドレスは $0\sim2^7-1$ までとし op a_1, a_2, a_3 (各8ビットで一語 32 bit) の3アドレス法を採用する．つまり，一つの命令語で $a_1 :=$ op(a_2, a_3) を実行する．3アドレス法は 0, 1, 2 アドレス法を包括している．3アドレス法実現上の問題点はスタックフレームに二つの読出し，ポートが必要である点にあるが，これは，書込みを共通とし，読出しが独立にできる，スタックフレーム用キャッシュメモリを二つ装備することによって対処できる．なお a_i $(i=1, 2, 3)$ を指定する8ビットのうち最上位ビットで使用中のフレームと汎用レジスタ (General Register) のいずれかを指定することにすれば，IBM 360 系の汎用レジスタスキームをも包括することになる．

3. タグアーキテクチャ

表-2 のデータ型識別法はすべて FLATS で用いられる．MIT CONS は，記号処理におけるタグアーキテクチャの適切性を実証しており，FLATS においても，タグアーキテクチャを積極的に取り入れる．タグの種類は，ソフトウェアにも依存しており，タグの変更に対処できるようなハードウェアの設計が要求されよう．

4. ハードウェアによる LISP のサポート

3章で述べた LISP 高速化ハードウェア機構で FLATS で実現するものは

（1） 実行時データ型検出
（2） ハードウェア・スタック
（3） 高速記憶
（4） 主記憶空間のセグメンテーション
（5） 配列の境界検査機構
（6） LISP 基本関数である CAR, CDR, CONS, ATOM 等のハードウェア解釈実行機構

である．

このうち，（3）は前述のキャッシュメモリによって実現できる．（4）は論理空間の静的分割（FLATS では論理空間が $0 \sim 2^{24}-1$（ワードアドレス）の4種の空間に分割され，さらに各空間が16個のセグメントに分割される）とセグメント単位でのデータ型の割付けを指す．（5）は，ベクトルのアクセス範囲のチェックである．（6）に関しては，簡単な命令は，2マシンサイクルで実行されるが，複雑な命令はマイクロプログラム，および割出し処理ルーチンによる解釈実行を併用して，実行する．

5. ハッシングハードウェア

筆者らの提案している並列バンクハッシング[26]をFLATS では実現する．ハッシングハードウェアは，ハッシュ番地列生成機構，ハッシュ操作制御機構，およびハッシュ表部の三つに分れる．このうち前二者はFLATS 中央演算処理部に組み込まれる．ハッシュ表は前述データ用空間にとられ，ハッシュ鍵の比較はこの空間の持つキャッシュメモリの読出しと同時に行われる．数バンク幅分の並列読出し機能およびハッシュ鍵の比較，削除語，空語を示すビットパターンの検出回路をキャッシュメモリ内に設ける．比較・検出回路からの出力はこのキャッシュをアクセスする際には常に得られるものであるが，ハッシングによるアクセス命令のときにのみその出力は意味を持つ．

6. ガーベッジコレクタ

FLATS ではビット表を用いた圧縮 GBC 法[13]を用いる．ビット表は GBC 起動時に主記憶空間にとられる．この方法を高速化するために，ビット列を扱うための特別な命令を用意する．（例えば，ある領域に存在する'1'の計数を行う）トレース・マーク時にビット表の1ビットで，1セルの要／不要の対応がつけられ，さらに，圧縮コピー時にこの表をみて，ポインタの再調整を行う．

7. 主記憶・二次記憶の管理

ページング機構を採用し，論理空間から実空間へのマッピングを行う．マッピングはハッシングを用いる．

二次記憶のアクセスを主記憶のアクセスと同程度の時間で行うよう，仮想テープ[38),39)]の考え方*をFLATSでは採用する．概して順次的アクセスの多い大型行列の演算などは，行列要素を仮想テープにとることにより，あたかも，大容量行列がすべて主記憶に存在するがごとくに行えるようになる．

6. おわりに

数式処理プロセッサの開発は，従来の計算機システムの開発に比べ，より「ソフト的」である．言語・アルゴリズムが先にあり，それをいかにハードウェアで高速化するかということである．初期の計算機システムの開発が，始めに，ハードウェアがあり，しかるのちにソフトウェアを考えたというのと，極めて，対照的なものとなっている．数式処理・人工知能の分野は今後も開発すべきアルゴリズムが数多くあり，数式処理プロセッサのハードウェアもある程度これら進歩に追随できる柔軟さが要求されよう．

参考文献

1) A. D. Hall Jr.: On ALTRAN System for Rational Function Manipulation—A Survey, CACM, Vol. 14, No. 8, pp. 517-521.
2) J. P. Fitch: CAMAL User's Manual, University of Cambridge, England (1975).
3) K. Bahr: Toward a Revision of FORMAC, SIGSAM Bulletin, Vol. 18, No. 1, pp. 10-16 (1974).
4) MIT The MATHLAB Group: MACSYMA Reference Manual, Laboratory for Computer Science, MIT, Massachussetts (1974, 1976, 1977).
5) A. C. Hearn: REDUCE-2 User's Manual, University of Utah, Utah (1973).
6) R. D. Jenks: The SCRATCHPAD Language, SIGSAM Bulletin, Vol. 8, No. 2, pp. 20-30 (1974).
7) 後藤英一，佐々木建昭：計算機による数式処理の現状，情報処理，Vol. 18. No. 8, pp. 830-837 (1977).
8) D. Barton and J. P. Fitch: A review of algebraic manipulative programs and their application, The Computer Journal, Vol. 15, No. 4,

* 二次記憶のファイル空間をテープに似た擬似順アクセス空間（仮想テープ）として，組織化し，必要に応じて生成する仮想的なヘッド（実体は一次記憶上のバッファ）を通してアクセスする．一次・二次記憶装置間のデータの転送は，アクセス法をこのように制限することにより，先行制御が可能となり，必要とするファイル上のデータは一次記憶並みの速度でアクセスできる．

pp. 362-381 (1972).

9) R. L. Williams: A multiprocessing system for the direct execution of LISP, Ph. D. dissertation, University of Illinois at Urbana-Champaign (1978).

10) E. Goto et al.: FLATS, a machine for numerical, symbolic and associative computing, Proc. 6 th Annual Symposium on Computer Architecture, pp. 102-110 (1979).

11) W. L. Van der Poel: A manual of HISP for the PDP-9, Technical U., Delft, Netherland.

12) J. B. Marti et al.: Standard LISP Report, UUCS-78-101, University of Utah (1979).

13) M. Terashima and E. Goto: Genetic order and compactifying garbage collectors, Information Processing Letters, Vol. 7, No. 1, pp 27-32 (1978).
他の GBC 法に関する文献は，本引用文献を間接ポインタとして，たどることができる.

14) L. P. Deutsch: Experience with a microprogrammed Interlisp system, Zerox Palo Alto Research Center (1979).

15) D. E. Knuth: The Art of Computer Programming, Vol. 1, pp. 445-446, Addison-Wesley Pub. Co. (1968).

16) J. K. Iliffe: Basic Machine Principles, McDonald Computer Monographs, London (1968).

17) E. A. Feustel: On the advantage of tagged architecture, IEEE Transactions on Computers, Vol. C-26, No. 2, pp. 112-125 (1977).

18) K. Taki et al.: The experimental LISP machine, Proc. 6 th IJCAI, pp. 865-867 (1979).

19) T. Ida and E. Goto: Overflow free and variable precision computing in FLATS Journal of Information Processing, Vol. 1, No. 3, pp. 140-142 (1978).

20) E. Goto and Y. Kanada: Hashing lemmas on time complexity with application to formura manipulation, Proc. ACM-SYMSAC, pp. 154-158 (1977).

21) E. Goto: Monocopy and associative algorithms in an extended LISP, Technical Report 74-03, Information Science Laboratory University of Tokyo (1974).

22) M. Sassa and E. Goto: A hashing method for fast set operations, Information Processing Letters, Vol. 5, pp. 31-34 (1976).

23) W. Teitelman: Interlisp Reference Manual, Xerox Palo Alto Research Center (1978).

24) D. W. Bobrow: A note on hash linking, CACM, Vol. 18, No. 17, pp. 413-415 (1975).

25) R. Cowan and M. Griss: Hashing, the key to rapid pattern matching, Proc. EUROSAM (1979).

26) T. Ida and E. Goto: Performance of a parallel hash hardware with key Deletion, Proc. IFIP Congress, pp. 643-647 (1977).

27) D. G. Bobrow and B. Wegbreit: A model and stack implementation of multiple environments, CACM, Vol. 16, No. 10, pp. 591-603 (1973).

28) R. Greenblatt et al.: The LISP machine, Artificial Intelligence Laboratory, MIT (1979).

29) M. Nagao et al.: LISP machine NK 3 and measurement of its performance, Proc. 6 th IJCAI, pp. 625-627 (1979).

30) W. Kahan: A proposed standard for floating arithmetic, University of California at Berkeley (1978).

31) J. T. Coonen: Specifications for a proposed standard for floating point arithmetic, University of California at Berkeley (1978).
文献 30), 31) に関しては一松による解説が本誌 20 巻 7 号 (pp. 793-797) にある.

32) G. L. Steele Jr.: CADR (1979), MIT AI. Lab, Working Memo として発表予定.

33) R. M. Metcalfe and D. R. Boggs: Ethernet: Distributed packet switching for local computer networks, Xerox Palo Alto Research Center (1975).

34) A. K. Hartley: Interlisp-11, A personal LISP machine, BBN (1979).

35) M. L. Griss and R. L. Kessler: REDUCE/1700: A micro-coded Algebra system, Proc. 11 th annual microprogramming workshop, pp. 130-138 (1978).

36) M. Flynn: Some computer organizations and thier effectiveness, IEEE Transactions on computers, Vol. C-21, No. 9, pp. 948-960 (1972).

37) 例えば，ACM SIGARCH Vol. 6, No. 3, No. 5 を参照のこと.

38) M. Sassa and E. Goto: "V-tape", a virtual memory oriented data type, and its resource requirements, Information Processing Letters, Vol. 6, No. 2, pp. 77-82 (1977).

39) K. Itano and E. Goto: Realization of a processor with virtual tapes and its evaluation, Journal of Information Processing, Vol. 2, No. 2, pp. 65-71 (1979).

40) G. L. Steele Jr. and G. J. Sussman: Design of LISP-Based Processors or, SCHEME; A Dielectric LISP or Finite Memories Considered Harmful or, LAMBDA: The Ultimate Opcode, MIT AI Memo No. 514 (1979).

(昭和 54 年 9 月 12 日受付)