記号、数式処理向き計算機FLATSの構想

後藤英一、井田哲雄、相馬嵩（理化学研究所）

1. はじめに

FLATSマシンは、数式処理をはじめとする記号処理、アルゴリズムを、高速度で実行することを目的とする計算機である。FLATSとはFortran、Lisp、Algol 60、Tuple（順序付れ組）、Tree、Setの頭文字の綴り合わせである。[1] 各文字で示されるプログラム言語（あるいはデータ構造）の存在意義は、次のとおりである。

(1). Fortran, Algol 60で書かれた科学計算用プログラムとアルゴリズムは多数ある。また、数式処理システムによって導出された数式も、多くの場合、これら、数値計算用言語を用いた数値計算によって、最終的な結論を得る必要がある。従って、これら言語で書かれたプログラムが"まともな"速度で実行されることは、数式処理向き、計算機の必須条件である。

(2). Lispで書かれた記号処理、AI（人工知能）プログラムは、数多くある。FLATSシステムで作動を意図する数式処理システム、REDUCEは、LispSで書かれている。

(3). Tuple, Setは、ハッシュ符号法を利用して実現するデータ構造である。その実現法は文献[2, 3,]にある。多項式に関する諸演算は、Setを用いると高速に実行できることが、純ソフトウェア(HLISP-REDUCE2)でも実証されている。更に、後述するハッシュハードウェアを用いれば、より高速化が、可能である。

(4). Treeは、記号処理で用いられるデータ構造のうち、最も基本的である。

これらのFLATSマシンの目標を実現するには、通常の計算機アーキテクチュアでは不十分である。筆者らは論文、[4, 5, 6, 7,]に、数式処理に要求されるハードウェアを提案してきたが、本稿では、それらをFLATSシステムとしてまとめあげる上での、ソフトウェア、ハードウェアを包括した設計思想について論ずる。

ハッシュハードウェア、ハードウェアスタック、タグ付データ語とその解釈ハードウェアは、FLATSシステムを特徴づけるものである。後二者は、いくつかの研究用、商用計算機に[14, 15]、すでに付設されているものであるが、FLATSでは、ソフトウェアを含めて検討を加え、高速かつ単純な、実現法を考案した。

2. ハッシュハードウェア

2-1. ハードウェアの構成

図1にハッシュハードウェアを示す。[6]、このハードウェアは主記憶をいくつかの記憶バンクに分けて、各バンクを並列に動作させることにより、ハッシングにおける、鍵読み出し、鍵の比較の回数を大幅に減少させることを目的としたものである。このハードウェアはFLATS主記憶と中央演算装

置（CPU）とのインタフェイス部に付設されるものである。ハッシュ表用記憶は、主記憶空間にとられ、通常の記憶としても使用される。図1におけるハッシュ番地生成器（HAG）はCPUとの並列処理が不要であれば、CPUの算術論理回路を利用することによっても実現できる。

このハードウェアでは、ハッシュ関数列の計算は高速に行うことができ、しかも、ハッシュ表アクセスと並列に行える。従って、最終的にハッシュ演算の速度を決めるものは、記憶アクセスの回数である。表1、2に、最悪条件にでの平均探索回数を示す。この結果から、ほぼ一間接アドレスに匹敵する速度でハッシュ演算を行えることがわかる。

スース、ハッシングの応用

前述ハードウェアは、固定長鍵を前提とする。可変長鍵は図2のように、表わす。

今迄に提案されているハッシングの応用を分類すると次のようになる。

(1). 連想処理

(2)、唯一性を保証されたデータ構造の実現と、それに伴う同一性チェックの高速化

(3) ポインタChasingの高速化、不要なポインターの節約

(1)については、アセンブラ、コンパイラの名前表、高速検索が、代表的である。つまり鍵を与えて、鍵に付随する値を取り出す操作である。鍵の種類によって、(その作成法は後述)、数多くの応用が考えられる。後藤ら、[8.9.10]の作成したHLISPでは、ASSOCCOMP、TABGG、連想子による連想検索の機能が付加されている。ASSOCCOMPとは、鍵として、評価を受ける関数名と実引数リストの対を、値には関数の評価値をとり、同一関数を同一パラメータで何度も評価するような関数評価手続、(例えば、ACKERMANN、関数評価)を高速に行うものである。TABGGは、多分岐ジャンプを高速に行うものである。LISP 1.5におけるSELECT[(1)]でジャンプ先を割り出すことを除いて、機能的には類似する。[10]、鍵は分岐の際に満たすべき条件値で、対応する値は分岐先アドレスを持つ連想子が、システムにより自動的に作成される。分岐をサブルーチン呼び出し、トラップ等にすることにより、TABGGのいくつかのVariationが考えられる。(2)については、FLATSではSET、およびhashed LIST、(Tuple)が基本的データ構造である。より一般的にはn個の鍵、$k_0, k_1, \cdots, k_{n-1}$が与えられたとき、新たな鍵$k^{(0)}$を生成する関数を、hconstruct$(k_0, k_1, \cdots k_{n-1})$とすると、ハッシングを用いて、$k^{(0)}$を決定すれば、$k_0, k_1, \cdots, k_{n-1}$によって$k^{(0)}$は一意的に決定する。したがって、hconstructで作成される、データ構造の比較が$O(1)$で行えるとともに、記憶の節約が計れる。$n=2$でk_0, k_1の順序を問題とするとき、この関数をHCONSとFLATSでは、呼んでいる。SETは要素の重複のない順序なし集合を示すデータ構造である。(3)は、(1)、(2)を用いた連想処理の一変形と言える。属性リストの連想子を用いた高速化は、ポインタchasingの高速化と見なすことができる。Bobrowの、提案したhash-linking[18]も、このはんちゅうにはいる。

3. スタック

3-1. ソフトウェアからみたスタック

Valueスタック(V-スタック)とControlスタック(C-スタック)を設ける。[12] 前者はコンパイルされた変数が割当てられる領域である。後者は、主として、制御情報(例えばサブルーチンの戻り番地、旧フレームポインタ)の保存に用いる。コンパイルの容易さ、backtraceの効率化のためには、subroutine/procedure/function ごとに、frameと呼ぶ単位で、V-スタックを分割して、frameの先頭からの相対位置でもって、スタック上の変数を参照する方法をFLATSでは、基本とする。スタックの先頭からの相対番地で、すべてのスタック上の変数をアクセスする方式で、Fortran、Lispプログラムをコンパイルすることも可能である。[12] このための命令語セットをもFLATSは有する。

Algol60のようなブロック構造を持った言語では、非局所的変数を効率良く参照には、frameとframe先頭からの相対位置をもって行うのが通例である。[13] この場合、ブロックの静的なネストの数だけ、frame先頭ポインタのセット(display)が必要となる。この方式では、displayの管理を専用ハードウェアで行わないとすれば、ブロックの出入りにおけるオーバーヘッドを増加させる可能性がある。最も汎用的と思われる変数参照法は動的なフレーム番号と相対位置の対で表わされる変数名を鍵とするハッシュ表を実行時に保持する方法である。(この方法はLisp1.5の、A-listの高速化である。) この他の方法としては、トップレベル(グローバル)のframeと純局所的なフレームの参照で、すむような、コンパイルの方式(従って、純局所的でない変数の値の保存には、Lisp、λ-bindingを用いる。)も考えられる。Fortranのようにサブルーチンの静的なネストが許されない言語の場合には、上記方式で支障はない。Fortranは、その実行時記憶構造として、スタックを用いるため、①再帰性呼出しを許す、②局所性が保障されている変数のセルはスタック上にとられる、等の副次的な仕様の拡張が得られる。

尚、配列はスタック上にはとられないで、別にブロックとして、確保する。

3-2. スタック向き命令語体系

現在、FLATSマシンのシュレーターによるLisp、Fortran、の実行は、純スタックマシン(Sマシン、スタック上にオペランドをPushする命令とアドレスなしの、n-オペランド命令Opnを持ち、n個PopしたスタックにPopに結果をのせる。)に拠っている。Sマシンはコンパイラの、コード生成が容易であるという利点があるが、IBM360系のGeneral Registerマシンにくらべると、一般に実行速度が遅い。実行速度の低下する主なる原因は、Opnの実行によって、スタック上に積まれた値が、すべて失なわれてしまうことにある。FLATSでは、Sマシンに代って、変形Sマシン(S*マシン)を考える。S*マシンは、Sマシン命令系に加えて、スタックをOpn実行後もPopしない、n-1だけPopする等の、変形命令を持つものである。Sマシンは、コンパイラ作成の上でも、計算量を論ずる上でも重要であるが、高速性を要求される所では、S*マシン系の命令を用いた最適化が可能であるように、FLATSハードウェアを、設計

することが肝要であると考える。

3-3. スタックハードウェア

スタックは一般には、参照の局所性が、非常に高い記憶構造である。しかも通常のプログラムでは、配列等をスタック上にとらなければ、深さが1Kを越えることは少ない。FLATSでは、スタックを1KのbipolarRAMで実現する。1Kを越える場合には、ソフトウェア(ファームウェア)で、オーバーフローの処理を行う。スタックの先頭部数個を常に高速レジスターに入れるために、ハードウェアによる複雑な記憶管理をする他の、方式[14,15]に比べて、単純かつ高速である。一語を32ビットとすれば、V-スタック, Cスタックは各々、1KRAM(40nsアクセス時間)、32ことで実現できる。スタックに付随するポインタの管理には、U/Dカウンター加算器以外の特別なハードウェアは必要としない。

4. タグ

4-1. ソフトウェアに対する影響

記号、数式処理システムでは、実行時におけるデータタイプのチェックは必須であると考えられる。衆知のように、LispではFLATSのFunarallの機能を拡張するにせよ、実行時データタイプチェックを避けては、その使用用途が限定されてしまうであろう。FLATSでは、データ語にタグを設け、ハードウェアによるデータタイプチェックを行う。

タグの効用については文献[16,17]に論じられているが、タグに要するビット損失が大きいことや、ハードウェア設計時にソフトウェアに対する、周到な考慮が、必要である、等の理由から、現行の代表的大型計算機には、付設されていない。しかし、タグをencodeしてデータ語の一部に、付すか、あるいは、一ビットのタグモード指定のビットを各データ語に付加すれば、ビットの損失は最小限に抑えることができる。記憶用素子が低廉化している現状では、記憶の節約よりも、タグの導入に伴うソフトウェアの、統一化、簡素化等の便益のほうが、はるかに大きいものと思われる。タグの効用は、データタイプチェックの高速にとどまらず、数値表現の合理化、Call by name, by reference, by value などのパラメータ機構の合理化にも有効である。

一例として、一語32ビットの計算機へのタグの導入を考える。一倍長浮動小数点数, nを、$n = m \cdot 16^e$, $-64 \leq e \leq 63$ として表現する。$e = -64, -63, -62, +62, +63$をタグ付き語を示す予約(reserved)値とする。$-61 \leq e \leq 61$は一倍長浮動小数点数の指数とする。この場合のビット損失は $\log_2(123/128) = -0.055$にすぎない。

タグの種類については、次のようなものが、考えられる。

e.	名称	意味
63	BADNUM	演算過程中の零除算等々の不合理な結果.
62	BIGFLOAT	仮数部(24bit)は大指数型、又は多精度型浮動小数点数データへのポインタ.
-62	SINT	仮数部は短整数($|I| < 2^{24}$)を表わす。
-63	BIGINT	仮数部は、多倍長整数($|I| \geq 2^{24}$)データへのポインタ.

e	名称	意味
-64	TRAPNUM	仮数部はソフトウェア的に定義されたデータ型へのポインタで、ソフトウェア割出し("TRAP")によって処理する。

異なったタイプへのCoercionはタグの付けかえを含め、実行時に行う。これによって整数と浮動小数点数の指数は常に自動的に任意倍長整数として、取扱われ、4倍長以上の超高精度計算も容易にできる。

4-2. タグ解釈ハードウェア

タグの種類をnとし、二項演算を例にとれば、n^2の組合せの演算がある。これだけの組み合せをμプログラムで実行したのでは、Control Storageの不足は明らかである。

FLATSでは使用頻度に応じて、hard logicによる実行、μプログラムによる実行、トラップを利用したマクロプログラムによる実行、の3種類に分けて、タグ付き演算を行う。高速を要する基本的データタイプどおしの演算は、hard logicで実行する。使用が確立したタイプどおしの演算は、順次μプログラム化を計る。マクロ命令レベルでのタグ解釈過程に、前述のハッシングハードウェアを用いて、解釈ルーチンの先頭アドレスをO(1)で割り出すことも可能である。

5. その他

5-1 インディックスレジスタ

ここでいう、インディックスレジスタとは、アドレス計算に使用されるレジスタの意味とする。FLATSマシンではポインタ(即ち、アドレス)のタグの付いたデータ型に対する、演算を設けることにより、任意個のインディックスレジスタがあるのと同様な効果を得ることをまず考える。これは一般的であるが遅い。計算機の最も基本的演算の一つであるアドレス計算の高速化に最も有効なのは、多倍長インディックスレジスタであろう。通常のインディックスレジスタに1個のアドレスしか入れられないが、これを、Channel, Command語のようにいくつものアドレスとタグが入るようにしようというのである。複数、並列作動Bank構成のメモリーの場合には、インディックス語長までは、殆んど余分のハードウェアを付加することなく高速度で働かすことができる。n語長の連想レジスタをLRUで管理すれば、n語長のインディックス・レジスタは高い確率で最高速のメモリーに入っていることになり、主記憶との転送もn語単位なので速い。(インディックスは論理上は主記憶上に取られるので個数は任意)、B(Base), D(Baseからの Displacement), L(Limit, $|D| \leq |L|$ で、DとLは同符号)の3組で使われることが多い。基本演算は $D \pm n \to D$, $0 \leq D \leq L$ か否かの検査, $B+D$ 番地のメモリーへのアクセス(R/W)などである。アクセス長(1, 2, 4, 8長 Byteアドレス、bit アドレスとバイト・アドレスの差 etc)の変換指数mとデータ型T、アクセス制限(Read Only R/W, Execute Only etc.)を含めてインディックスは少なくとも(B, D, L, m, T, A)の6組の情報をもたせるようにし、かつこれらの情報に直接演算する回路をつけなくてはアドレス計算の高速化は望めないであろう。FLATSマシンではこのような、多倍長インディックス・レジスタの使用を検討する。

5-2. 記憶構成

数式処理の実用的な諸問題を解くには、1Mバイト程度の主記憶が必要である。二次記憶には最低限10Mバイトは必要であろう。

記憶構成法としては

(1)、bulk memory （例えば、サイクルタイム 1.5μsコア）とbipolar buffer memoryで構成する。（後者はいわゆるcache memoryである。）

(2)、すべて高速n-MOS memoryを用いる。

の2つを検討中である。（memory素子製造技術の著しい進歩に見合うアルゴリズムの開発、記憶構成法の研究が必要である。）

前者の記憶構成を採用すると仮定して、以下は議論をすすめる。最小限1Mバイト構成を考えると、bulk coreは128bit×64Kで、128bit十を一度に読み出す。これはハッシュハードウェアの項で論じたバンク数m=2の場合に相当する。但し、ここでは、パリティに要するビット数は考えない。cacheにはアクセス時間40nsのbipolar memoryを128個おき、（アドレス空間を指定するには14ビットいるとすれば、ディレクトリーに14×4個記憶素子がいる）、4語×1Kの通常のset associative cache memoryを実現する。bulk coreとの転送は128ビット単位に行う。

図3は、ハッシュ鍵をもって、対応する記憶ブロックをアクセスする、より汎用なcache方式を示したものである。ハッシュ演算の時は、ハッシュ鍵部との比較をみて、対応するセルのcache上の有無を調べる。LRUに従い、cacheの内容が、Coreに書きこまれるに当っては、cacheに登録された時点で計算された番地のセルに値が書きこまれる。通常の"番地"による記憶の参照では鍵との比較は行なわれない。このような方式にすれば、頻繁に参照される鍵、及びハッシュ検索列は、cache上に存在することになり、ハッシュアドレス計算をバイパスして、目的とするセルを高速にアクセスすることができよう。

次に、二次記憶（ディスク）と主記憶との間の記憶空間の仮想化の問題がある。記憶空間は、物理記憶空間と、論理記憶空間とに区別される。前者が二次記憶と主記憶とから、構成される空間である。コンパイラは論理空間上の番地をもった命令語コードを生成する。論理記憶空間は、リンケージエディタを介して、物理記憶空間へと写像される。ダイナミックリンクはハッシュハードウェアによって高速に実行される。多重プログラムを考慮しなければ、リンク後の物理アドレスは直接命令語に書き換えれば良いであろう。

物理空間の仮想化は直接マッピング方式を採用する。4Kバイトを1ページとした時、物理空間1ページごとに対応する記憶セルの番地を格納した表を中央処理装置に設け、ページ番号を直接索表インディックスとして、対応番地をとり出す。16Mバイトを物理記憶空間の最大値とすれば、4K容量のエントリーをもつ表が必要となる。この方式は、連想レジスタを用いる従来の方式より簡単かつ高速であると思われる。

物理空間の仮想化は、あくまでも、大容量空間の組織化にすぎず、リスト処理を効用する数式処理の効率を上げるには、十分な主記憶が必要なことは言うまでもない。

6. まとめ

FLATSは研究用システムである。従って、開発中の多少の設計変更、稼動後のレベルアップに対して、容易に応じられるような柔軟性を持つ必要がある。当初は本稿で述べたすべてのfeatureを実現せず、最小限動くシステムを作り上げて、機能を拡大するといった開発手順が要求されよう。今後、実装設計を進めていく上で、保守の問題（例えば、ハードウェアの障害を容易に発見できるような、回路実装の検討）や、信頼性の維持の問題を検討していく必要がある。

使用する論理素子をTTLで、しかも、数式処理システムが現在大型計算機で稼動する純ソフトウェア版よりも、高速に稼動するためには

(1). 記憶読み出し幅を大きくとり、（例えば、128ビット）、データ転送レートをあげる。

(2). 高速記憶素子を多量に使用し、論理回路を単純化し、諸演算の高速化を計る。

等の、現在の計算機技術の進展に見合った、計算機システムの設計手法が必要であると筆者らは考える。

計算機システムの汎用性と速度の追求は、重要な課題である。この2つが相反する課題であるとして、システムによるidiosyncrasyをいつまでも放置することは情報科学発展の阻害要因となろう。重要なことは、稀にしか使用されないfeatureは、当初低速にせよ使用可能にすることにより、汎用性をあくまでも追求することではなかろうか。FLATSは、科学技術計算の道具として、そのような汎用性を追求するシステムでもある。

文献

(1). 後藤英一 "言語FLATSとSP討論" 数理科学、76年7-8月号、サイエンス誌

(2). Goto, E. & Kanada, Y. "Hashing Lemmas on Time Complexities with Applications to Formula Manipulation" Proc. SYMSAC 1976

(3). Sassa, M. & Goto, E. "A Hashing Method for Fast Set Operations" Information Processing Letters. Vol 5, 31-34

(4). 後藤英一、井田哲雄 "データタイプに関する一考察" 第18回プログラミングシンポジウム. 1977

(5) 後藤英一、井田哲雄 "ハッシングマシン" 情報処理 Vol 18, No. 4.

(6) Ida, T. & Goto, E. "Performance of a Parallel Hash Hardware with Key Delection". (IFIP 77. 発表予定)

(7) Goto, E., Ida. T. & Gunii, T. "Parallel Hashing Algonithms" Information Processing Letters. Vol. 6, No. 1

(8). Goto, E. "Monocopy and Associative Algonithms in an Extended LisP" 東大理学部情報科学テクニカルレポート. 74-03.

(9) Kanada, Y. "Implementation of HLISP and Algebraic Manipulation Language REDUCE-2" ibid 75-01

(10) Terashima, M. "Algorithms Used in an Implementation of HLISP" ibid 75-03.

(11) LISP/.5 マニュアル MIT Press.

(12) Motoyoshi, F. "A Fortable LISP Compiler on a Hypothetical LISP Machine" 東大理学部情報科学テクニカルレポート、76-05.

(13) Randell, B. & Russel, L.J. "Algol 60 Implementation" Academic Press 1964
(14) Allmark, R.H. & Lucking, J.R. "Design of an arithmetic unit incorporating a nesting store." Proc. IFIP Congr. 62, PP 694-698.
(15) Lonergan, W. & king, P. "Design of the B 5000 system", Datamation Vol. 7, No. 5, PP 28-32 1961.
(16) Illife, J.K. "Basic Machine Prinoiples" Macdonald computer monographs 1968.
(17) Feustel, E.A. "On the advantages of tagged architecture" IEEE Transactions on Computers. Vol. C-22, No. 7, 1973
(18) Bobrow, D.G. "A note on Hash linking" CACM Vol.18, No. 7, 1975

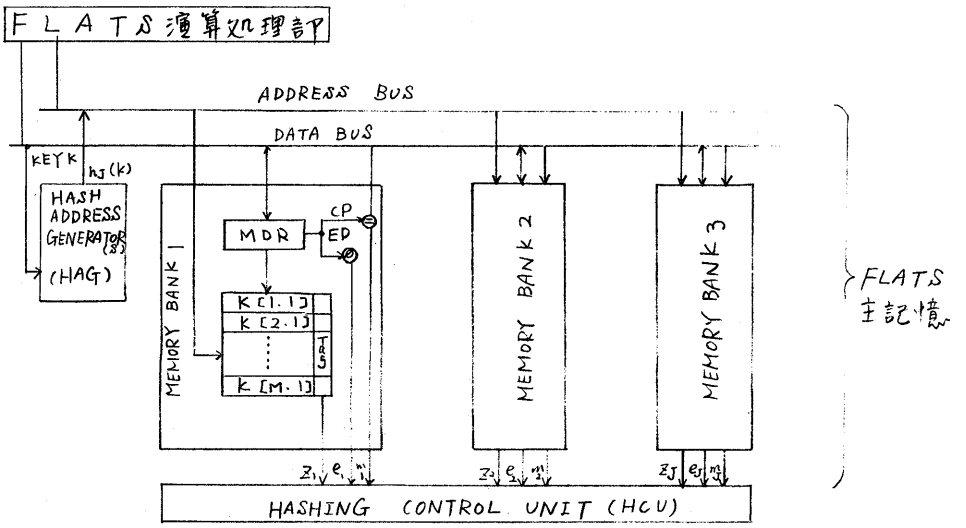

図 1 ハッシュハードウェア

表 1 PU: Average number of probes in an unsuccessful search

		α_m=0.6	0.7	0.8	0.9	0.95
J=1	PU	2.27	4.43	18.8	1.96E+3	3.07E+7
J=2	PU	1.98	3.66	14.2	1.22E+3	1.48E+7
J=4	PU	1.65	2.81	9.53	5.85E+2	4.46E+6
J=16	PU	1.13	1.51	3.36	68.0	8.04E+4
J=64	PU	1.00	1.03	1.37	7.43	6.35E+3

表 2 PS: Average number of probes in a successful search

		α_m=0.6	0.7	0.8	0.9	0.95
J=1	PS	2.50	3.33	5.00	10.0	20.0
J=2	PS	1.67	2.09	2.94	5.46	10.5
J=4	PS	1.27	1.48	1.91	3.19	5.72
J=16	PS	1.02	1.06	1.17	1.50	2.15
J=64	PS	1.00	1.00	1.01	1.09	1.26

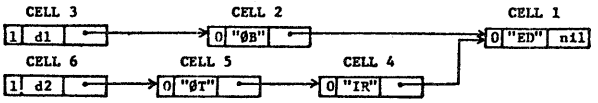

Fig. 2. Sharing of common sub-data.

図　2.

図.3 ハッシュ鍵を含んだ Cache directory