

FLATS, A MACHINE FOR NUMERICAL, SYMBOLIC AND ASSOCIATIVE COMPUTING

Eiichi Goto^{*,**}, Tetsuo Ida^{*}, Kei Hiraki^{**}, Masayuki Suzuki^{**} and Nobuyuki Inada^{*}

* Institute of Physical and Chemical Research, 2-1, Hirosawa, Wako-shi,
Saitama 351 Japan

** Department of Information Science, Faculty of Science,
University of Tokyo, 7-3-1, Hongo, Bunkyo-ku, Tokyo 113 Japan

Abstract

Functional aspects of a machine called FLATS, presently (June 1979) in the design stage, are described. FLATS aims to efficiently run both numerical and algebraic programs. Overflow free and variable precision arithmetic, table look-up computation, and associative computation based on single-hit content addressed tables are introduced for advanced numerical, algebraic and symbolic computing. Hashing hardware, tag mechanism and hardware list processing are used to realize these features.

1. Introduction

Numerical computation and symbolic formula manipulation are the major practices in scientific computation. While numerical computation has a long history, computer-aided algebra or large scale formula manipulation has become practical only recently.

In this paper, we describe the architecture of FLATS, a machine which aims to efficiently run scientific computations. We consider Fortran (F) and Lisp (L) as major scientific programming languages for reasons:

- (i) Scientific numerical programs are written mostly in Fortran.
- (ii) Lisp is the major host language used for symbolic and algebraic systems such as REDUCE¹ and MACSYMA². Moreover, Lisp has been used for writing a very large number of programs in the AI area.

Hence, high efficiency in running programs written in 'F' and 'L' is considered a minimum requirement, which is symbolized in the first two letters of 'FLATS'. 'A' in FLATS stands for associative capabilities; 'T' and 'S' stand for Table, Set and String which are the data types used in FLATS.

Many efforts have been made to increase the speed of computations by introducing parallelism^{3,4,5}, in which we include the pipe line architecture as a specific implementation (of parallelism). Vectorized operations on floating and fixed point numbers of fixed bit length are recognized as a class computations suited for highly parallel machines. It should be noted, however, that

highly parallel algorithms are not known for or not applicable to certain computations. For example, take the Euclidian gcd algorithm. The gcd of two integers r_0 and r_1 , say, is obtained by successively computing the remainder sequence $r_0, r_1, r_2, r_3 \dots$ (r_i is the remainder of r_{i-2} divided by r_{i-1}) until it reaches the first zero remainder $r_n = 0$ with r_{n-1} giving the gcd.

Parallel schemes faster than the Euclidian algorithm are not known. Hence, for gcd, a machine equipped with a single sophisticated high speed remainder unit would do better than parallel machines with hundreds of slower units. List processing, arbitrary precision integer arithmetic, and variable precision floating point arithmetic are other examples of operations which are not suited for contemporary highly parallel machines. The objective of FLATS is to efficiently run those operations which are not suited for highly parallel machines.

FLATS is basically a single instruction stream machine and parallelism shall be pursued only to a limited extent such as advance control, parallel hashing and parallel interpolation but it shall not incorporate parallel processor arrays.

FLATS may hence be regarded complementary to the highly parallel machine approach.

The numerical support to algebraic manipulation systems is indispensable since the process of algebraic manipulation requires exact calculation of intermediate (and final) numerical coefficients as opposed to approximations done by conventional fixed precision floating arithmetic.

Furthermore, the results of the algebraic computation must often be given to numerical computation systems.

Functional requirement for FLATS is therefore to incorporate the different characteristics of the two languages (F and L), e.g. compile-time vs. run-time data type checks, and static vs. dynamic storage allocation, into a single architecture, without impairing the efficiency of either one.

Besides the advanced numerical features, associative capabilities have been found to be very powerful⁶ in efficiently carrying out basic operations in formula manipulation.

We first give in section 2 functional specifications of FLATS from users' point of view, and in section 3 consider the implementation from architectural point of view.

2. FLATS from users' point of view

2.1 Overflow free and variable precision computing

We observe that most contemporary computer systems neither provide adequate means for handling (1) overflows, (2) precision higher than built-in standard precisions, e.g. double or quadruple, nor (3) variable precision arithmetic. The design and implementations of some important class of algorithms^{7,8,9} are hampered by the lack of the systematic support of these three features.

To remedy such defects, FLATS is provided with following means:

- Integers are treated with automatic multiple-precision.
- Exponents and mantissas of floating numbers are expressed in integers as stated above, hence practically no overflow would occur.
- A variable precision scheme¹⁰ is adopted, in which precision is specified by a status word called an ASW (Arithmetic Status Word), which is part of the PSW (Program Status Word).

These features are made available to users in an extended FORTRAN called BFORT (Bignum Fortran).

2.2 Tabulative computing

Before high speed computers came into extensive use, look-up of mathematical tables was an essential process of numerical calculation. This fact is in quite contrast to 'on-demand' computing prevalent today. To pursue further the speed-up of the present numerical computation, table look-up is becoming again a practical method, owing to the recent advances in memory fabrication technology. In FLATS, we provide following means for tabulative computing¹¹;

- (i) Tabulative computation of numerical functions with a floating number argument such as x^{-1} , $\sqrt{x}$, e^x , $\sin x$, $\log x$ and $\operatorname{atan} x$.
- (ii) Associative tabulation for (sparse) integer and/or symbolic arguments.

Linear (or quadratic) interpolation shall be used for 24 bit (single length) mantissa and cubic interpolation for 48 bit (double length) mantissa. Namely $f(x)$ is to be computed by $f(x) = f(x_0) + f'(x_0)\Delta x + (1/2)f''(x_0)\Delta x^2 + (1/6)f'''(x_0)\Delta x^3$ where $x = x_0 + \Delta x$ and x_0 is the most significant 12 bit part of the 48 bit mantissa x and each term is assumed to be less than the

previous term by a factor of 2^{-12} . The coefficients f , f' , f'' and f''' are tabulated (in ROM or RAM). This cubic interpolation can be made within time of two stages of multiplications by introducing parallelism:

Stage 1. Read the coefficient table and the table of cubes of the 12 bit (in precision) Δx^3 and compute the 24 bit Δx^2 in parallel. (Table look up is assumed to be no slower than multiplication.)

Stage 2. Multiply and add in parallel.

For computing the 48 bit inverse $y = x^{-1}$, this scheme is faster than Newton iteration, which would consist of five stages of successive operations:

Stage 1. Read the 12 bit approximate inverse $y_0 = x^{-1}$ from table.

Stage 2,3. Two multiplications to get the 24 bit inverse $y_1 = (2 - xy_0)y_0$.

Stage 4,5. Two multiplications to get the full 48 bit inverse $y = (2 - xy_1)y_1$.

When a function, say g with (sparse) integer or symbolic arguments, is computed in tabulation mode, the value $v = g(i_1, \dots, i_n)$ is tabulated in a content addressed table (CAT), using the tuple $(g, i_1, \dots, i_n)$ as a search key in the CAT. In the subsequent computations of g with the same values of the arguments, the value v is fetched immediately from the CAT. Associative tabulation effectively and automatically speeds up recursions by avoiding repetitive computation. For example, the notoriously slow (exponential time) recursive computation of the Fibonacci numbers:

```
procedure Fib[n] = [n<1 -> 1;
                  T -> Fib[n-1] + Fib[n-2] ]
is turned into a fast algorithm (linear time for the first evaluation and O(1) time for the subsequent evaluations).
```

2.3 Associative capabilities and built-in data types

Starting from the pioneering associative language

LEAP¹², a variety of associative language and architectures for the associative processing have been proposed and implemented.

We consider an AMT (Address mapping table) and an h -op hash table (Fig. 1) as the two basic entities for building associative data structures¹³. We define an AMT as a mapping from bit patterns of fixed sizes (actually 32 bits/64 bits) into addresses (consecutive integers). An AMT may be regarded as a single-hit associative memory. A multi-hit associative scheme such as found in LEAP is to be dealt with by software, making use of arrays, linked-lists and AMT's. Table 1 gives the built-in data types in FLATS. The data types STR, INT, FLOAT, VECT (one dimensional array) are the same as in most other languages except that INT and FLOAT are of arbitrarily high precision as stated in 2.1. PAIR is the same as in Standard Lisp¹⁴.

The data types prefixed by H denote those types obtained by hashing (h -op). h -type objects are guaranteed to be unique by virtue of hashing. The Lisp concept of 'intern' corresponds to that of h -op with the following differences; that h -op does not allow remob¹⁵ and that h -op can apply to all data types in FLATS, whereas intern op in Lisp 1.5 is applicable only to atomic symbols. h -type objects must be read-only so as for the hashing search mechanism to work properly. Hence, HSTR objects are the same as atoms of Lisp; i.e. every different string of characters is represented as a unique pointer to a table called oblist in Lisp 1.5, or to name tables in assemblers and compilers. HINT and HPAIR objects are multi-precision integers and hashed lists, respectively. Equality check of two long integers of type HINT or two lists of type HPAIR is reduced to that of two pointers. HPAIR and HINT can be made by the shared linked list method as shown in Fig. 1.

Insertion, deletion and membership test of an AMT entry correspond to the same operations on a set. A 'set' created by h -op (represented by an AMT) is given^{6,16} and shown in Fig. 1. In FLATS the hashed AMT (HAMT) is a synonym to 'set'.

A CAT consists of an AMT and RAM (random access memory, i.e. hardware term for VECT sharing the same addresses). A CAT can be hashed similarly to an AMT and the resultant data type is called HCAT.

These data types, especially the hashed types, have been found to be very useful in polynomial manipulation. For example, take a polynomial $P = 3AX^2 + 4BY^3$, where AX^2 and BY^3 are term identifiers; and 3 and 4 are coefficients. There are many (non-unique) ways to represent a term identifier in list forms owing to the com-

mutative nature of multiplication. AX^2 , for example, is represented in a list form ((A 1) (X 2)) and ((X 2) (A 1)) corresponding to AX^2 and X^2A respectively. The HCAT representation {A:1, X:2} is unique (AX^2 and X^2A are represented by a pointer to a unique HCAT structure). In this case, P is represented uniquely by nested HCAT's as

$$P = \{\{A:1, X:2\}:3, \{B:1, Y:3\}:4\}.$$

A FLATS FORTRAN program for multiplying polynomials in the HCAT normal form can be written in 9 lines:

```

FUNCTION CATMULT (P,Q)
  C = MKCAT(0)
  SWEEP 1, P, TP=KEY, FP=VAL
  SWEEP 1, Q, TQ=KEY, FQ=VAL
  T = H(CATADD(TP, TQ))
  C(T) = C(T) + FP*FQ
  CATMULT = H(C)
  RETURN
END

```

Here SWEEP is similar to DO in FORTRAN; SWEEP scans all the occurrences of keys in the specified CAT's P and Q. Function H corresponds to h -op above mentioned. This program would give the best known time complexity for multiplying multi-variate polynomials⁶.

3. FLATS from architects' point of view

3.1 Tagged architecture

A practical solution for efficient detection of the built-in data types given in 2.3 is a tagged architecture^{10,17}. Table 2 gives tags to be used in FLATS.

Take numerical computation for example. Arithmetic operations on small numbers tagged as "sint" (short integer) or "sfloat" (short floating number) are to be made with standard hardware at high speed, while those on tagged "big" numbers are to be trapped and handled by micro-programming. Since "sint" or "sfloat" numbers are likely to appear with high probability in most programs, the slow down factor due to the "big" numbers is very small.

Bit-loss, a common objection against tagbit(s), can be remedied by using a specific exponent value as a 'trapping tag'. In the case of 7-bit exponent $-64 \leq p \leq 63$ FLOAT representation, $p = -64, +63, +62$ are used as the tags to denote data types other than "sfloat" with the mantissa part having different meanings given in table 2. The effective bit loss is only $\log_2(123/128) = -0.06$ bits in this scheme.

3.2 Hashing Hardware

AMT's can be implemented either by using CAM

(content addressable memory) chips or by hashing. In a previous paper¹⁸, we presented a hardware hashing scheme which makes use of parallel read-out mechanism of memory. A parallel hashing hardware consists of multiple RAM banks, hash address sequence generators and a hashing control unit. A single comparator is attached to every RAM bank (cf. Fig. 2) in the parallel hashing hardware, whereas in CAM chip such as Intel 3104 a comparator is attached to every memory cell. Even at a chip level, a CAM requires several times more gates than a RAM chip. Hence total number of gates required for realization of CAM is greater by at least one order of magnitude than that for the hash table, with no significant speed gain over parallel hashing: The items in the hash table are searched in average $O(1)$ memory cycle (a quantity independent of the number of items entered in the hash table), in comparison with strict $O(1)$ used to search items entered in CAM. Since we are interested in the overall efficiency in our intended applications, it is more advantageous to invest the resource on enlargement of RAM memory than on still expensive CAM chips. Moreover, h -op of complex types such as AMT, CAT and VECT cannot be performed with CAM's.

Figure 2 shows the schematic diagram of the parallel bank hashing hardware of FLATS. Hashing is essentially the cycle (hash probe cycle) of (i) hash address generations, (ii) memory accesses, (iii) key comparisons and (iv) judgement of termination based on the comparisons. The hash tables, (i.e. AMT's and CAT's and an h -op table), are taken in main memory consisting of J banks. In a simple parallel hashing scheme with J banks, a single hash address generator and J comparators are used together with some auxiliary hash sequence control circuits. To handle key deletion, detectors of reserved words for denoting empty and deleted states are provided in each bank.

Assuming that the operations (i) - (iv) are performed in a single memory cycle, the performance of parallel hashing is given in number of memory cycles used until the completion of the probe cycles. Figure 3 shows that the average number of memory cycles in a successful search (finding a key in the table) in the worst case¹⁹. We presume the worst average performance in the performance evaluation when key deletion and insertion are repeated alternately. It shows that the basic hash operations can be performed in a time comparable to an indirect addressing operations, e.g. for $J = 16$ and the load factor of a hash table $\alpha = 0.9$. Note that the hashing hardware handles a fixed length key at a time. For handling variable length strings, arbitrarily long integers and lists, a shared linked list

above mentioned is used with the parallel hashing hardware.

3.3 Hardware support for list processing

The hardware mechanisms essential to the speed-up of Lisp are following;

- (1) run time data type checking,
- (2) hardware stack,
- (3) high memory bandwidth
- (4) segmented memory space with automatic boundary checking for arrays etc., and
- (5) instructions for Lisp primitives, such as car, cdr, cons and atom.

Note that the tagged architecture given in 3.1 is a solution to point (1).

As for point (2), we incorporate in FLATS both general (global) register and stack frame machine architecture. The basic op codes consist of four 8 bit fields as (op, r_1 , r_2 , r_3), where r_1 , r_2 , r_3 are register addresses each specifying either one of 128 global registers or one of 128 registers on the current stack frame. While the global registers are used to hold global entities, the stack registers are used for local entries and for (recursive) subroutine linkage.

In contrast to the two register address operations, $r_1 := \text{op}(r_1, r_2)$ notably used in IBM 360/370, three register address operations $r_3 := \text{op}(r_1, r_2)$ are used in FLATS. The three register scheme would greatly reduce the number of register to register transfer operations. For example, while no register to register transfers are needed in the three register scheme for the following codes

```
a:=add(x,y);
b:=sub(x,y);
c:=mul(x,y);
d:=div(x,y); ,
```

50% of the corresponding codes would be register to register transfers in the two register scheme as in

```
a:=x; a:=add(a,y);
b:=x; b:=sub(b,y);
c:=x; c:=mul(c,y);
d:=x; d:=div(d,y);.
```

The unit of memory banking used in hashing is actually taken to be the bit width of a single list cell (64 bits). Expansion of memory to increase the degree of parallelism (i.e. to increase J) will speed up hashing operations as shown in Fig. 3; this does not imply the requirement of higher memory bandwidth since the comparators are provided in the memory units.

Need for segmentation is clear since the main memory is conceptually divided into several segments; AMT's, CAT's, arrays, list areas, program space and so on.

Fundamental Lisp primitives given above are executed in a single memory cycle. In fact, when run-time data type check is made by hardware, car and cdr instructions are equivalent to simple load operations. Complex Lisp primitives such as member and reverse are microprogrammed.

Furthermore, by virtue of the hashing hardware, operations on property lists and the oblist which rely on linear search in most Lisp systems are performed more efficiently in FLATS.

3.4 Hardware garbage collector

Dynamic storage allocation of variable length segments of memory in FLATS requires a compactifying garbage collection scheme. The most time-consuming process in compactifying garbage collection schemes^{20,21} is pointer adjustment. FLATS is to incorporate the pointer adjustment hardware given in Fig. 5.

The pointer q in Q register is to be adjusted to be $q' = q - \Delta q$, where Δq is the total number of inactive cells in the memory block between base q_0 and q . A bit table of 2^n width is set up in the memory local to the hardware. The bit table is marked by a garbage collector: '1' in the m -th bit position in column ω corresponds to the inactive cell at $q_0 + \omega 2^n + m$ in the main memory. Therefore Δq is obtained by counting 1's between the base q_0 and q . The column $(q - q_0)/2^n$ of the offset table contains the cumulative bit count Σ . The adjustment can be done in a main memory cycle when fast memory for the offset table and the bit table is used.

4. Concluding Remarks

At a users' level, FLATS is a machine both for numerical processing and for formula manipulation. Instead of devising yet another language, we provide the facilities in the framework of existing languages, Fortran and Lisp (REDUCE is to be used for the user interface language). New compilers for Fortran and Lisp are under development, which incorporate the facilities for associative processing and overflow free and variable precision arithmetic, at the same time to accept programs written in Lisp and Fortran, which have been accumulated in libraries of scientific application packages. The underlying techniques include hardware hashing, tag mechanism, and hardware list processing (including hardware garbage collection). Concurrent garbage collection is not considered, since FLATS is not intended to be used for real time applications. FLATS is to be used as a back-end processor to a conventional computer system; hence the average performance, i.e. throughput, is a more important factor than responsiveness. For the same reason, hashing is

best suited for the associative processing discussed.

The associative capabilities of FLATS would also be suited for some operations for data base management.

Implementation of unified memory management of secondary and main memories using virtual tapes²² is under consideration.

Substantial amount of software for FLATS has already been written and has been running under a FLATS simulator. Architectural design of FLATS is an outcome of the experiences with the development of the software system HLISP^{6,16}. REDUCE is now running under HLISP system and are used for various applications. A version of BFORT mentioned before which translates the extended Fortran into HLISP has already been running.

References

- [1] Hearn, A.C. REDUCE2 user's manual, Utah Computational Physics UCP-19 (March 1973)
- [2] The MATHLAB Group, MACSYMA reference manual, Laboratory for computer science, MIT (1974)
- [3] Watson, W.J. The TI ASC - A Highly Modular and Flexible Super Computer Architecture, AFIPS 1972 FJCC, AFIPS Press
- [4] Russell, R.M. The CRAY-1 Computer System, CACM, Vol. 21, No. 1 (1978)
- [5] Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L. and Stokes, R.A. The ILLIAC IV Computer, IEEE Trans. Computers, Vol. C-47, No. 8 (1968)
- [6] Goto, E. and Kanada, Y. Hashing Lemmas on Time Complexity with Application to Formula Manipulation, Proc. ACM-SYMSAC, (1976)
- [7] Takahasi, H. and Mori, M. "Double Exponential Formulas for Numerical Integration", Pub. Research Institute for Math. Science, Kyoto University, 9 (1974), 721-741
- [8] Grau, A.A. "Algorithm 256, Modified Graeffe Method", Collected Algorithms from ACM, 256-P1-0
- [9] Brent, R.P. "Fast Multiple-Precision Evaluation of Elementary Functions", JACM, Vol. 23, No. 2, (April 1976), 242-251
- [10] Ida, T., and Goto, E. Overflow Free and Variable Precision Computing in FLATS, Journal of Information Processing Vol. 1, No. 3 (1978)
- [11] Goto, E. and Terashima, M. MTAC - mathematical Tabulative Automatic Computing, Technical Report of Information Science, 77-03, Faculty of Science, University of Tokyo, (1977)
- [12] Feldman, J.A. and Rovner, P.D. An Algol-Based Associative Languages, CACM, Vol. 12, No. 8 (1969)
- [13] Goto, E., Sassa, M. and Kanada, Y. Algor-

- ithms and Programming with CAM's (Content Addressable Associative Memories) Technical Report of Information Science Department, 78-04, Faculty of Science, University of Tokyo, (1978) submitted for publication
- [14] Marti, J.B., Hearn, A.C., Griss, M.L., and Griss, C. Standard Lisp Report, UUCS-78-101, University of Utah, 1978
- [15] McCarthy, J., et. al., Lisp 1.5 Programmers Manual, MIT Press, Cambridge, Mass., 1962
- [16] Sassa, M. and Goto, E. A Hashing Method for Fast Set Operations, Information Processing Letters, Vol. 5, No. 2, (1976)
- [17] Feustel, E.A. On the advantage of tagged architecture, IEEE Trans. Computers, Vol. C-21, No. 9 (1972)
- [18] Ida, T. and Goto, E., Performance of a Parallel Hash Hardware with Key Deletion, Proc. IFIP Congress 77, North-Holland (1977)
- [19] Ida, T. and Goto, E. Analysis of Parallel Hashing Algorithms with Key Deletion, Journal of Information Processing, Vol. 1, No. 1 (1978)
- [20] Wegbreit, B. A Generalized Compactifying Garbage Collector. The Computer Journal, Vol. 15, No. 3, (1972)
- [21] Terashima, M. and Goto, E. Genetic Order and Compactifying Garbage Collectors, Information Processing Letters, Vol. 7, No. 1
- [22] Sassa, M. and Goto, E. "V-tape", a Virtual Memory Oriented Data Type, and its Resource Requirements, Information Processing Letters, Vol. 6, No. 2, (1977)
- [23] This paper is the updated version of the paper by the same authors with same title given at ACM IEEE Symposium on Computer Architecture, May 1979, Philadelphia, PA.

main types	examples	hashed types	examples
ATOM			
STRing	3STOM	HSTR	3HTOM, 'TOM', TOM (three optional representations)
INT	-0137 +137	HINT	-137 137
FLOAT	3.1415	HFLOAT	3.1415
PAIR		HPAIR	
DLIST	(A.B)	HDLIST	(A;B)
LIST	((A 1) (B 2))	HLIST	((A,1), (B,2))
FUNC	Fib		
VECT	[3 5 6 7]	HVEC	[3,5,6,7]
AMT	{A B}	HAMT	{A,B}
CAT	{A:1 B:2}	HCAT	{A:1,B:2}

Table 1. Built-in data types in FLATS

Fig. 2 Parallel Hashing Hardware

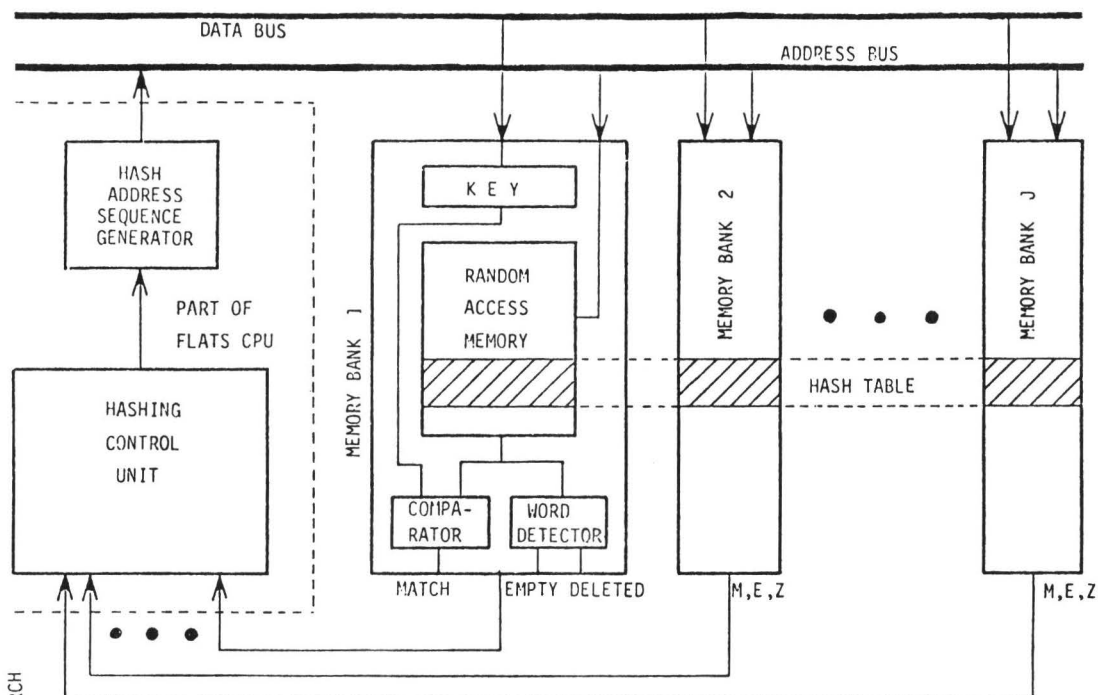
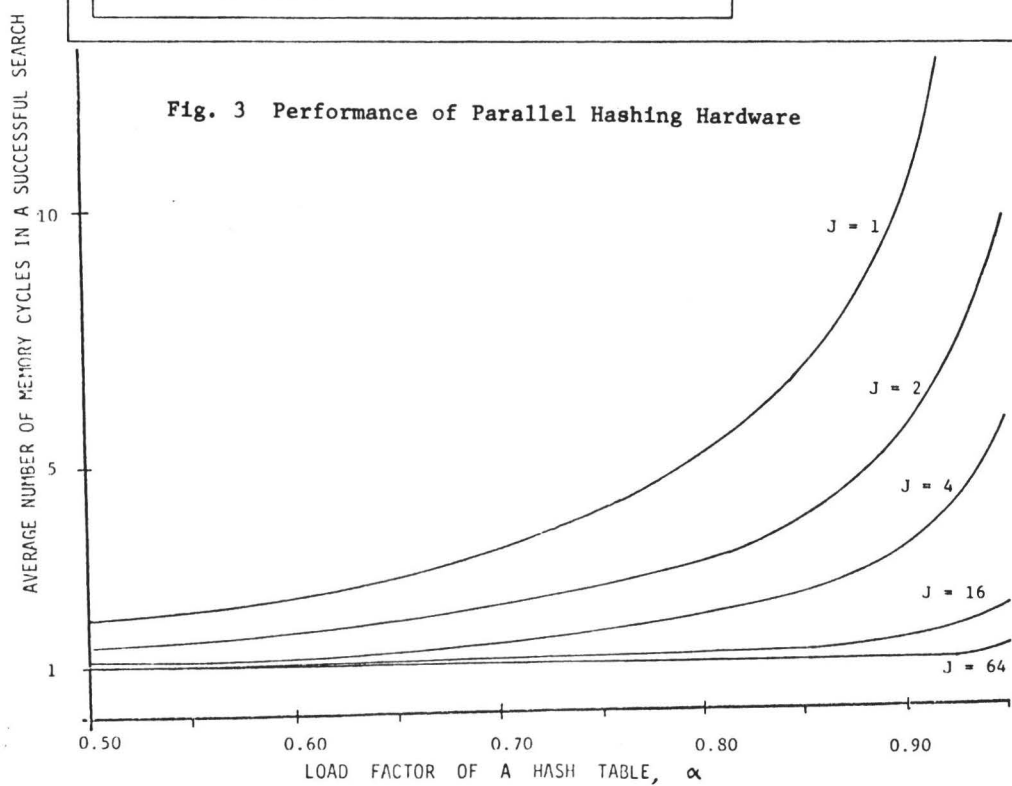


Fig. 3 Performance of Parallel Hashing Hardware



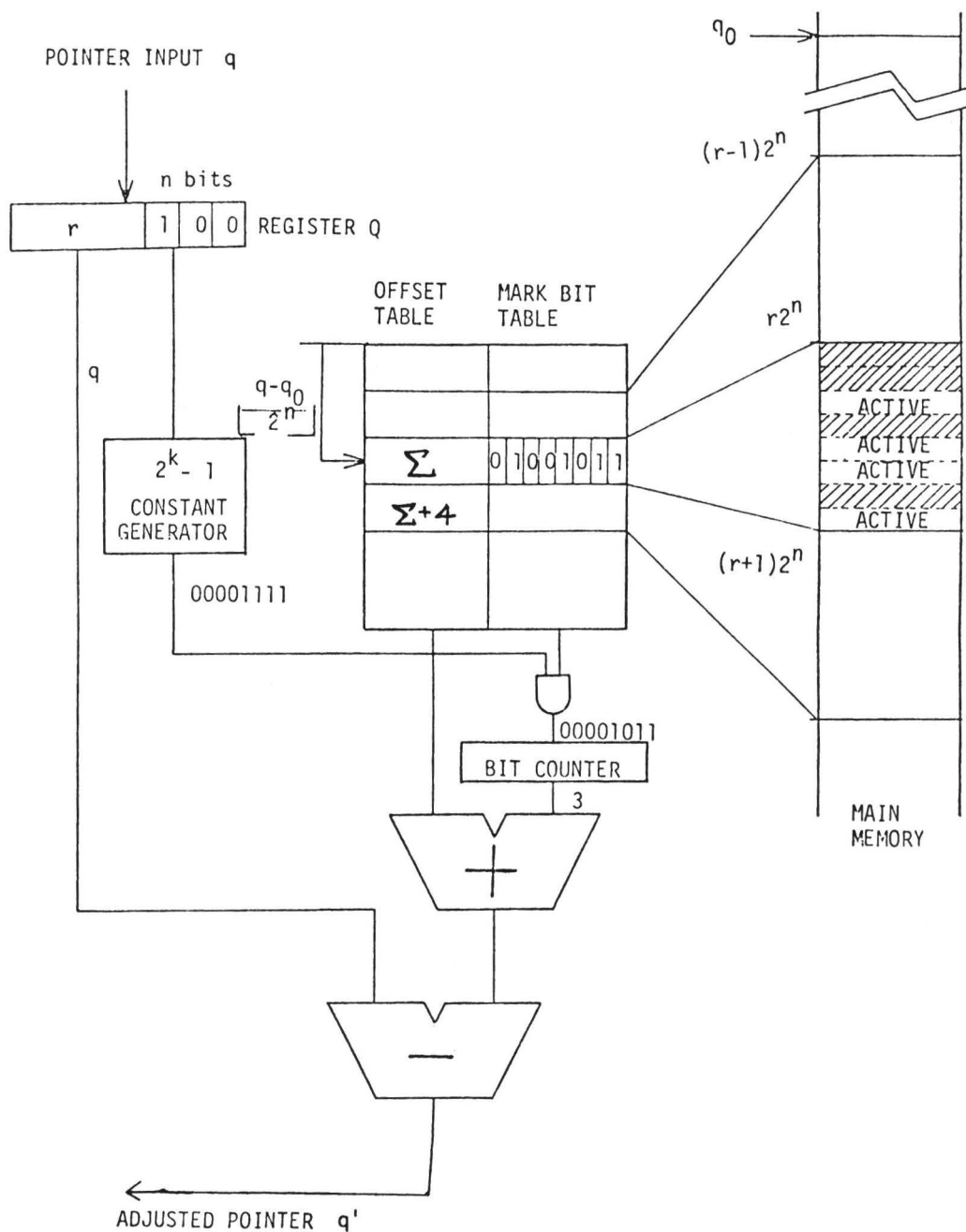


Fig. 4 Hardware for Pointer Adjustment