Dynamic Memories with Rapid Random and Sequential Access

ALFRED V. AHO AND JEFFREY D. ULLMAN

Abstract—We propose a new architecture for dynamic memories in which the contents of any cell in memory can be accessed by applying a sequence of two primitive memory operations. The advantage of our memory over previous designs is its fast sequential access. Any word in an n cell memory can be accessed in $0(\log n)$ steps. However, once two consecutive words have been accessed, following words can be accessed in one step per word.

Index Terms—Access time, memory architecture, random access, sequential access.

I. Introduction

A *dynamic memory* is an array of n cells, each of which can hold one data word. See [1]-[3] for example. One cell, called the *window*, is the only cell in the memory whose contents can be read or written externally. However, the contents of the cells in the memory can be internally rearranged by applying a sequence of operations called *memory transformations*. In this paper we consider dynamic memories having memory transformations that are permutations of the contents of the memory. For example, if the permutation $\pi = (i_0, i_1, \cdots, i_{n-1})$ is applied to an n cell dynamic memory, then in one step, the contents of cell j are transferred to cell i_j, for all j, $0 \leqslant j \leqslant n$.

A simple example of a dynamic memory is a circular shift register with one readout position. The cells may be thought of as the n bit positions of the register. The window is the readout position. The only memory transformation available is the cyclic permutation.

It should be obvious that if only one memory transformation is used, then access is slow, since as many as $n - 1$ steps (applications of the memory transformation) can be required to bring the contents of a particular cell to the window. However, if more than one transformation is permitted, then the access speed of a memory can be dramatically improved. For example, if two transformations are allowed, then it can be shown that there are transformations that will bring the contents of any memory cell to the window in $\lceil \log_2 n \rceil$ steps for any $n \geqslant 1$.

The case $n = 2^k$ was considered by Stone [3]. Two permutations were exhibited which meet this lower bound on access time, in that the contents of any cell can be brought to the window by applying at most $k = \log_2 n$ transformations to the memory. The two transformations proposed by Stone were *shuffle* π_s, where

$$\pi_s(i) = \begin{cases} 2i & i = 0, 1, \cdots, n/2 - 1 \\ 2i - n + 1 & i = n/2, n/2 + 1, \cdots, n - 1 \end{cases}$$

and *exchange shuffle* π_{es}. We define *exchange* to be π_e, where for $0 \leqslant i < n$

$$\pi_e(i) = \begin{cases} i + 1 & i \text{ even} \\ i - 1 & i \text{ odd} \end{cases}$$

The exchange-shuffle permutation π_{es} is π_e followed by π_s.

II. Dynamic Memories with Rapid Sequential Access

In many applications, it is common for several words of data to be accessed sequentially, e.g., when a program or block of information is accessed. Let *datum* j denote the initial contents of cell j. To access a block of data, say datum j, datum $j + 1$, datum $j + 2$, $\cdots$, in a dynamic memory we must first locate the cell that currently contains datum j and move the contents of that cell to the window. Then we must locate the cell that contains datum $j + 1$ and move its contents to the window, and so on. Using Stone's shuffle memory, once we have transferred datum j into the window, on the order of $\log_2 n$ transformations are typically needed to bring each of datum $j + 1$, datum $j + 2$, $\cdots$, into the window [3]. Thus to access a block of b consecutive words would require approximately $b \log_2 n$ transformations.

In this paper we propose a new architecture for a dynamic memory. Our memory has $n = r^k - 1$ cells where r and k are positive integers. We use a pair of transformations that can bring any datum in the memory to the window in $1.89 \lceil \log_2 n \rceil$ steps.[1] (This is achieved by the case $r = 3$ in the developments to follow.) However, if we are accessing a block of b words, then once we have brought the first two words to the window, each successive word in the block can always be brought to the window in one step. In fact, we shall see that a block of b words can be accessed in at most $2.52 \lceil \log_2 n \rceil + b$ steps.

We begin by considering the case $r = 2$ using a pair of transformations based on binary notation. Subsequently, we

Manuscript received December 1, 1972; revised April 24, 1973.

A.V. Aho is with Bell Laboratories, Murray Hill, N.J.

J.D. Ullman is with the Department of Electrical Engineering, Princeton University, Princeton, N.J.

[1] We observe that if an implementation of Stone's shuffle memory "wires in" π_s and π_e, rather than π_s and π_{es}, and π_{es} is simulated by π_e followed by π_s, then in the worst case $2\lceil \log_2 n \rceil$ steps would be required to fetch any datum.

shall generalize the ideas to arbitrary radix r. As indicated previously, there are certain advantages to radix 3, and radix 4 also has certain good properties.

The two transformations we propose for radix 2 are π_m and π_c, defined by[2]

$$\left.\begin{aligned}\pi_m(i) &= 2i \bmod n\\ \pi_c(i) &= (i-1) \bmod n\end{aligned}\right\} \quad \begin{cases} i = 0, 1, \cdots, n-1 \\ n = 2^k - 1 \text{ for integer } k. \end{cases}$$

π_m is a shuffle on an odd number of cells, and π_c is a cyclic shift. The restriction that n be one less than a power of two introduces useful algebraic properties familar from ones' complement arithmetic.

Example 1: Let us consider π_m for the case $k = 3$, $n = 2^3 - 1 = 7$. We see that π_m is a permutation with 3 cycles, two of length 3 and one of length 1.

i	rep i	π_m
0	000	
1	001	
2	010	
3	011	
4	100	
5	101	
6	110	

Note that if we represent i as a binary number, then the binary representation of $\pi_m(i)$ is simply the binary representation of i left rotated one bit.

III. Memory Maps

To access a given datum, we must keep track of its current address in the memory. We define a *memory address map* as a function f from data to cells such that $f(j)$ is the cell currently containing datum j. We shall maintain a record of the memory map in order to determine the current position of each datum.

The memory map changes whenever a memory transformation is applied. However, we shall show that the memory map can be represented by two integers and readily updated whenever either one of our memory transformations is applied. Then we shall show how the current address of a datum can be used to generate an appropriate sequence of transformations to bring the given datum to the window.

We first observe that if we use the transformations π_m and π_c, the memory map always has the form

$$f(j) = (2^p j + q) \bmod n$$

for integers p and q, where $0 \leqslant p < k$ and $0 \leqslant q < n$. (Recall that n and k are related by $n = 2^k - 1$.)

We may verify the preceding by observing that π_c moves datum j from cell $f(j)$ to cell $(f(j) - 1) \bmod n = [2^p j + (q - 1)] \bmod n$, and that π_m moves datum j from cell $f(j)$ to cell $2f(j) \bmod n = (2^{p+1} j + 2q) \bmod n$. Since $2^k \equiv 1 \bmod n$, we can characterize the effects of the two transformations on the memory map solely in terms of the parameters p and q as follows:

$$\pi_c \text{ maps } \begin{cases} p \text{ into } p \\ q \text{ into } (q-1) \bmod n \end{cases}$$

$$\pi_m \text{ maps } \begin{cases} p \text{ into } (p+1) \bmod k \\ q \text{ into } 2q \bmod n. \end{cases}$$

Example 2: Let $k = 3$, so $n = 7$. In Fig. 1 we see the memory map at each step when the sequence $\pi_m \pi_c \pi_m \pi_c$ is applied to the initial configuration.

Initially, the memory map is $f(j) = j$, characterized by $p = q = 0$. After π_m, the map is $f(j) = 2j \bmod 7$, characterized by $p = 1$, $q = 0$. After $\pi_m \pi_c$, we have $f(j) = (2j + 6) \bmod 7$, characterized by $p = 1$, $q = 6$. Note that $-1 \bmod 7 = 6$. The next two memory maps are $f(j) = (4j + 5) \bmod 7$, characterized by $p = 2$ and $q = 5$; and $f(j) = (4j + 4) \bmod 7$, characterized by $p = 2$ and $q = 4$. □

IV. An Efficient Accessing Algorithm

We shall assume the window is cell 0. An *accessing sequence* for datum j is a sequence of transformations that will move datum j into the window. There are many possible accessing sequences for datum j. For example, we could apply just π_c's until datum j has been moved to the window. However we shall now present an algorithm to generate an accessing sequence that never contains more than $2 \log_2(n + 1) - 2$ transformations, no matter what previous sequence of π_m's and π_c's has been applied.

In what follows, *locj* is a variable that represents the current location of datum j. We will represent *locj* as a k-bit number. The function *lsd*(*locj*) stands for the least significant (rightmost) digit of *locj*. All arithmetic is modulo $n = 2^k - 1$ except where otherwise indicated.

Algorithm 1: Accessing sequence for datum j.

Input: p, q (memory map) and j (the datum to be fetched).

Output: A sequence of π_m's and π_c's that will move datum j to cell 0. As a side effect, p and q are updated so they continue to represent the memory map.

Method:

$locj \leftarrow 2^p * j + q$

loop: **if** *lsd* (*locj*) $\neq 0$ **do**

$\{\pi_c; locj \leftarrow locj - 1; q \leftarrow q - 1\}$

if $locj \neq 0$ **do**

$\{\pi_m; locj \leftarrow 2 * locj; q \leftarrow 2 * q;$

$p \leftarrow (p + 1) \bmod k$; **goto** *loop*$\}$. □

We can view Algorithm 1 in two ways, depending on how

[2] $a \bmod b$ is the remainder when a is divided by b. See [4] e.g., for an discussion of modular arithmetic.

Datum	Initially in cell	After π_m	After $\pi_m\pi_c$	After $\pi_m\pi_c\pi_m$	After $\pi_m\pi_c\pi_m\pi_c$
0	0	0	6	5	4
1	1	2	1	2	1
2	2	4	3	6	5
3	3	6	5	3	2
4	4	1	0	0	6
5	5	3	2	4	3
6	6	5	4	1	0
p	0	1	1	2	2
q	0	0	6	5	4

Fig. 1. Memory maps.

we interpret the instructions π_c and π_m. If π_c and π_m are taken as instructions to emit π_c and π_m, respectively, then Algorithm 1 will generate a sequence of transformations that will cause datum j to be brought to the window. On the other hand, if π_c and π_m are viewed as instructions that apply the corresponding transformations to the memory, then Algorithm 1 will itself move datum j to the window. We will not distinguish between these two interpretations.

It is straightforward to check that Algorithm 1 causes p and q to continually represent the memory map and *locj* to continually represent the location of datum j. One simply observes that after a π_c or π_m is generated, these parameters are correctly updated.

We must also show that Algorithm 1 terminates. Since it only terminates when $locj = 0$, we will have then shown that the algorithm generates a correct accessing sequence.

In ones' complement notation multiplying *locj* by 2 rotates the bits of *locj* one position left (with end around carry). Each time through the loop, either we find the rightmost bit of *locj* is zero or we set this bit to zero. Thus, after at most k passes through the loop, we have $locj = 0$.

As a corollary, we note that the sequence of π_m's and π_c's generated by Algorithm 1 has at most $k-1$ π_m's and $k-1$ π_c's. (The possibility of k π_c's is ruled out. Since we represent zero by k 0's, it is not possible that *locj* initially is all 1's.) We conclude that Algorithm 1 brings datum j to the window in no more than $2\log_2(n+1) - 2$ steps no matter what previous sequence of π_m's and π_c's has been used.

V. Sequential Access

Suppose that we sequentially access a block of words: datum j, datum $j+1$, datum $j+2$, ···. The advantage of our proposed memory over the shuffle memory becomes evident when we access datum $j+2$, datum $j+3$, ···. Specifically, we shall show that if we have brought datum j and then datum $j+1$ to the window using the accessing sequences generated by Algorithm 1, then the accessing sequences for each of datum $j+2$, datum $j+3$, ··· are of length 1, namely, π_c. Moreover, for datum $j+1$ Algorithm 1 generates only π_m's followed by a single π_c, and thus accessing datum $j+1$ requires less than average time. These facts are shown in the following theorem.

Theorem 1: Suppose we have just used Algorithm 1 to bring datum j and datum $j+1$ to the window. Then Algorithm 1 will bring datum $j+2$ to the window with the single transformation π_c.

Proof: Let $f(i) = (2^p i + q) \bmod n$ be the memory map after bringing datum j to the window. Since $f(j) = 0$, we have $(2^p j + q) \equiv 0 \bmod n$. If we now apply Algorithm 1 with input $j+1$, we initialize *locj* to $(2^p(j+1)+q) \bmod n = 2^p \bmod n$.

Thus, the ones' complement representation for *locj* has a single 1, which is positioned so it has p 0's to its right. It is easy to see that for this value of *locj*, Algorithm 1 generates $(k-p) \bmod k$ π_m's followed by π_c. After applying Algorithm 1, we thus have $p = 0$. At this time, we must have $f(j+1) = (2^0(j+1)+q) \bmod n = 0$. Thus we see that $q = -(j+1)$.

Then, when we apply Algorithm 1 with input $j+2$, we have $p = 0$ and $q = -(j+1)$. Hence we now initialize *locj* to 1, and Algorithm 1 generates a single π_c. □

Example 3: Suppose that we wish to access the words datum 6, datum 0^3 and datum 1 in the memory of Example 2, when initially $p = q = 0$.

The accessing sequence generated by Algorithm 1 for datum 6 is $\pi_m\pi_c\pi_m\pi_c$. At this point $p = 2$ and $q = 4$.

The accessing sequence for datum 0 is then $\pi_m\pi_c$. At this point $p = 0$ and $q = 0$ and the contents of the memory are back in their original order with datum 1 in cell 1.

The accessing sequence for datum 1 is then π_c. □

VI. Generalization to Arbitrary Radix

We can generalize our memory structure using an arbitrary radix in the following manner. If r is any integer, $r \geq 2$, we may use as our two transformations, π_c and $\pi_m{}^r$ where π_c is as before and $\pi_m{}^r$ is defined as:

$$\pi_m{}^r(j) = rj \bmod n, \quad \begin{cases} j = 0, 1, \cdots, n-1 \\ n = r^k - 1 \text{ for integer } k \end{cases}$$

Note that $\pi_m{}^2$ is just π_m.

We must now generalize Algorithm 1 to produce a sequence of π_c's and $\pi_m{}^r$'s. To do so, we assume that *locj* is represented in base r, using $(r-1)$'s complement notation. Thus multiplication by r is simply a left rotation of one base r digit. Also, we can assume that the current memory map is always of the form $f(j) = (r^p j + q) \bmod n$.

[3] We assume datum 0 follows datum 6.

Algorithm 2: Accessing sequence for datum j

Input: p, q (memory map) and j (datum j).

Output: A sequence of π_c's and ${\pi_m}^r$'s that will move datum j to cell 0. The values of p and q are adjusted to represent the memory map.

Method:[4]

$locj \leftarrow r^p * j + q$

loop: **while** $(lsd\ (locj) \neq 0)$ **do**

$\{\pi_c; locj \leftarrow locj - 1; q \leftarrow q - 1\}$

if $locj \neq 0$ **do**

$\{{\pi_m}^r; locj \leftarrow r * locj; q \leftarrow r * q;$

$p \leftarrow (p + 1) \underline{\text{mod}}\ k;$ **goto** *loop*$\}$ □

The only noteworthy difference between Algorithms 1 and 2 is that in Algorithm 2 it may take as many as $r - 1$ repetitions of π_c to zero the least significant digit of *locj*. Thus, if $locj = a_1 a_2 \cdots a_k$ in base r notation, then Algorithm 2 will generate the accessing sequence

$${\pi_c}^{a_k} \pi_m {\pi_c}^{a_1} \pi_m {\pi_c}^{a_2} \cdots \pi_m {\pi_c}^{a_s}$$

for datum j. Here π^a is a sequence of a π's, and a_s is the rightmost nonzero digit in $a_1 \cdots a_{k-1}$. That is, $a_{s+1} a_{s+2} \cdots a_{k-1}$ is a string of all 0's.

The results of the previous section can be generalized to the base r situation in a straightforward manner. For example, we observe that during Algorithm 2, p and q will continue to represent the memory map, and that Algorithm 2 terminates with $locj = 0$ after generating at most $k\pi_m$'s and $k(r-1)-1$ π_c's.

In analogy with Theorem 1, we state the following result.

Theorem 2: If Algorithm 2 has just been applied to bring datum j and datum $j + 1$ to the window, then Algorithm 2 will bring datum $j + 2$ to the window by π_c.

VII. Optimization of the Radix

The average and maximum access times for a datum is a function of the size of the memory and the radix r. The average access time can be approximated by assuming that all digits from 0 to $r - 1$ are equally likely to appear anywhere in the current address of a datum. (Actually, the frequencies of the digits are almost the same, but there is one fewer $r - 1$ than the others.)

The expected number of π_c's generated by Algorithm 2 is $k(r-1)/2$ under our assumption. Also, the expected number of π_m's generated is closely approximated by $k - 1 - (1/(r-1))$. We thus see that a good approximation to the average length of an accessing sequence is $k - 1 + k(r-1)/2 - (1/(r-1))$.

Now $k = \log_r(n+1)$, so the average length is $\log_r(n+1) + [\log_r(n+1)]\ (r-1)/2 - (1/(r-1)) - 1$. For large n, this function is approximately $((r+1)/2) \log_r n$. We can decouple r and n by expressing the later formula as $((r+1)/2 \log_2 r) \log_2 n$.

We thus see that for all r, the average length of the generated accessing sequence varies as the logarithm of n. The interesting feature is the coefficient $(r+1)/2 \log_2 r$. Some values of this function are tabulated here

r	$\frac{r+1}{2 \log_2 r}$
2	1.500
3	1.262
4	1.250
5	1.292
6	1.354
7	1.425
8	1.500

Thus, a minimum is obtained by letting $r = 4$, and $r = 3$ gives almost comparable performance.

We might also be interested in the maximum length of the sequence. This occurs when all digits are $r - 1$, except one which is $r - 2$. This maximum length is $kr - 2$, or $r \log_r(n+1) - 2$. For large n, the function is $r \log_r n$, or $r/(\log_2 r) \log_2 n$. Some values of the coefficient are tabulated here.

r	$\frac{r}{\log_2 r}$
2	2.000
3	1.893
4	2.000
5	2.153
6	2.321
7	2.493
8	2.667

Here, a minimum is seen to occur when $r = 3$. When $r = 3$, to access a block of words would require at most $1.89 \log_2 n$ steps to access the first word plus at most $0.63 \log_2 n$ ($= \log_3 n$) steps to access the second word plus one step for each successive word. Thus b successive words can be fetched in at most $2.52 \log_2 n + b - 2$ steps.

VIII. Comparison with Shuffle Memory

Let us compare our two transformations ${\pi_m}^r$ and π_c with π_s and π_{es}, the two transformations used in Stone's shuffle memory. We notice a curious similarity in the case $r = 2$. Our transformations give a worst case access time of $2 \log_2(n+1)$ steps and fast sequential access. Stone's transformations have a worst case access of $\log_2 n$ but lack the fast sequential access. However, suppose we use transformations π_m and $\pi_c \pi_m$ instead of π_m and π_c. Then it is easy to show that each accessing sequence can be a sequence of at most $\log_2(n+1)$ of the transformations π_m and $\pi_c \pi_m$. This bound is virtually the same as for the shuffle memory; the only difference is that the shuffle memory has one more cell (2^k versus $2^k - 1$) for equivalent worst case access times. It is also interesting to note that the transformations π_m and $\pi_c \pi_m$ do not yield fast sequential access. That is, the accessing sequence π_c must be

[4] Note that this algorithm can be implemented using binary logic.

replaced by $\pi_c\pi_m$ followed by $\log_2(n+1)-1$ π_m's when π_m and $\pi_c\pi_m$ are used.

However, if an implementation of the shuffle memory used π_s and π_e to simulate π_{es}, then the worst case access time would be $2\log_2 n$, just as for our proposed memory. This implementation of the shuffle memory is thus quite similar to our proposal, but the shuffle memory would not possess the fast sequential access feature.

If we compare our memory with $r = 3$ and the shuffle memory using π_s and π_e as transformations, we find that ours is superior both in worst case (1.89 versus 2) and in expected access time (1.26 versus 1.5). All numbers are coefficients of $\log_2 n$. The case $r = 4$ gives a wider improvement in expected access time (1.25 versus 1.50). Of course, if the shuffle memory is implemented by π_s and π_{es}, the worst and expected time coefficients both become 1. We cannot compete successfully with this figure without losing the fast sequential access feature.

Acknowledgment

The authors would like to thank M.D. McIlroy and P.J. Plauger for their helpful comments.

References

[1] J. Bruno *et al.*, unpublished memorandum, Princeton Univ., Princeton, N.J.

[2] H.S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, pp. 153-161, Feb. 1971.

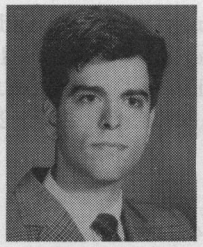
[3] —, "Dynamic memories with enhanced data access," *IEEE Trans. Comput.*, vol. C-21, pp. 359-366, Apr. 1972.

[4] G.H. Hardy and E.M. Wright, *An Introduction to the Theory of Numbers*. New York: Oxford, 1962.

Alfred V. Aho was born in Timmins, Ont., Canada, on August 9, 1941. He received the B.A. Sc. degree in engineering physics from the University of Toronto, Toronto, Ont., the M.A. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, N.J., in 1963, 1965, and 1967, respectively.

Since 1967, he has been with the Computing Science Research Center at Bell Laboratories, Murray Hill, N.J. In addition, he is currently an Affiliate Associate Professor and Chairman of the Computer Science Program at Stevens Institute of Technology. His main fields of interest are language theory, compiling theory, and the theory of algorithms. He is an author of numerous papers in the computer science field and a coauthor of the textbook, *The Theory of Parsing, Translation and Compiling*.

Dr. Aho is Vice Chairman of the Association for Computing Machinery Special Interest Group for Automata and Computability Theory.



Jeffrey D. Ullman was born in New York, N.Y., on November 22, 1942. He received the B.S. degree in applied mathematics from Columbia University, New York, N.Y., and the Ph.D. degree in electrical engineering from Princeton University Princeton, N.J., 1963 and 1966, respectively.

From 1966 to 1969 he was with the Computing Science Research Center of Bell Laboratories, Murray Hill, N.J. Since 1969 he has been with the Department of Electrical Engineering of Princeton University where he is now an Associate Professor. His main research interests are in language theory, compilers and theory of algorithms. He is the author of numerous papers and books in the computer science field.

Dr. Ullman is a member of the Sigma Xi, Tau Beta Bi, and Secretary-Treasurer of the Association of Computing Machinery Special Interest Group on Automata and Computability Theory.