Letters to the Editor

On Addressing Multidimensional Arrays

Dear Editor:

The article "Addressing Multidimensional Arrays" by H. Hellerman in the April issue of the *Communications* concerns a problem that has been of much interest at Rice University. Our computer was designed and constructed at the University. The instruction format provides indirect addressing to any level and six general purpose index registers, any number of which may be used additively at any level of indirect addressing. Since arrays of more than six dimensions are seldom of concern, we have available those features which Dr. Hellerman requested for his indirect scheme of addressing array elements. In our case, we started with flexible hardware and developed programming techniques to take advantage of it, whereas the author has derived machine specifications from programming theory.

I make reference to the paper "Array Manipulations and the Formula Language" [Rice Computer Programming Memorandum #1, Mar. 1959] by J. K. Iliffe (now with Ferranti, Ltd., London). Most of the ideas presented in this text have since been realized in programming systems for the Rice computer, and these are described in detail in subsequent internal publications. I shall mention some particulars of our scheme for indirect addressing of array elements.

All words associated with an array that are *not* elements of the array are called *codewords*. The structure and organization of codewords are essentially those described by Dr. Hellerman for his "tables." Thus, for a two-dimensional array the principal codeword addresses a vector of codewords, each of which addresses a vector of data, a row of the matrix.

Several features of this scheme have not been emphasized by Dr. Hellerman, and in view of our extremely rewarding experience I believe they are worth stressing. First, the additional storage required may well be offset by the simplicity with which an array element may be addressed; that is to say, it is certainly easier, and in most cases faster, to set index registers to the values i and j and make one indirect reference to memory than it is to compute any function of i and j before making a direct reference to memory. Second, for any array only the principal codeword address need be known for any reference; thus, the storage for the remainder of the array may be located arbitrarily with respect to those programs which refer to the array. Third, the storage of each vector associated with the array (whether it contains codewords or data) is entirely independent of every other so that the problem of storage allocation becomes one of handling several small vectors rather than one of considerable length. Fourth, addressing of nonrectangular arrays is exactly as easy as addressing of rectangular arrays, which is certainly not the case when address computation is carried out; and the complexity of addressing is not a function of the dimension of arrays. Fifth, the address portion of a codeword may contain the number $f - k$ (where f is the address of the first word of the vector and k is any integer) so that the first element is addressed with index k, a very convenient variable in many applications. Sixth, programs may be thought of as vectors and transfers to them made through their codewords without indexing; this concept generalizes with no difficulty to arrays of programs which are often very useful.

JANE G. JODEIT
Rice University
Houston, Texas

Hardware Conversion of Decimal and Binary Numbers

Dear Editor:

A recent technical note by W. C. Lynch [1] prompts me to describe the online hardware conversion system [2, 3] which has been in use on the DRTE solid state computer for the past year and a half [4]. The system is based on a general purpose shift register and counter [5] capable of shifting right or left one binary place per clock period and adding or subtracting 3 in one clock period [2]. The same basic method is used for both binary-to-decimal and decimal-to-binary conversion of integers, fractions and floating-point numbers. The method for integers will be described, the extensions for fractions and floating-point numbers being fairly obvious. The basic methods will be described first, followed by their implementation using shift registers.

Decimal-to-binary conversion of an integer is accomplished by successive division by two. The answer consists of the accumulated remainders from each step. Consider conversion of the decimal integer 19 to binary. First, write the decimal number and the target binary number (initially 0) side by side.

decimal	*binary*
19	00000

Now divide both numbers by 2 and place the remainder, 0 or 1, from the decimal division in the most significant bit position of the binary number. We now have

decimal	*binary*
$19 \div 2 = 9$	10000

Continuing we get:

decimal	*binary*
$9 \div 2 = 4$	11000
$4 \div 2 = 2$	01100
$2 \div 2 = 1$	00110
$1 \div 2 = 0$	10011

Thus, the binary equivalent of 19 is 10011. It is apparent that if the number of divisions by 2 is equal to the number of bits in the binary number, leading zeros will be automatically taken care of. There must of course be sufficient binary bits to hold the equivalent of the largest decimal number to be converted.

Conversion from binary to decimal is similar except that multiplication by 2 is used instead of division. To reconvert the above result we write:

decimal	*binary*
0	10011

Now, multiply both numbers by 2 and add any carry out of the binary register into the decimal result.

	decimal	*binary*
$2 \times 0 + 1$ carry	= 1	0011–
$2 \times 1 + 0$	= 2	011—
$2 \times 2 + 0$	= 4	11——
$2 \times 4 + 1$	= 9	1———
$2 \times 9 + 1$	= 19	————

Again, if the number of multiplications is equal to the number of binary digits, leading and trailing zeros are automatically accounted for. It will now be apparent that this method requires a simple method of multiplying and dividing BCD numbers by 2. In the binary system this is of course accomplished by shifting.

Binary coded decimal numbers can also be multiplied and divided by 2 by shifting, provided corrections are made after each shift in accordance with simple rules. Division will be considered first. The BCD number (8, 4, 2, 1) is placed in a set of 4-bit shift registers arranged with the most significant digit on the left. The coding is then changed to "excess 6 code" by adding 3 twice to each decimal digit independently. Division by 2 is now accomplished by linking the 4-bit registers and shifting the whole number one binary place to the right. Each decimal digit is now individually corrected in accordance with the following rule:

RULE 1. If the most significant bit in any decimal digit is 0, add 3 to that decimal digit. If the most significant bit is 1, no correction is needed—i.e. if a zero was carried *in* on the shift, add 3; if a one, make no correction.

The remainder from the division is the bit shifted out of the right-hand end of the least significant decimal digit. The complete conversion of the decimal number 19 is shown below. The digits examined for application of this rule are underlined.

	decimal		*binary*
	0001	1001	00000
add 3	0100	1100	
add 3	0111	1111	
shift	0011	1111	10000
correct	0110	1111	
shift	0011	0111	11000
correct	0110	1010	
shift	0011	0101	01100
correct	0110	1000	
shift	0011	0100	00110
correct	0110	0111	
shift	0011	0011	10011

Multiplication by 2 follows an analogous procedure except that the "excess 3" code is used for the BCD number; i.e., 3 is added only once. After left shifting to multiply by 2, each decimal digit is corrected by the following rule.

RULE 2. If the least significant bit of the next left decimal digit is "1", add 3. If the least significant bit of the next left digit is "0", subtract 3. I.e. if a "1" was carried *out* of any digit on the preceding shift, add 3 to that digit; if a "0" was carried *out*, subtract 3.

The complete binary to decimal conversion is shown below.

		decimal		*binary*
		0000	0000	10011
add 3		0011	0011	
shift	0	0110	0111	0011
correct		0011	0100	
shift	0	0110	1000	011
correct		0011	0101	
shift	0	0110	1010	11
correct		0011	0111	
shift	0	0110	1111	1
correct		0011	1100	
shift	0	0111	1001	
correct		0100	1100	
subtract	3	0001	1001	

Digits carried out are underlined. Note that the BCD register must be sufficiently large to contain the largest binary number to be converted and in this case the carry out is always "0".

The final subtraction of 3 is done on all decimal digits and is necessary to reduce the result from "excess 3" code to normal 8, 4, 2, 1 code.

Assuming the decimal registers are capable of adding or subtracting 3 on one clock period, the conversion time for decimal to binary is $(1 + 2n)t$ where n is the number of bits in the target binary number and t is the duration of a clock period. For the reverse conversion, the time is $(2 + 2n)t$. For a clock period of 5μs and a 40-bit word, these times are 405 and 410μs, respectively.

The author is grateful to the Defence Research Board of Canada for permission to publish the above work.

REFERENCES:

1. LYNCH, W. C. On a wired-in binary-to-decimal conversion scheme. *Comm. ACM 3* (1962), 159.
2. FLORIDA, C. D. Decimal to binary and binary to decimal conversion methods with particular reference to their use in floating point computers. DRTE Report EL5059-4, Oct. 1955.
3. LAKE, G. T. A digital decimal to binary and binary to decimal converter. DRTE Report No. 1044, July 1960.
4. FLORIDA, C. D. The DRTE solid state digital computer. Proc. Computing and Data Processing Society of Canada (1960).
5. FLORIDA, C. D. A floating point arithmetic unit. DRTE Report, No. EL 5083-7, Feb. 1959.

G. T. LAKE
Defence Research Telecommunications Establishment
Ottawa, Canada

Computation of *e* on Variable Word Length Machine

Dear Editor:

We wish to join Fred Gruenberger of the RAND Corporation in praising the variable word length IBM 1620. We have the value of e to 9790 decimal places, computed on the basic machine with 20,000 memory storage positions and automatic divide feature at Wabash College. The computation was done by summing one over $n!$, finding each term by dividing the previous term by the appropriate integer, continuing until the quotient was zero. The last divisor was 3191. The program took advantage of the wrap-around feature of the memory and the automatic divide instruction (not programmed division) to work with a dividend and quotient length of 9797 places and a divisor length of 4 places.

The time of execution was $18\frac{1}{2}$ hours plus output time. Thus the main loop (divide, test for zero quotient, add in latest term, reload dividend, increment divisor, branch to divide instruction) required an average of 21.5 seconds to execute.

Our value checks exactly to 2553 places with the Ballistic Research Laboratories table of e to 2556 places (published by the National Bureau of Standards). The discrepancy is probably due to round-off or truncation error in their value. We have not yet checked our value farther.

DANIEL HERRICK ('63)
NEAL BUTLER ('65)
Wabash College
Crawfordsville, Indiana