

A design for an auxiliary associative parallel processor

by M. A. WESLEY, S.-K. CHANG and J. H. MOMMENS

*IBM Thomas J. Watson Research Center
Yorktown Heights, New York*

INTRODUCTION

The use of highly parallel processing units for computing problems that are highly parallel in structure has been widely studied. The range of systems varies from the duplication of complete processing elements,¹ through the provision of a set of specially tailored small processors attached to a main processor,² to the use of cellular arrays;³ other writers have exploited the inherent parallelism of associative memories as components of parallel processing systems.⁴⁻⁷

Associative memories have been proposed either as true content addressable memories,⁵ or as processing units.⁴⁻⁶ In general, for use as a processing unit, each word in the memory, or possibly pairs or groups of words, is regarded as a serial by bit processing unit, all operating in parallel and controlled by a single program. These proposals have included rather complicated control systems to perform bit indexing and other functions necessary to sequence the memory through a program.

An important extension to the concept of associative memories as processing elements was proposed by McKeever,⁸ who described the use of three state storage elements with increased logic function at each storage cell; a memory with this feature is referred to here as an associative functional memory. The use of three state cells as a general system technology for conventional sequential processors has been described;^{9,10} it is the purpose of this paper to demonstrate that:

1. An associative functional memory with suitable peripheral features could be used to implement many of its own control functions as well as performing processing operations, and could readily be assembled into a complete auxiliary parallel processor,
2. Such a processor would be an attractive means of enhancing the performance of small conventional processors in a wide range of problems.

SYSTEM DESCRIPTION

The associative processor to be described here is intended for use as a programmable auxiliary processor to assist a conventional main processor in special problems. Programs are loaded from the main processor and are used to load data, to process it, and to return results to the main processor. The main processor has at all times the ability to force the auxiliary processor to accept a new program or to branch to a specified location in its program. For applications involving the processing and reduction of very large amounts of raw data, for example, radar signal processing, it would be wasteful to transfer data to the associative processor by way of the memory and channels of the main processor. In these circumstances, the associative processor could be modified to accept data directly from its source, that is, to act as a pre-processor, but would not be expected to exercise control over the data source.

The overall design goals have been simplicity of implementation and generality of application. Simplicity of implementation has been achieved by construction from units which could be standard modules⁹ with a minimum of additional special logic, and has led to a potentially fast cycle time. Generality of application has been achieved by implementing many control functions in memory and by the inclusion of some extra associative memory features which are not necessarily required in all applications. The proposed processor consists of two main components (Figure 1): a 1024 word \times 64 cell associative functional memory and a 512 word \times 50 bit read/write control store. The associative memory is used to store both data being processed and control information. An alternative would have been to have used separate memories; however, the use of a single unit permits the ratio of data to control information to be tailored to any given problem and enables a very simple control system to be used. On the other hand, the single array approach reduces the speed of data processing since many associative

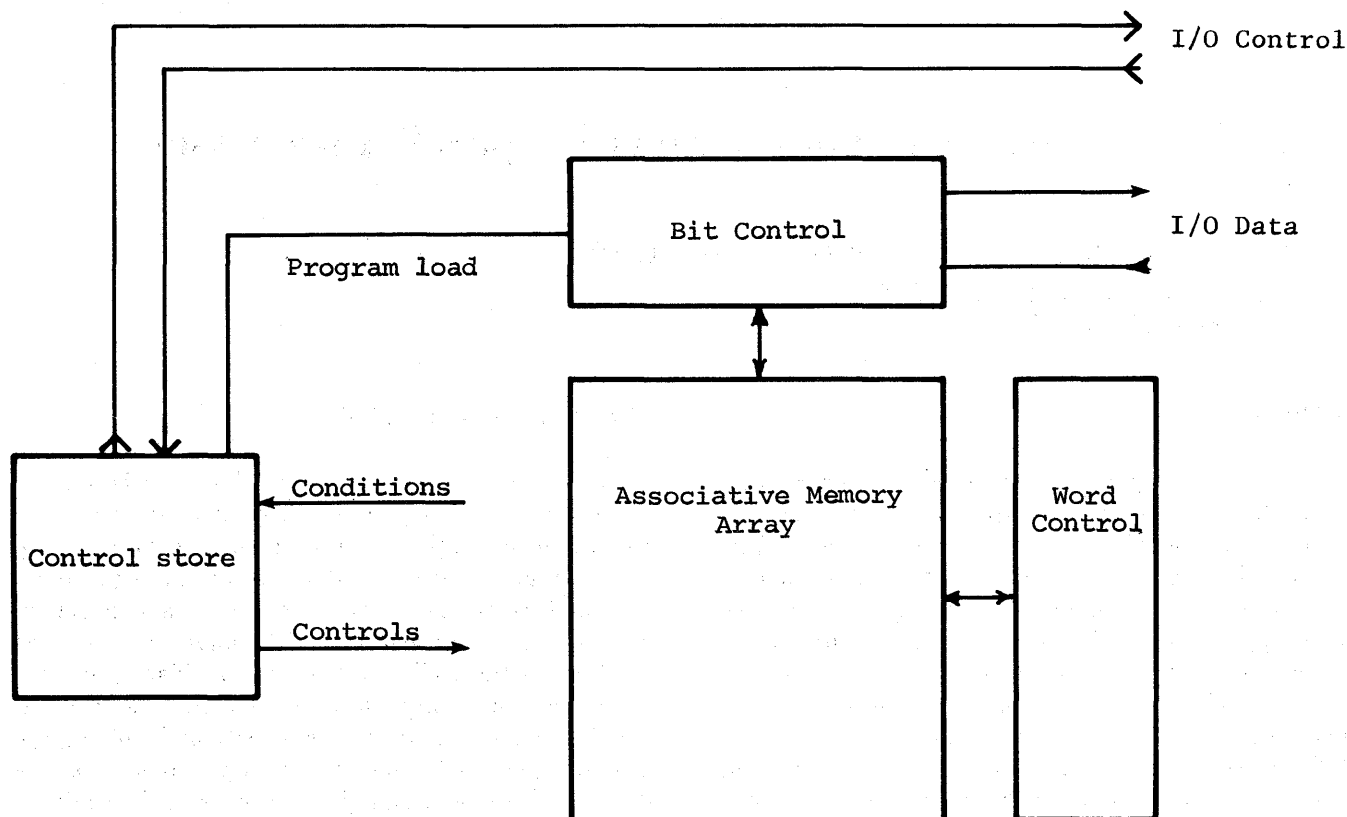


Figure 1—Block diagram of the proposed associative processor

memory cycles have to be used for control operations. It tends to be wasteful in the use of associative cells for control tables, and requires the introduction of extra features to reduce the interference between data and control.

Control sequences for the execution of a program are contained in a read/write control store normally operating in a read-only mode. Conditional branches in the program may be made by testing the condition of various signals in the processor and its I/O interfaces. Program loading, i.e., writing into the control store, is performed under the control of a short, permanent, initial load program.

Input and output data transfers are made by way of the associative array bit control unit. Basic interface control is carried out by the control store which can generate outgoing and test incoming control signals; more complex I/O control, such as an IBM Standard Interface, requires the addition of an interface control unit. Attachment closer to the main processor (e.g., interfacing the main memory) would give higher performance but would imply modifications to the main processor.

Associative processing array

The associative processing array is a two-dimensional array of three state (0, 1, X="don't care") associative storage cells with arbitrarily chosen dimensions of 1024 words \times 64 cells. The array is connected in the word direction to the word control unit and in the bit direction to the bit control unit. In an LSI implementation, the basic module could be a self-contained associative functional memory unit of, say, 128 words, complete with bit and word controls. Modules could readily be extended in the word direction by suitable interconnection of data and control lines; extension in the bit direction may be simulated by software.

Three basic operations may be performed on the array: search, read, and write.

Search

A ternary search argument is generated in the bit control unit between the specified data register (R1, R2) and the specified mask (M, all 1's, all 0's) on a bit

by bit basis:

	Data		
	0	1	
Mask	0	X	X
	1	0	1

X = don't care

Generation of search arguments.

All cells, in parallel, compare their contents with the search argument for that bit column and generate a mis-match signal in accordance with the truth table:

	Cell Content			
	0	1	X	
Search Argument	0	0	1	0
	1	1	0	0
	X	0	0	0

Generation of mismatch signals

Mismatch signals for a cell are ORed to give a mismatch signal for the word; word mismatch signals, in true or complement form, are sent to the word control unit where they may be ANDed or ORed with, or replace the contents of one of two sets of selector latches (P and S).

Read

The contents of a specified set of selector latches (P, S, all 0's) in true or complement form are used to select words to be read. The contents of cells from selected words are ORed in the bit direction onto a read bus (an X state reads as zero) and sent to the bit control unit where they are used to load a specified register (R1, R2, M) based on the value of mask specified (M, all 0, all 1):

	Mask		
	0	1	
Read Bus	0	No change	0
	1	No change	1

Effect of Read operation on specified register.

Write

Two write commands are provided: Write Normal, and Write Special. In either case a ternary argument is generated in the same manner as a Search argument and acts on the contents of cells in selected words, as defined by the specified selector register in true or complement form (P, S, all 0). The effects on a cell are shown in the two truth tables below:

	Write Argument				Write Argument			
	0	1	X		0	1	X	
Cell Content	0	1	No change		X	X	No change	
	Write Normal				Write Special			

The word control unit may also perform a one bit shift of a selector register up or down with end around carry, or fill with 0 or 1; a shift takes the same time as an array operation or may be overlapped with an adjacent preceding array operation using the same selector register. This provides the only parallel means of communicating vertically between words. Other writers (e.g., McKeever, Reference 8) have usually specified other operations in the word control unit, such as isolate first match. Although provided by our simulator we have found little use for such operations, which tend to be serial in nature, and for the most part found that they can be economically simulated by software, e.g., by use of a code field. The exception was sorting with an arbitrary number of identical items, when a means of separately identifying multiple matches is necessary.

The bit control unit contains three registers: two data registers (R1, R2) defining a data source or sink for an array operation; and one mask register (M) defining a field for an array operation. Any array operation may use either data register and the mask register, or may replace the mask by a source of all 0's or all 1's. In addition, the control store may specify directly the leftmost four bits each for the mask and data registers. These bits (the immediate field) are ORed into the register outputs without affecting the contents of the register. A non-array operation, a single bit shift operation on any register may be specified; this feature is assumed to take the same time as an array operation unless it is overlapped with an adjacent array operation in which the register being shifted is a data source or sink; again, fill with 0 or 1 may be specified.

Input-output operations

Input-output operations for the associative processor take place through the bit control data register R1. The register is divided into fields each of the same width as the I/O interface data busses. Data may be gated to or from the register under program control and is interlocked with the main processor by interface synchronizing signals. Outgoing interface control signals are generated by the control store and by the run control logic. Incoming interface control signals are either tested as machine conditions by the program, or act directly on the run control logic.

Operation as a pre-processor, taking data from but not controlling another source, would require the ability to transfer into the processor from another interface and generate and test another set of I/O synchronization signals. This modification requires at least two extra bits in the control word and some extra logic, but is not expected to be very difficult to implement.

Control store

The control store (Figure 2) is a conventional (as opposed to associative) read/write store used to hold a program defining the sequence of operations to be performed by the associative memory.¹¹ During the execution of a program, the control store normally operates in a read-only manner. Each word read out specifies the operations to be performed in the array and also the address of the next program word. The next address may be modified by a condition in the machine, specified by the program word, enabling conditional branches to be made in the program.

The control store contains 512 words of 50 bits, though these numbers may vary, depending on the features included. When formed into groups of mutually exclusive options, the operation options to be specified for the array processing unit fall into rather small groups, so that coding within a group is not very advantageous, and bit significant operation has been chosen. This has other advantages as it increases flexibility and eliminates timing delays through decoders.

It is expected that a semiconductor memory will be necessary to be able to operate at the same speed as the array. Such a memory will have nondestructive read out so that writing into the control store will require special control features. Subroutining capability is provided by a data path to the bit control register R1, enabling subroutine return addresses to be stored in the associative array.

Program loading

Program loading is performed under the control of a small fixed routine held in the first few words of the control store. The program load routine assembles data from the I/O interface into the bit control register R1. This data is interpreted as a control word and the address of the location in the control store into which it is to be stored. The program load routine then gives a special signal "write next cycle" which causes the run control logic to break its normal cycle of read-only operation, and to spend one cycle writing into the control store from R1. Note that the control store data register is not altered and is available for normal operation on the cycle after the write operation is performed. The "write next cycle" control also permits the transfer of programs from the associative array to the control store.

Programming techniques

The guiding principle behind the design of the control system has been to make the hardware simple whilst keeping the system flexible. This principle led to the use of a single associative memory, controlled by a single conventional control store, with both data and control information stored and processed in the associative memory.

Three classes of control information are held in the associative memory:

- (1) mask and data register contents for operating on data. In the case of relatively simple operations, such as addition, these register contents are stored in consecutive locations in the sequence in which they will be needed, and are accessed by shifting a selector register reserved for the purpose. In more complicated operations, such as multiplication, where the total number of masks is proportional to p^2 (where p is the field width) and may be large, it may be advantageous to process the masks as data in the manner described in References 9 and 10, and to generate the required sequence of masks; the number of control words now becomes essentially proportional to p .
- (2) program flow logic, including counts and logical decisions. These may be programmed directly or, in simple cases, may be implemented by inserting blank words in mask sequences and test-

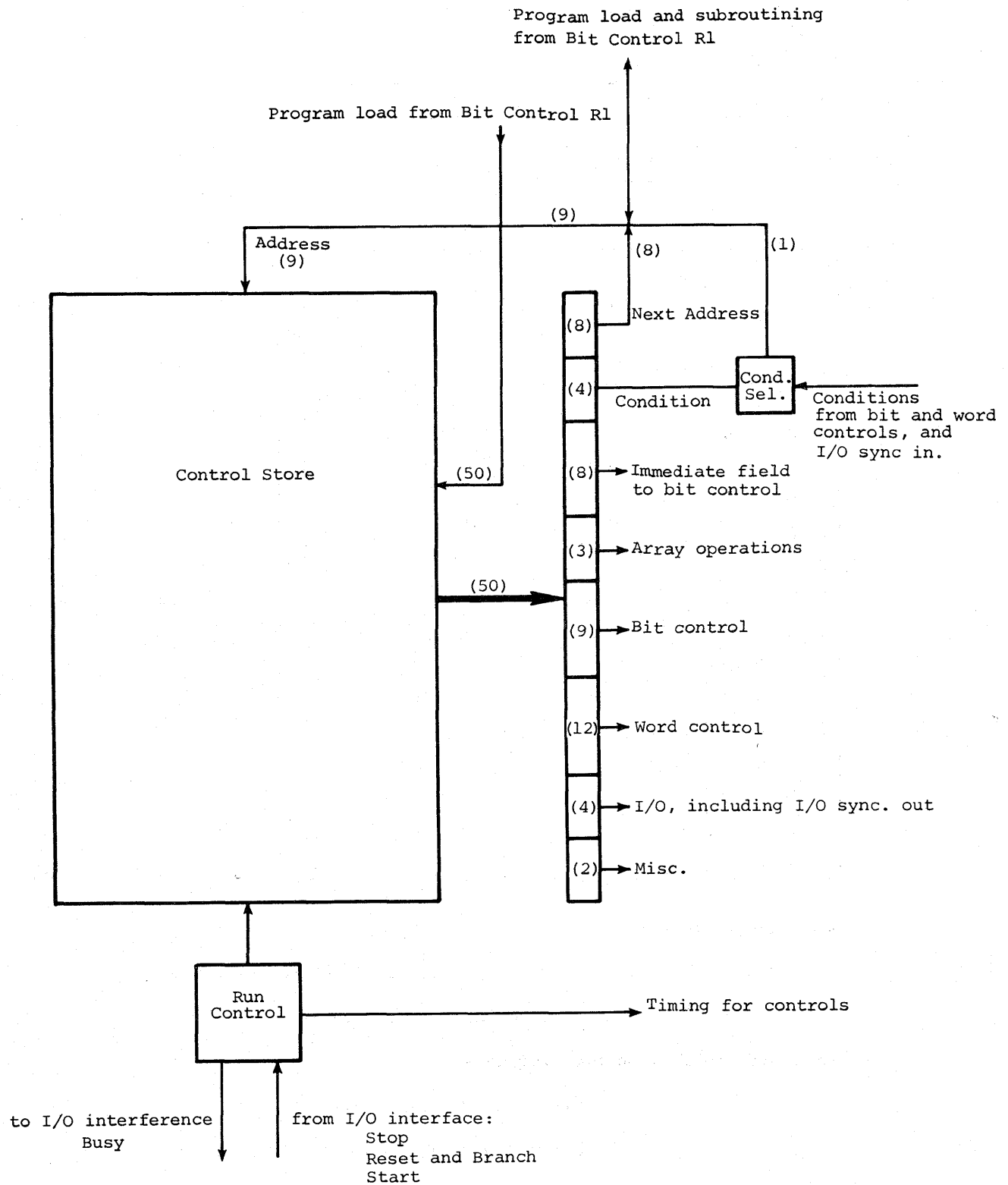
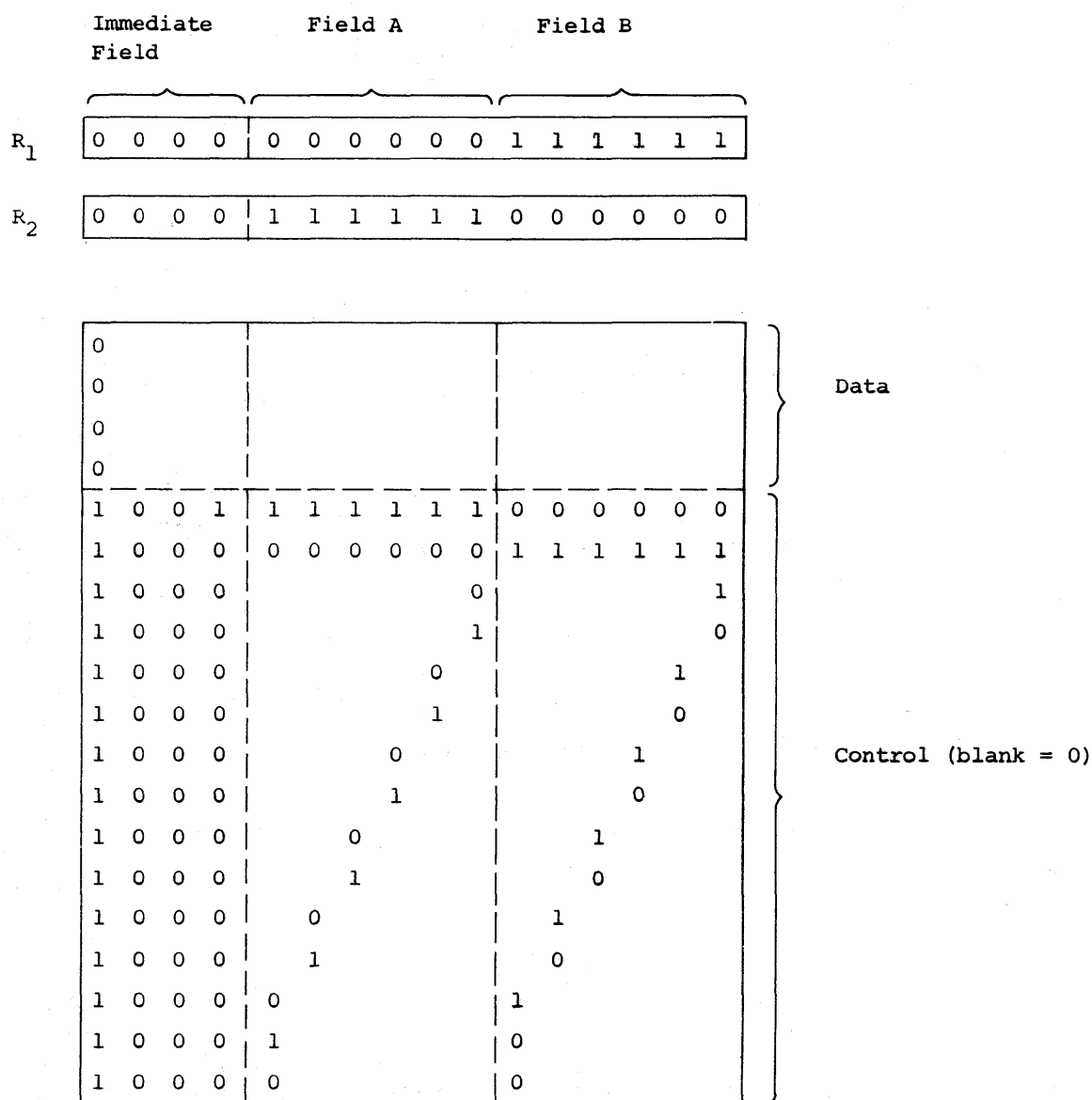


Figure 2—Control store connections



Immediate field codes: 0 - - - data word
 1 0 0 1 start of control sequence
 1 0 0 0 control word

Immediate field allocation for data words:

bit 1 0 : data word
 bit 2 0/1 : not active/active marker
 bit 3 0/1 : carry 0/I

Figure 3a—Memory organization of addition: $A' = A + B$

Location	Operation	Immediate Field	Selector	Data Source or Sink	Mask	Next Address	Comments
14	Search	1001	P	-	0	15	load R1, R2, M. keep mask table pointer in P.
15	Read	0000	P, shift down	R2	1	16	
16	Read	0000	P, shift down	R1	1	17	
17	Read	0000	P, shift down	M	1	18	
18	Search	0---	S	-	0	19	reset carry and active markers in data words
19	Write	-10-	S	-	0	20	
20	Search	010-	S	R2	M	22	identify no change combinations and mark as inactive
22	Search	011-	OR into S	R1	M	23	
23	Write	00--	S	-	0	24	
24	Read	0000	P, shift down	M	1	25	read new mask
25	Search	01--	S	R2	M	26	identify a field bit changing to 0; update
26	Write	001-	S	R1	M	27	
27	Search	01--	S	R1	M	28	identify a field bit changing to 1, update
28	Write	000-	S	R2	M	29	
29	Search	0---	S	-	0	30	set active markers
30	Read	0000	P, shift down	M	1	31	read new mask
31	Write	-1--	S	-	0	20/21 (M=0)	test for mask = 0

Figure 3b—Program for addition: $A' = A + B$

ing for an all zero read out. Note that the only internal condition tests available to the programmer are zero tests on the bit and word registers; an alternative would have been a test on a single bit.

- (3) partitioning. The immediate field provides a fast software technique for partitioning the single array into groups of words. The four bits of the field permit 16 interleaved partitions of arbitrary size. This feature is particularly valuable for distinguishing and separating data and control information; for example, a 0 in the leftmost bit position may signify data while a 1 signifies control.

A further consequence of the use of a single array is the need to load and store the mask register from and to the array. The three array operations have been generalized for this purpose.

Programming example: Serial-by-bit addition

This example is given to show:

- (1) the use of the immediate field
- (2) the use of the associative array for both data and control information
- (3) the ability to define fields independently of the program by means of control tables.

Suppose we wish to perform the addition of two fields, A and B , the result to overwrite field A , i.e., $A' = A + B$. The minimum possible number of array operations per bit is 6 (4 Search and 2 Write); however, this assumes no performance loss handling control operations. The addition algorithm given below takes 11 operations (9 if the inner loop is expanded to handle two bits consecutively). The algorithm uses $2p+3$ memory words to store masks and data register contents (p is the field width); we have found that, in general, it is possible to trade less speed for less control storage.

The algorithm is illustrated in Figure 3. The first six instructions locate the start of an addition control table, load the two data registers R1 and R2 with constants which remain unchanged throughout the algorithm, load an initial pattern into the mask register, and initialize the immediate field. Three bits of the immediate field are used: bit 1 indicates data or control words, bit 2 is an activity marker used to indicate whether a word has been completely processed in the current bit position, and bit 3 is a carry and is initially zero.

Instructions 20-31 make up the main loop of the algorithm which proceeds in a serial by bit manner starting with the least significant bit. At each bit position the no change condition in the A and carry bits is detected, and these words are marked as inactive. The remaining words are tested for changes in the A field and are updated. Indexing across the fields is achieved by the mask register contents, which are read sequentially from the control table. Execution of the loop ceases when an all zero mask is read out.

APPLICATIONS

The principal mode of parallel processing employed in this associative processor is serial by bit, parallel by word, over some selected subset of words in the memory. Thus a memory of 1024 words has a potential processing parallelism of up to 1024. Operating in a serial-by-bit manner across fields inherently requires more cycles than a conventional machine with bit parallel processing. This is particularly significant in arithmetic operations; for example, 16 bit addition requires about four times as many control cycles as a System/360 Model 30, 32 bit addition requires about eight times as many, and this must be more than cancelled by the parallelism used. At present we are limited to fixed or block floating point operation; normalization in general floating point is prohibitively time consuming. In bit manipulation operations, the programmable field feature (i.e., the ability to define fields by mask control tables stored in the associative memory) may enable the associative processor to take fewer operations than a sequential machine.

The overall performance of the associative processor is affected by a number of overheads. It is assumed that the processor would be used for repetitive execution of a program, so that program and control table loading times need not be included in the problem-solving time. Input and output of data is sequential by word and can be very significant. In general, the processor as described with a single I/O data path is only suited to problems with a high processing to I/O ratio; however, multiple

I/O data paths could be provided to each of a number of partitions. After each stage of parallel computation (e.g., after a vector addition) it is generally necessary to reorganize the data for a subsequent stage of processing; this too can use significant amounts of time and must be minimized by careful algorithm selection and memory organization.

The performance of the processor has been studied with the assistance of a very flexible simulator program which allowed function truth tables to be defined at object time. Execution times, including processing, input/output, and data reorganization, have been computed assuming a cycle time of 100 nsec, which is believed to be within the capability of an LSI technology.

A wide range of examples have been studied for the associative processor and are discussed here without details of programming techniques. The aim in choosing examples has been to investigate the versatility of the associative processor and to demonstrate its performance on problems for which special purpose processors are being built. The examples are summarized in Table I; performance figures for the associative processor are based on a cycle time of 100 nsecs and an I/O data rate of 1.5 μ secs per byte.

Picture processing

The functional memory may be regarded as a two-dimensional array of storage cells. Given a memory with suitable dimensions, two-dimensional pictures may be stored in two-dimensional form and, since neighboring point relationships are preserved, local processing operations may be performed directly and with a high degree of parallelism. Analog picture element values may be coded into a number of adjacent bits in either the bit or word direction; pictures too large for the memory may be partitioned and processed in separate pages, but this requires care in piecing the edge results together.

As an example, consider the application of a two-dimensional binary mask operator ($n_x \times n_y$) to a binary picture ($N_x \times N_y$) stored in the functional memory. The algorithm proceeds by searching sequentially for each line of the operator, centered on one column of the picture. The result of the first search operation is loaded into a selector register and shifted one position; the results of subsequent searches are ANDed into the previous selector register contents before shifting. After n_y search operations, the selector register contains the full result of applying the operator to the column and may be either stored back into the memory or output; further columns may be processed sequentially. With $N_x = N_y = 144$, application of 25 operators with $n_x = n_y =$

7 takes 120 milliseconds and is estimated to be 610 times faster than a 360/30. Note that this problem gains performance through both the parallelism of the associative processor and its ability to tailor data fields to the needs of the current algorithm.

An alternative approach, suitable for on-line character recognition, would be to exploit the symmetry of the picture-operator system and hold the mask operators in the memory and search them with the picture as received from a scanner. This operation is the "feature extraction" process of character recognition; the resultant feature vector may subsequently be matched against a stored library of standard reference feature vectors; in both operations the three call states may be used to represent ternary data. Distance measures between the feature vector and all the reference vectors may then be computed in a serial-by-bit manner; the recognition process may be completed by testing for the minimum distance using parallel search techniques.

Lewin sorting algorithm

The Lewin sorting algorithm¹² was originally proposed for an associative memory with a special hardware feature to indicate whether a column contained all 0's or all 1's. This feature may readily be simulated by software on this processor; for example, searching for 1 on a data column and a subsequent read of a marker column containing all 1's will indicate whether or not the data column contains all 0's. An all 1's condition may be similarly detected.

The algorithm finds, for example, the largest of a set

TABLE I—Summary of the Performance of the Associative Processor

	Distribution of total processing time			Total Processing Time	Speed Up Over 360/30
	Processing	Reorganization	I/O		
Picture Processing	97%	—	3%	122 millisec.	610×
Sorting	70%	—	30%	20 millisec.	110×
Matrix Mult.	31%	7%	62%	1 millisec.	78×
Fourier Transform	17%	44%	39%	31 millisec.	75×
Hadamard Transform	4%	46%	50%	12 millisec.	79×
1-D Filter	40%	—	60%	10 millisec.	280×
2-D Filter	50%	—	50%	20 sec.	510×

of numbers by searching for columns containing a mixture of 0's and 1's. If no such columns exist, all the numbers are identical and are equal to the largest one. Otherwise, the leftmost mixed column is searched for numbers with 1 in this position, and the operation is repeated on this new subset.

The number of operations taken by the associative processor to execute the algorithm is very data dependent; worst-case figures are given in Table I for an internal sort of 1000 items using 16 bit keys and show a speed up of a factor of 110 over a 360/30.

As mentioned previously, a sort of identical items requires a means of isolating the components of multiple matches; in this example, where an arbitrary number of identical items may be present software techniques require a wide code field and are therefore expensive. We have assumed the existence of an isolate first hit feature.

Tree searching

One of the major problems in artificial intelligence is to perform efficient tree searching. Since the number of nodes of a tree grows exponentially with respect to the depth of the tree, the tree searching time also increases exponentially, rendering deeper search impractical. It is clear that in tree searching the same sequence of computation and condition testing is performed on every node. Thus the basic requirement of "Single Program Multiple Data" processing is satisfied and we can perform computations upon all nodes in parallel. The tree may still have to be grown step by step, but this is probably unavoidable.

It is difficult to define a typical tree searching problem and, since performance of both the associative processor and a conventional processor are highly problem dependent, no performance comparisons are given. However, we note that the performance improvements in the region of 2-3 orders of magnitude have been found in simple game-playing problems.

Matrix operations

Many matrix operations are inherently parallel in nature and may readily be programmed for the associative processor. Vector addition, subtraction, and multiplication operations, and summation of elements of a vector, may be executed very efficiently; division may be performed only with difficulty. Thus, matrix multiplication is very attractive, but operations involving a high proportion of divisions is not likely to show any great advantage on the associative processor. When only a small number of divisions are required.

they may be performed by the main processor (e.g., pivotal element normalization in matrix inversion²).

Fixed point multiplication of 10×10 matrices at 16 bit precision gives a performance improvement of 78 times. Larger matrix sizes may be partitioned to fit the processor and show approximately the same processing performance improvement because I/O time dominates.

Fast Fourier and Hadamard transforms

The fast Fourier¹³ and Hadamard¹⁴ transforms are closely related operations used particularly in signal and image processing. The radix-2 fast Fourier transform computes the Fourier transform of a set of points $A_1^0 \dots A_n^0$ by means of a sequence of transformations $A^0 \rightarrow A^1 \rightarrow \dots \rightarrow A^{m-1}$, where $n = 2^m$. Each of these transformations is made up on $n/2$ pairs of elementary operations of the form

$$A_p^{i+1} = A_p^i + W_p^i A_q^i$$

$$A_q^{i+1} = A_p^i - W_p^i A_q^i$$

where W_p^i is a complex 2^{i+1} th root of unity in the fast Fourier transform, and 1 in the Hadamard transform. Each of the pairs of elementary operations in a transform may be performed in parallel and consists of a complex multiplication followed by a complex addition and subtraction; the result of a transformation may overwrite the input to the transformation.

Many algorithms have been proposed for the selection of the pairs of indices p and q . The procedure chosen for use here selects the indices in a regular manner and allows efficient use to be made of the select latches as a means of parallel communication between words. For the first transformation $A^0 \rightarrow A^1$, A_p and A_q are $n/2$ words apart; for $A^1 \rightarrow A^2$, $n/4$ words apart, etc. However, this procedure has the disadvantage that if the input data are in order, the results will be permuted with their addresses in bit reversed form, though this may be corrected when the results are transferred back to the main processor. The basic steps of the algorithm have already been described¹⁵ for an associative processor with external storage and separated data and control functions.

The implementation of a useful size of Fourier transform within this associative processor requires the use of a larger memory array. The principal reasons are the need to store the complex roots of unity and the inclusion of an address field to enable blocks of operands to be identified rapidly. A 1024 point complex transform with 14 bit precision may be fitted into an associative memory of 1273 words of 89 bits with a performance approximately 75 times faster than a 360/30.

The Hadamard transform may be regarded as a

square wave analog of the sine and cosine wave Fourier transform and has many advantages from a computational point of view. In particular, the use of square waves of amplitude ± 1 makes multiplication unnecessary, and an ability to generate square wave transition lengths for a transform of length $2N$ from a transform of length N removes the need for a stored table of coefficients. The Hadamard transform also has a fast Hadamard transform algorithm. Performance on the associative processor for a 1024 point real transform is shown in Table I. Note that in both these transforms data reorganization becomes very significant.

Digital filtering

Digital convolutional filters of the form:

$$y(t) = \sum_{\tau=1}^n x(t-\tau)g(t)$$

where $g(t)$ is a filter of length n

$x(t)$ is the filter input

$y(t)$ is the filter output

may be implemented on the associative processor in a number of ways, the choice depending principally on the dimensions of the problems, e.g., filter length, data record length, and number of filters. The most efficient method, in the sense that I/O operations are minimized, is to store the filter vector g permanently in the memory and to regard the data points as scalar inputs operating on all elements of the filter vector. This method is applicable when $N \approx$ number of words in the memory, where N represents either a single long filter or a number of shorter filters of equal length. Note that, in the Single Program Multiple Data form of parallel processing, a scalar operation on a vector differs from element operations between a pair of vectors in that it is now possible to perform look-ahead operations when processing the scalar quantity, thereby approximately halving the execution time.

The algorithm assumes that the memory is partitioned into two fields of equal size, one for the filter vector g and the other for partial results. Processing proceeds in a pipeline manner—a new data point is received and used as a scalar multiplier on the filter vector, the products being added into the adjacent partial result field. The partial results are shifted one word position and the process repeated with the next data point. After the first n data points have been processed, one output result will be available for each filter held in the memory; thereafter, output results are available after each new data point has been processed.

An alternative method to be used when the filter is short is to load the memory to capacity with data points and to apply the filter coefficients as external scalars. When the whole filter has been applied, all the results may be read out. The processing time for this method is the same as that for the stored filter, but the I/O time is significantly greater.

Two examples have been considered, both using the stored data method. The first is a typical seismic signal processing problem and has a 1000-point data record, a 25-point filter, and operates at 16 bit precision. The second is a picture processing problem similar to that posed by Mariner pictures with a picture of 600×684 elements, a two-dimensional filter of 15×15 points, and operates at 8 bit precision.

The results are shown in Table I. In spite of the I/O overheads, the performance improvements are large; in particular, the space picture processing performance reflects the ability of the associative processor to tailor its field lengths to the problem.

Convolutional decoding

In this example, the associative processor is used to perform error-correcting decoding operations. The Viterbi decoding algorithm¹⁶ is given as an example; however, in order to understand the decoding algorithm it is necessary to first describe the coding process.

The encoder has the canonical form of a shift register of length S . Each time an information digit is encoded, the contents of the shift register are shifted right, the rightmost bit being discarded, and the information digit is stored in the leftmost bit of the register. The encoded message bits are the modulo 2 sums of some bits in the shift register; the ratio of information bits to encoded message bits is known as the code rate.

The present contents of the shift register may be regarded as the state of the encoder, and a state transition diagram may be constructed for every input of an information bit. The Viterbi decoding algorithm is based on storing, for each possible state of the encoder, a history of the most probable, in some sense, sequence of information digits to reach that state. Each state and its history has a distance measure associated with it. When a new set of encoded message bits are received, the histories are updated by computing the error distance between the received bits and the true bits corresponding to each state transition, and adding this to the distance measure for the corresponding history. The histories are arbitrarily restricted to a length $3(S-1)$ and, after each updating, the bit $3(S-1)$ bits away in the history with the lowest distance measure is output as a decoded bit.

A number of examples have been studied with various values of S and code rate. Comparisons have not been made with a conventional machine because special purpose processors are being built for these decoding problems. For $S=6$, rate $=\frac{1}{2}$, the associative processor takes 100 μ seconds per bit, i.e., 10K bits/sec. For $S=9$, rate $=\frac{1}{3}$, the processor takes 370 μ secs. per bit, i.e., 2.7K bits/sec. This variation in performance with shift register length S is almost entirely caused by an increase in data reorganization overheads caused by a larger number of encoder states.

CONCLUSIONS

The auxiliary associative processor described in this paper has been shown to have a high performance on a wide range of problems which are inherently parallel in structure. The major drawbacks have been found to be in the processor's ability to handle only fixed point or block floating point arithmetic, and the difficulty of performing division. The principal system problems have been in the operating overheads of I/O and data reorganization. The I/O overhead could be reduced by integrating the associative processor into the main processor, which would also permit more complex interaction between the processors or by providing multiple I/O paths. The data reorganization overhead is caused mainly by long shift operations in the selector latches; these could be reduced by hardware and/or software partitioning of the memory, enabling inactive blocks of words to be by-passed. In Reference 17 the data reorganization problem is studied in detail.

The consequences of using a single array to hold both data and control information are hard to isolate. In operation time the overhead for control operations is always less than 50 percent (arithmetic operations) and is generally much less. In memory space, the price is at least one bit of each word (the immediate field) and up to 25 percent increase in size (Fourier transform). In contrast, the flexible partition between data and control, and the ability to tailor fields to the problem have proved very powerful.

The decision to use the three-state cells of McKeever⁸ was based on the aim for generality of application. In parallel binary arithmetic operations only two of the three states have been used; however, the third state has been used for data representation in picture processing and tree searching, and for implementation of control functions in all examples. In practice the three-state cell may be implemented by 2 two-state cells; the possibility then exists of having two-state cells individually for 2 state operations, and in conjunction for larger numbers of states. In this paper we have not pursued such an approach.

Economic realization of the processor requires the availability of high performance, low-cost integrated circuit technologies. However, the system design has aimed at the use of only a small number of different components, most of which are memory rather than random logic. The components used could be a standard technology suitable for both conventional and parallel systems.

The performance figures quoted in this paper have been obtained with the aid of a software simulator at the microprogram level. Little work has been done on the development of a higher level language or assembler for the processor.

ACKNOWLEDGMENT

We are indebted to Dr. R. Lyons for drawing the Lewin sorting algorithm to our attention and pointing out its suitability for execution on an associative functional processor.

REFERENCES

- 1 D L SLOTNIK W C BORCK
R C MCREYNOLDS
The Solomon Computer
Proc FJCC pp 97-107 1962
- 2 B A CRANE J A GITHENS
Bulk processing in a distributed logic memory
IEEE Trans on Elect Computers Vol EC 14 pp 186-196
April 1965
- 3 J H HOLLAND
A universal computer capable of executing an arbitrary number of sub-programs simultaneously
Proc FJCC pp 108-113 1959
- 4 G ESTRIN R FULLER
Algorithms for content-addressable memories
Proc IEEE pp 118-130 Pacific Computer Conf 1963
- 5 R G EWING P M DAVIES
An associative processor
Proc FJCC pp 147-158 1964
- 6 R H FULLER R M BIRD
An associative parallel processor with applications to picture processing
Proc FJCC pp 105-115 1965
- 7 J A GITHENS
An associative, highly parallel computer for radar data processing
Parallel Processor Systems Technologies and Applications
editor L C Hobbs pp 71-86 Spartan Books 1970
- 8 B T MCKEEVER
The associative memory structure
Proc FJCC pp 371-388 1965
- 9 M FLINDERS P L GARDNER J G MINSHULL
R J LLEWELYN
Functional memory as a general purpose systems technology
1970 IEEE Computer Group Conference June 1970
- 10 P L GARDNER
Functional memory and its microprogramming implications
IEEE Trans on Computers Vol C20 No 7 pp 764-755
July 1971
- 11 D A SAVITT H H LOVE
Association storing processor study
Hughes Aircraft Technical Report No TR-66-174
(AD 488538) June 1966
- 12 M H LEWIN
Retrieval of ordered lists from a content addressed memory
RCA Review June 1962 pp 215-229
- 13 G-AE SUBCOMMITTEE ON MEASUREMENT
CONCEPTS
What is the fast Fourier transform
IEEE Trans Audio and Electroacoustics Vol AV-15 pp 44-55 June 1967
- 14 W K PRATT J KANE H C ANDREWS
Hadamard transform image coding
Proc IEEE Vol 57 No 1 Jan 1969 pp 58-68
- 15 M A WESLEY
Associative parallel processing for the fast Fourier transform
IEEE Trans on Audio and Electroacoustics Vol Au-17 No 2
pp 162-165 June 1969
- 16 A J VITERBI
Error bounds for convolutional codes and an asymptotically optimum decoding algorithm
IEEE Trans on Inf Theory April 1967 Vol IT-13 No 2
pp 260-269
- 17 S K CHANG
Parallel computation of local operations
Proc Third ACM Symposium on Theory of Computing
May 1971 pp 101-115