A Content Addressable Parallel Array Processor

Caxton Foster
Charles Weems
Steven Levitan

COINS Technical Report 83-32

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts, 01003

September 1983

Table of Contents

The papers contained in this collection were funded by a variety of granting agencies and sources. These are individually acknowledged in each of the papers.

A Content Addressable Parallel Array Processor

- SIMD Parallel Processor
- 512 x 512 (262,144) Processing Elements
- Rectangular (4-way) Interconnect
- Fast (30mS) Load/Dump Time
- Broadcast Data
- Some / $\overline{\text{None}}$
- Responder Count
- Versatile Control of Edge Connect; Row, Column and PE activity
- 40 Billion Fixed Point Additions per Second

The Titanic Parallel Array Processor is a Single Instruction Stream Multiple Data Stream (SIMD) machine which will be connected, as a slave device, to a VAX-11/780. Like the Floating Point Systems AP120B slave processor, it can be used for vector processing, but it is actually a general purpose parallel processor like the Illiac IV or Staran. The Titanic will be used both as a research project for the investigation of parallel processing systems and as a tool for other researchers who are investigating such computationaly intensive applications as static image processing, motion image processing, robotics, and neural-network algorithms.

The Titanic general purpose parallel processor is necessary because there are many problems which are so computationally intensive that even the fastest general purpose serial processors are unable to solve them in real time, for example, real time signal processing, computer vision, speech understanding and robot sensory analysis. Although special purpose machines exist to solve some of these problems in a limited domain, they do not provide the flexibility necessary to allow the exploration of new solutions to the problems or adaptation to new domains. The Titanic will provide this flexibility as well as the raw computing power necessary to solve these problems.

There are many problems which are still compute-bound, even in this era of Cray 2s and Floating Point Systems vector processors. Examples of such problems include image processing, motion analysis searching, sorting, pattern recognition, database management, etc. Speaking generally, the availability of a CAM enables one to approach the solution of a problem in ways that would be rejected out-of-hand in a von-Neumann organization. Below are some examples of the kinds of problems and solutions we have worked on to date.

* Storing and searching a dictionary of several thousand common English words, Wall found that the CDC-6600 might take up to several hours to solve a single simple substitution cryptogram. The same approach implemented on a CAM could solve the same cryptogram in approximately half a second. Suddenly, an "obvious" approach that had been completely impractical becomes very attractive.

* In a data base application, storing key descriptive words in the CAM, we can let a user enter information in very nearly normal English, pick out the key words he uses, and do an incremental search while he is still inputting his description or query.

* One of the problems that inhibits the use of LISP for real-time applications is due to the unpredictable and arbitrarily long delays that occur when main memory becomes full and garbage collection must be performed.

  A typical LISP cell in a CAM might contain three fields (at least) called "garbage" (one bit long), "left" (CAR), and "right" (CDR). When a free cell is needed, we search for a cell whose garbage bit is "1". Before we can use this cell, we must discover if it is the <u>only</u> cell in memory now pointing to the cell named in the "left" field (call it TARGET). In a CAM we can ask, in parallel, if there exist any cells which point to TARGET in either their CAR or CDR. If there are none, then TARGET is also a garbage cell, and we set its garbage bit. A similar search must be done for the cell pointed at by the "right" field. Once these four searches have been done, and garbage bits set if required, the original cell can be reused. This takes only four exact match searches. Thus, LISP becomes available for real-time use where arbitrary delays might be fatal.

* Local, window-type convolutions are very common in signal processing and image processing applications, as well as many other areas. Indeed, much of the interest in Fast Fourier Transforms is because once the transform has been carried out, a simple multiplication followed by an inverse transform can be used to perform a convolution. In a CAM we can perform a simple convolution in about one hundred microseconds directly on the image in question without having to perform the transform and its inverse. The is some times faster than is possible on, for example, a Floating Point Systems AP120B vector processor.

* In the analysis of images taken by a moving camera one tries to identify "interesting points". Successive positions of an "interesting" point establish a "flow-vector" for that point and a set of such flow-vectors, when projected, can be used to identify the FOE (focus of expansion) which is the point toward which the camera is moving. Due to noise in the images and inherent digitization noise, the set of flow-vectors will not all intersect at a single point. Taken pairwise, the flow-vectors will intersect in a set of points and the center of mass of these intersection points is a good estimate of the FOE. Such a center of mass can be found in a CAM in under a millisecond , or roughly 1,400 times faster than a conventional machine.

Besides our own research groups at the University of Massachusetts, groups at Digital Equipment Corporation and General Electric Corporation have shown interest in the research. These companies are both investigating computer designs which solve computationally intensive problems such as signal processing, machine vision and robot control.

Once the Titanic actually exists, even more progress will be made by researchers who have access to it. These people are committed to spending the time and energy necessary to develop new algorithms, new theories, and new applications which take advantage of the advanced architecture and processing power of Titanic.

Things that Titanic is good at

* Anything that an associative processor like STARAN is good at:

  - Database query/update
  - Text to speech synthesis
  - Real time LISP garbage collection
  - Radar analysis
  - Graph processing
  - Digital differential analysis
  - Air traffic control
  - Scalar arithmetic (scalar * vector)
  - Vector arithmetic (vector * vector)

* Anything that an SIMD parallel processing array like ILLIAC IV is good at:

  - Convolution
  - Relaxation
  - Simulation of planar dynamic physical systems (fluidics, weather, crystal lattices)
  - Modelling
  - Signal processing
  - Edge detection
  - Line thinning
  - Image processing

* Things that neither of the above architectures is good at but which the combination can do:

  - Segmentation
  - Converting a segmented image into a graph representation
  - Region specific adaptive image enhancement
  - Histogram analysis
  - Some forms of Hough transform
  - Hidden line/surface removal
  - Flow field determination from image sequences
  - Stereo image correlation / depth extraction

Things Titanic is bad at

* Anything that only an MIMD architecture is good at.

* Some things that a shuffle-exchange network is good at.

* Examples:

  - Distributed processing applications
  - Matrix multiply
  - Fast Fourier Transform
  - Anything which requires complex or random parallel communication between distant nodes in a network.

Note: Titanic gets around the inability to do Matrix Multiply and FFT by doing directly many of the operations that these are needed for in other architectures. For example, FFT's are commonly used to do convolutions in image and signal processing. Titanic simply does the convolve directly, avoiding the FFT altogether, which results in a substantial speed increase.

Titanic: A VLSI Based Content Addressable Parallel Array Processor*

by

C. Weems
S. Levitan
C. Foster

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

A design is presented for a Content Addressable Parallel Array Processor (CAPAP) which is both practical and feasible. Its practicality stems from an extensive program of research into real applications of content addressability and parallelism. The feasibility of the design stems from development under a set of conservative engineering constraints tied to limitations of VLSI technology.

*This work was supported in part by a grant from the Army Research Office DAAG29-79-G-0046.

A shortened version of this paper appeared in the proceedings of the 1982 International Conference on Circuits and Computers (September 29 to October 1, 1982, New York)

Origins

Starting in the Summer of 1979, after acquiring a small Content Addressable Memory (CAM), we conducted an extensive exploration of applications of content addressability and parallelism. During the ensuing three years some thirty applications have been developed with over a dozen being programmed to completion. All have been analyzed with an eye toward the design of more useful hardware. Application areas have included data base retrieval, LISP garbage collection, text-to-speech synthesis, and image convolution. Some of the results of this work are presented here as a rationale for some of our architectural design decisions.

More Cells With Less Memory

One of our major findings is somewhat counterintuitive. Normally, CAM designers give a large amount of memory to each cell of the CAM. This is so that each cell may hold a large record of logically associated data. Such a design attempts to maximize the benefits of the CAM's associativity. We have found, however, that a majority of interesting CAM applications require only one to sixteen bytes of memory in each cell, and that these applications benefit much more from the added parallelism of having more cells. Further, we have found that those applications which require more memory in each cell will work adequately if an efficient means of moving data between cells is provided. Thus, we conclude that the resources required to construct large cell memories would be far better spent in constructing more cells with less memory.

Need Fast Some/None and Find First

A common means of controlling loop execution in CAM algorithms is to continue processing until none or only one of the CAM's tag bits are turned on. It is thus essential that we have a fast means of determining this. The simplest way of doing this is to test whether any tags are on; if so, then we select one and turn it off, then repeat the some/none test. The find first operation is also used frequently when a search finds several data elements with the same key value. It then provides a way to select one of these for processing. These cases combine to emphasize the need for fast some/none and find first operations.

Slower Response Count is Acceptable

Many CAM designs devote much complex and expensive hardware to increasing the speed of the operation which counts the tag bits that are turned on. We have found, however, that the count of responding tags is used primarily as a way of gathering statistics for use at much higher levels of processing control to direct the strategic application of the CAM. It is thus rather infrequently applied as compared to operations such as comparisons and some/none tests. We thus feel that slower, simpler, less expensive response count hardware is quite acceptable. Further, we have designed a very simple response count system which runs only about an order of magnitude slower than much more complex designs.

Convenient Additions

CAMs typically have only one tag bit per cell. We have found, however, that most algorithms need two to four tags. Usually this is simulated by storing tag bits in the memory, however this becomes a major inconvenience when the cells have small memories. It is thus convenient to have multiple tag bits in each cell. Although any CAM with bit select and multi-write can perform bit-serial addition (and, incidentally, is thus called a Content Addressable Parallel Processor -- CAPP -- see Foster [1]), it is far more convenient and several times faster to perform additions if each cell contains a full adder. Finally, we have also found it quite convenient if each cell is provided with a way of logically combining stored tag bits. When a CAPP is provided with these capabilities at the individual cell level, the result is a Single Instruction Multiple Data (SIMD) parallel processor of considerable power.

An Image Processing CAPP

By the Winter of 1981 we had begun to examine application of a CAPP to image processing. We soon found that we were dealing with two kinds of problem solutions. One kind worked independently of where pixel values were placed in the CAPP. An example of this truly associative type of solution is histogram directed feature extraction. The other kind required that pixel data be combined and it was thus necessary to use inter-cell communication links to accomplish this. Although we had already considered a linear cell interconnect (as a way to simulate cells with larger memories), we were now faced with problems that required a rectangular interconnect (hexagonal and triangular interconnects were not considered because digitized images do not map well onto them). An example of this is contrast enhancing image convolution. We also discovered that the edges of an image require special processing. Our solution to this problem was to provide a four-way (N, S, E, W,) cell inerconnect network with three different edge treatments. The simplest edge treatment is dead-edging, that is making the edges of the grid act like a frame of inactive cells. Another treatment is circular-wrap. In this case each edge cell is logically connected back around to its counterpart on the opposite edge. The most complex treatment is zig-zag wrap in which each edge cell is logically connected to a cell on the opposite edge that is offset by one row or column. This last treatment provides a way to turn an essentially rectangular CAPP into a linear structure and thus make it more general.

Some practical aspects of designing an image processing CAPP include the need to be able to load the memory with an image in one video frametime (1/30 second). This may seem like a long time, but remember that a 512x512 image contains 262,144 pixels. For sixteen-bit pixels this means a data transfer rate of about sixteen million bytes per second. We have also considered types of secondary storage that will be needed to keep up with such transfer rates. Hardware testing and debugging have also been major concerns simply because of the large number of components involved.

Titanic

In the Summer of 1981 we started work on the design of a VLSI-based CAPP for image processing. Our intent was to produce a conservative design which would be simple enough for us to construct with reasonable confidence of success but which would also provide a significant advance in processing power. From the beginning we imposed a number of constraints on the design. For example, the CAPP would have to consist of no more than one hundred circuit boards and each board should have a maximum of one hundred off-board connections. As another example, the VLSI chips we designed would be restricted to no more that 40,000 devices, have a pin-out of no more than forty pins, and dissipate less that two watts.

We also set a number of goals which we hoped to achieve. It was decided that the CAPP should contain 262,144 cells arranged as a rectangular 512x512 array to facilitate image processing. Each cell would contain at least thirty-two bits of memory, multiple tags, and some bit serial processing power. One hundred nanoseconds was set as a goal for the minor clock cycle time. We also planned to meet as many of the design recommendations established by our CAM research as we could. Finally, it was decided that the CAPP would be built to be driven by another machine, such as a Digital Equipment Corporation VAX. Once the goals and constraints were set, work on the design got under way and, for obscure reasons, the project was given the name "Titanic".

Titanic and Its Environment

The Titanic is divided into two main parts: the central control and the parallel processor. The central control is a simple, fast, fetch-ahead pipelined processor which will be built from MSI devices. It issues instructions to the parallel processor, controls loading and unloading of data in the parallel processor, serves as an interface to the VAX or other host computer and to other data sources and secondary storage devices. The central controller contains a ROM with a set of micro-coded subroutines for performing commonly needed higher level CAPP operations, and a writeable control store which allows users to add their own special microcoded instructions. Also contained in the central controller is a small program memory into which subroutines or entire programs may be loaded. The writeable control store and program memory are loaded directly by the VAX. Once these memories are loaded, the VAX can issue commands to the central controller which tell it to execute routines stored in the program memory, to single step through a stored routine, or to execute a single instruction passed as a literal by the VAX. Figure 1 shows Titanic and its environment.

The Parallel Processor

The Titanic parallel processor consists of an 8x8 array of processing circuit boards and a set of special purpose boards which control how the edges of the CAPP are treated, buffer broadcast signals, and perform other functions such as collecting the some/none signals to a single line. The parallel processor receives data and instructions broadcast to it by the central controller. Each parallel processor instruction operates in exactly one minor cycle time. Some operations do require multiple clock cycles, but these are taken care of by having the central control rebroadcast the instruction as many times as necessary. Figure 2 shows the structure of the parallel processor.

Each processor board consists of an 8x8 array of special CAPP integrated circuits plus some random buffer logic. A list of the sixty-three I/O lines on each board is given in Table 1. Our current design calls for all sixty-four processor circuit boards to be placed in four card racks (sixteen per rack) and interconnected by a broadcast backplane and ribbon cables.

The Titanic IC

The heart of the Titanic design is a special purpose nMOS VLSI CAPP integrated circuit. Each of these chips contains sixty-four CAPP cells, an instruction decoder, and other miscellaneous logic. The design of this IC is actually much further along than the rest of the project (this being mainly due to test chip fabrication time constraints). To compensate for this somewhat bottom-up development we have designed the chip with as much generality as possible, knowing that such generality need not be fully used later on.

The Communications Interconnect

One of our biggest problems in designing Titanic was how to handle the rectangular interconnection of the cells. The number of wires required for such a network, even for bit serial communications, is staggering. This became most evident when we tried to design the IC communications interface. For sixty-four cells, the arrangement which gives the minimum number of external connections is an 8x8 grid. With a four-way N,S,E,W interconnect there are then only thirty-two neighboring cells to connect to. (We considered an eight-way N,S,E,W,NW,NE,SW,SE interconnect, but were forced to abandon it due to the wiring complexity.) By the time control, power, and clock signals were added to the thirty-two neighbor lines, we found that a sixty-four pin package would be required to hold the IC. Further examination also revealed that a full interconnect would require that each processor board have 256 ribbon cable communication lines -- in other words, a two foot wide swath of ribbon cable running between each pair of boards! Because this violated two of our main design constraints, we had to simplify the interconnect.

By 8:1 multiplexing the communications net as it crossed chip boundaries, we were able to reduce the IC pin count to twenty-two pins and the total board wire count to sixty-three (of which only thirty-two need to be run in ribbon cable). By going from sixty-four pin to twenty-two pin packages, the board size was also reduced significantly. Unfortunately, all of these benefits were paid for in a loss of speed. The new interconnect takes 0.8 microseconds to transfer one bit between cells (25.6 microseconds for thirty-two bits). We should also note here that the Titanic instruction set makes this multiplexing transparent to the user.

Some/None Logic

On-chip the Some/None signal is determined by feeding the output of the main tag bit into a sixty-four-way NOR with an inverter between its output and the Some/None pad driver. Once the signal goes off-chip, it passes through a four-level OR tree before reaching the central controller.

Count Responders

The count responders operation requires only three changes to be made to the CAPP circuitry to be feasible. Firstly, it must be possible to connect all of the response bits into a circular shift register. This is easily accomplished because the neighbor communication network already provides most of the necessary links. Secondly, a register, a counter, and a full adder must be added to each chip. Finally, the cards that control the top-bottom edge treatment must be modified to include column summing registers and a final sum register.

The algorithm used to count responders is given in Figure 3. This method is reasonably fast (about twenty-six microseconds), inexpensive, and most importantly it can be used with any size of array without having to modify the IC -- only the bottom row circuit board needs to be changed.

Device Floorplan

Figure 4 shows the Titanic IC's floorplan. The unit cells are arranged in two columns of thirty-two. This arrangement was chosen because we found that the best compaction would be obtained if we could share control and memory select lines among as many cells as possible. Each cell is thus very long and narrow. A column of thirty-two cells is almost covered by a river of metal control and select lines which run vertically over it. These lines are simply duplicated and mirrored for the two columns. Control is generated by a NOR-NOR network forming the instruction and address decoders. Responder count hardware is provided in a small block of random logic. The overall size estimate of the active chip area (excluding pads and drivers) is 2400x2400 lambda. Thus if lambda is three microns, the central portion of the die would be roughly 285 mils on a side. This is somewhat large, but not unreasonable. Power dissipation is estimated at 1.5 watts, which is low enough to allow forced-air cooling. Table 2 lists the pin functions of the Titanic IC.

The Unit Cells

A unit cell consists of thirty-two bits of fully static memory, four one-bit static tag "registers" called A, B, X, and Y, and a static carry bit "register" called Z. Each cell also contains an ALU which continuously generates X nand Y, X nor Y, and X + Y + Z. Finally, each cell contains logic for selecting some source of data (a register, memory, an ALU function, broadcast data, or a neighbor cell), possibly inverting the selected signal and storing it in a destination (memory or register). Neighbor communiation lines run vertically in two channels in the middle of the cell. The Z register is special in that it is not available for selection as a data source. It can be copied directly to the X register and can be loaded from the output of the selector. It also is loaded with the carry from X + Y + Z whenever that function is selected.

The X register is special in that its output is connected to the some/none logic and the neighbor communication network. In some sense it is the "main" tag bit.

The A register is also special. It controls whether the cell is active. If a cell is not active, it ignores all instructions broadcast by the central controller except a special few.

The Y register is intended to be used for storing a second set of tag bits which may eventually be combined with other sets through the logical operations provided by the ALU.

The B register is intended as temporary storage for a second set of activity bits, essentially providing a single level of "subroutine call" or an alternative activity "screen".

Figure 5 shows the logical arrangement of a unit cell while Figure 6 shows its silicon floorplan.

Titanic IC Instruction Set

Table 3 lists the instruction set of the Titanic IC. Each instruction executes in one minor clock cycle (100 ns). This was done to avoid feedback loops in the decoder on the chip and to avoid special instruction states in the central controller. This means that the central controller must be programmed to re-issue some instructions several times. For example, transferring data to neighbor cells across chip boundaries requires eight individual transfers because of the 8:1 multiplex. The central control must therefore issue the shift instruction eight times in a row. This, of course, will be encoded as a single operation in the controller's microcode ROM.

There are eight basic instructions recognized by the chip. Of these, six are memory transfer operations and use a five-bit address value to select the bit to be read or written. The other two instructions treat the address as a sub-operation specifier. For the most part these are non-memory data source to register data transfer operations with one op code causing the data to be inverted before storage and the other causing a direct transfer. There are nineteen special sub-ops, however, which are reserved for unusual operations such as transferring data on and off the chip or counting responders.

Some operations (those followed by exclamation points (!) in Table 3) are also designated as "jam transfers". This means that they are performed regardless of whether the A register contains a logic one. These provide a means of storing and retrieving different activity patterns and of applying global operations which ignore activity without the usual overhead of having to save the current activity pattern, and retrieve it later.

Current Status

As of this writing we have designed a sixteen-cell (4x4) test chip, and are negotiating for fabrication. Using a simple set of three micron design rules, we have succeeded in fitting the circuitry onto a 180x180 mil body area with room to spare. The actual cell area occupies only 130x106 mils. Estimated power dissipation is only 350 milliwatts. The design includes about 7000 transistors.

We have already written a number of programs for the Titanic and estimated their operation times by hand. For example, one special purpose convolution of interest in computer vision processing (a simple 3x3 mask) required only

97.8 microseconds for the entire 512x512 image. More complex convolutions take longer, of course, but most of interest can be performed in less than five milliseconds. We have also examined motion analysis and found the results to be quite encouraging.

Further Research

Based on the results of our test chip experience, we intend to proceed to full sixty-four cell ICs and, eventually, construction of the entire machine. Architectural changes which we intend to pursue are increasing the memory size to sixty-four bits per cell and perhaps going to an 8:2 communications multiplex (with a twenty-eight pin package) for a doubling in the data transfer rate.

We also plan to program a statistics gathering Titanic simulator which will allow us to experiment with software development and optimization.

Our work thus far has indicated that a Content Addressable Parallel Array Processor is extremely well suited for image processing, vision, and motion analysis. We intend to pursue further applications in these areas and also in new areas such as tactile object recognition in robotics.

Conclusion

Rationale and a design have been presented for a Content Addressable Parallel Processor suitable for both general use and image processing applications. The architecture of the processor is based in practical experience and the hardware design has been constrained to make it possible to construct using existing technology and with a high confidence of success. Despite these constraints, simulations have shown that such a machine would provide a significant increase in processing power over what is presently available.

References

[1] Foster, Caxton C., Content Addressable Parallel Processors, Van Nostrand Reinhold, New York, 1976.

List of Processor Board I/O Lines

Number of Lines	Function
8	Bidirectional North Neighbor Communications
8	Bidirectional South Neighbor Communications
8	Bidirectional East Neighbor Communications
8	Bidirectional West Neighbor Communications
8	Chip Column Select
8	Chip Row Select
4	Op Code
5	Bit Address or Sub Op Code
1	Broadcast Comparand Data
1	Some/None Output
2	Clock phases
1	Power
1	Ground
63	

Table 1

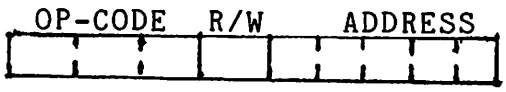
List of Titanic IC Pin Assignments

Pin	Function
1	West Neighbor Communications (Bidirectional)
2	Chip Select 1
3	South Neighbor Communications (Bidirectional)
4	Op Code Bit 1
5	Op Code Bit 2
6	Op Code Bit 3
7	Op Code Bit 4
8	Comparand in
9	Some/None out
10	Clock Phase 1
11	Ground
12	East Neighbor Communications (Bidirectional)
13	Chip Select 2
14	Spare (Test)
15	Address Bit 5
16	Address Bit 4
17	Address Bit 3
18	Address Bit 2
19	Address Bit 1
20	North Neighbor Communications (Bidirectional)
21	Clock Phase 2
22	Power

Table 2

<u>Titanic IC Instruction Set</u>

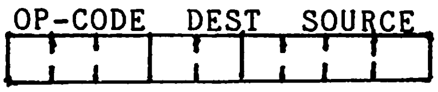
Memory Operations



Op	R/W	FUNCTION
0	0	M:=C
0	1	A:=M
1	0	M:=B
1	1	B:=M
2	0	M:=X
2	1	X:=M
3	0	M:=Y
3	1	Y:=M
4	0	M:=A!
4	1	A:=M!
5	0	M:=B!
5	1	B:=M!

!	-	Transfer ignores activity
A	-	Activity register
B	-	Secondary Activity register
C	-	Comparand
M	-	Memory
X	-	Main tag register
Y	-	Secondary tag register
Z	-	Carry register
N	-	Data from North
E	-	Data from East
W	-	Data from West
S	-	Data from South

Register Operations



OP CODE 6 - NORMAL
OP CODE 7 - INVERT SOURCE

	Destination			
Source	0	1	2	3
0	X:=A!	A:=B	B:=A!	Y:=A!
1	X:=B	A:=B	B:=B	Y:=B
2	X:=X	A:=X	B:=X	Y:=X
3	X:=Y	A:=Y	B:=Y	Y:=Y
4	X:=X+Y	A:=$\overline{X+Y}$	B:=$\overline{X+Y}$	Y:=$\overline{X+Y}$
5	X:=X^Y	A:=$\overline{X\text{^}Y}$	B:=$\overline{X\text{^}Y}$	Y:=$\overline{X\text{^}Y}$
6	X:=XvY	A:=$\overline{X\text{v}Y}$	B:=$\overline{X\text{v}Y}$	Y:=$\overline{X\text{v}Y}$
7	X:=C	A:=C	B:=C	Y:=C
8	X:=N	A:=N	B:=N	Y:=N
9	X:=E	A:=E	B:=E	Y:=E
10	X:=W	A:=W	B:=W	Y:=W
11	X:=S	A:=S	B:=S	Y:=S
12	X:=N˜!	A:=C!	X:=CN˜!	Z:=C
13	X:=E˜!	A:=B!	X:=CE˜!	(6)Z:=X (7)X:=Z
14	X:=W˜!	A:=X!	X:=CW˜!	(6)SCRR! (7)CRCR!
15	X:=S˜!	A:=Y!	X:=CS˜!	(6)SCRC! (7)PANS!

˜	-	Zig zag shift with data transfer in and out of chip
SCRR	-	Shift and count responders by rows
CRCR	-	Clear response count register
PANS	-	Pipelined add North to South
SCRC	-	Shift and count responders by columns

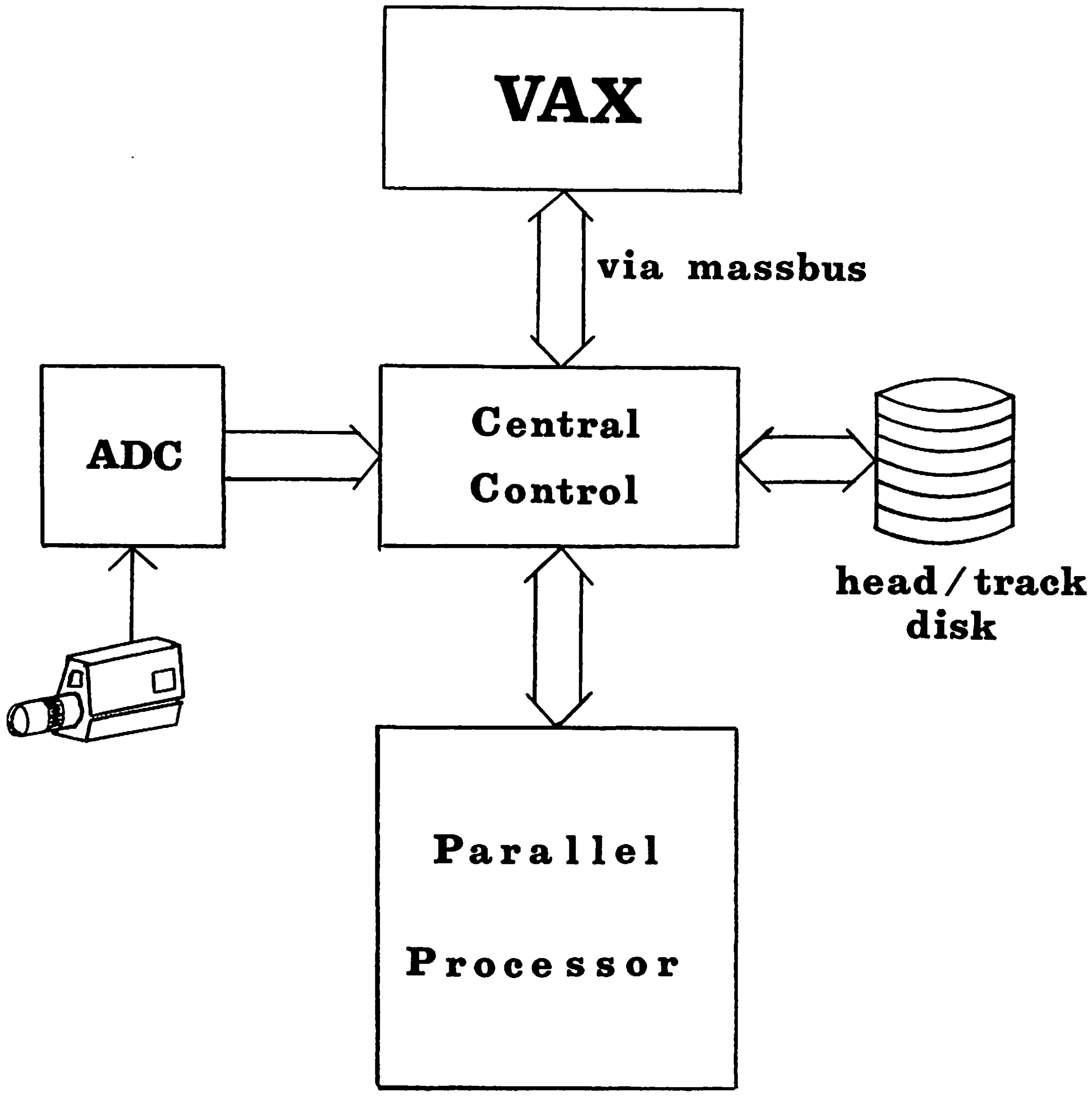

VAX
via massbus
ADC
Central Control
head / track disk
Parallel Processor

Fig. 1

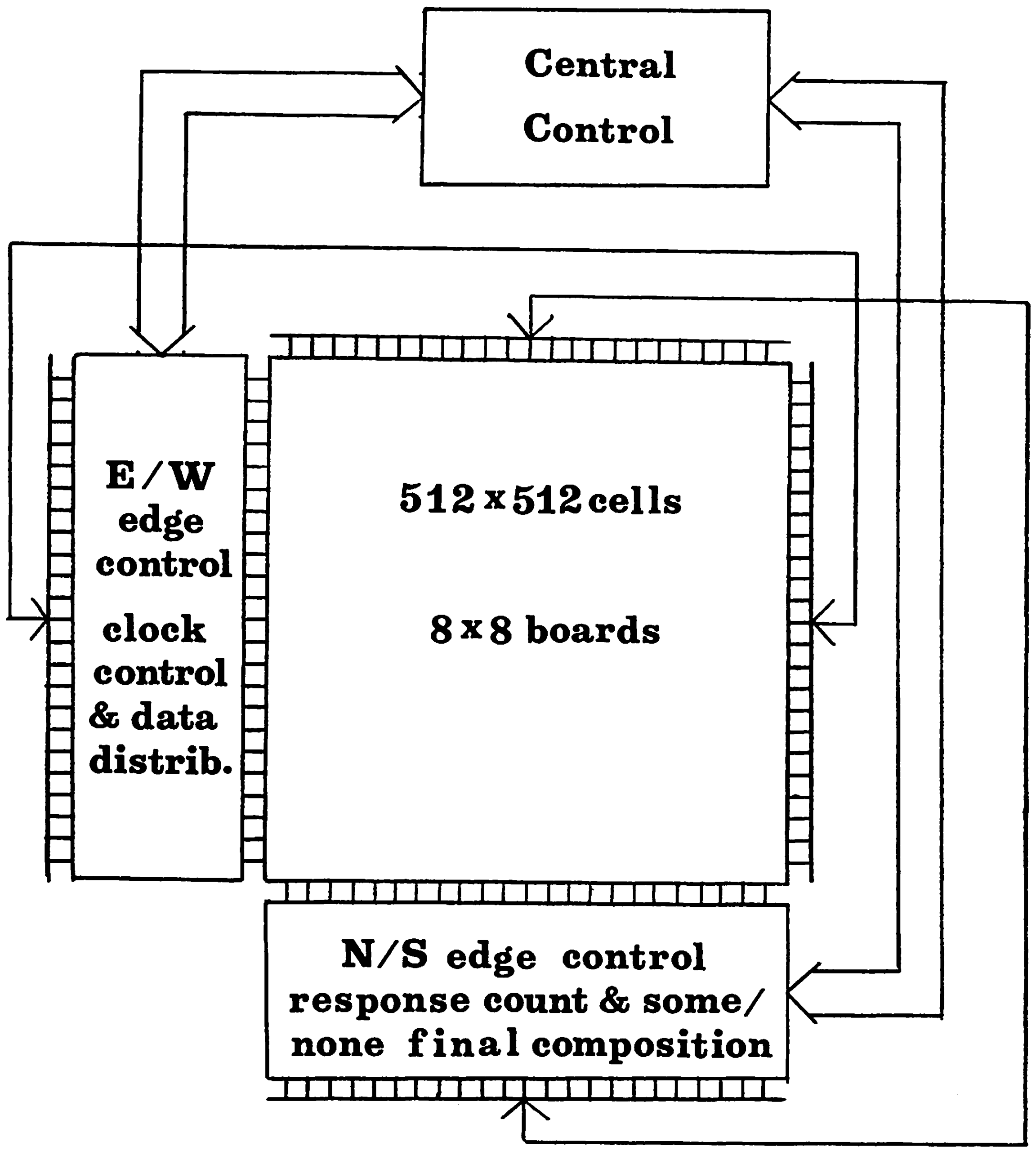

Central
Control
E/W
edge
control
clock
control
& data
distrib.
512 x 512 cells
8 x 8 boards
N/S edge control
response count & some/
none final composition

Fig. 2

```
Set all activity bits                                       0.1µS
Clear Response Count Register (CRCR)                        0.1µS
For I:=1 to 64 do
    Shift and Count Responder (SCR)                         6.4µS
Turn off all chip row select lines                          0.1µS
Turn on all chip column select lines                        0.1µS
For I:=1 to 64 do
    begin
        Turn on row select line I                          12.8µS
        Pipeline Add North to South (PANS)
    end
For I:= to 6 do (*Empty the pipeline*)
    Pipeline Add North to South (PANS)                      0.6µS
For I:=1 to 64 do
    Pipeline Add West to East on Bottom Row Board           6.4µS
                                                          ________
Response count is now available on Bottom Row
    Board                                                  26.6µS
```

Pipeline Add North to South (PANS) takes the low order bit from the response count register, adds it to a data bit input on the North line and outputs the result on the South line. The carry from the addition is stored in a temporary storage cell and used in the next PANS. The input and output operations are buffered and appropriately clocked to allow true pipelined operation. Row and column select lines are turned on and off by setting and clearing bits in registers on the edge control cards. Once a row is turned on, it remains on until it is explicitly turned off and vice versa.

Fig. 3

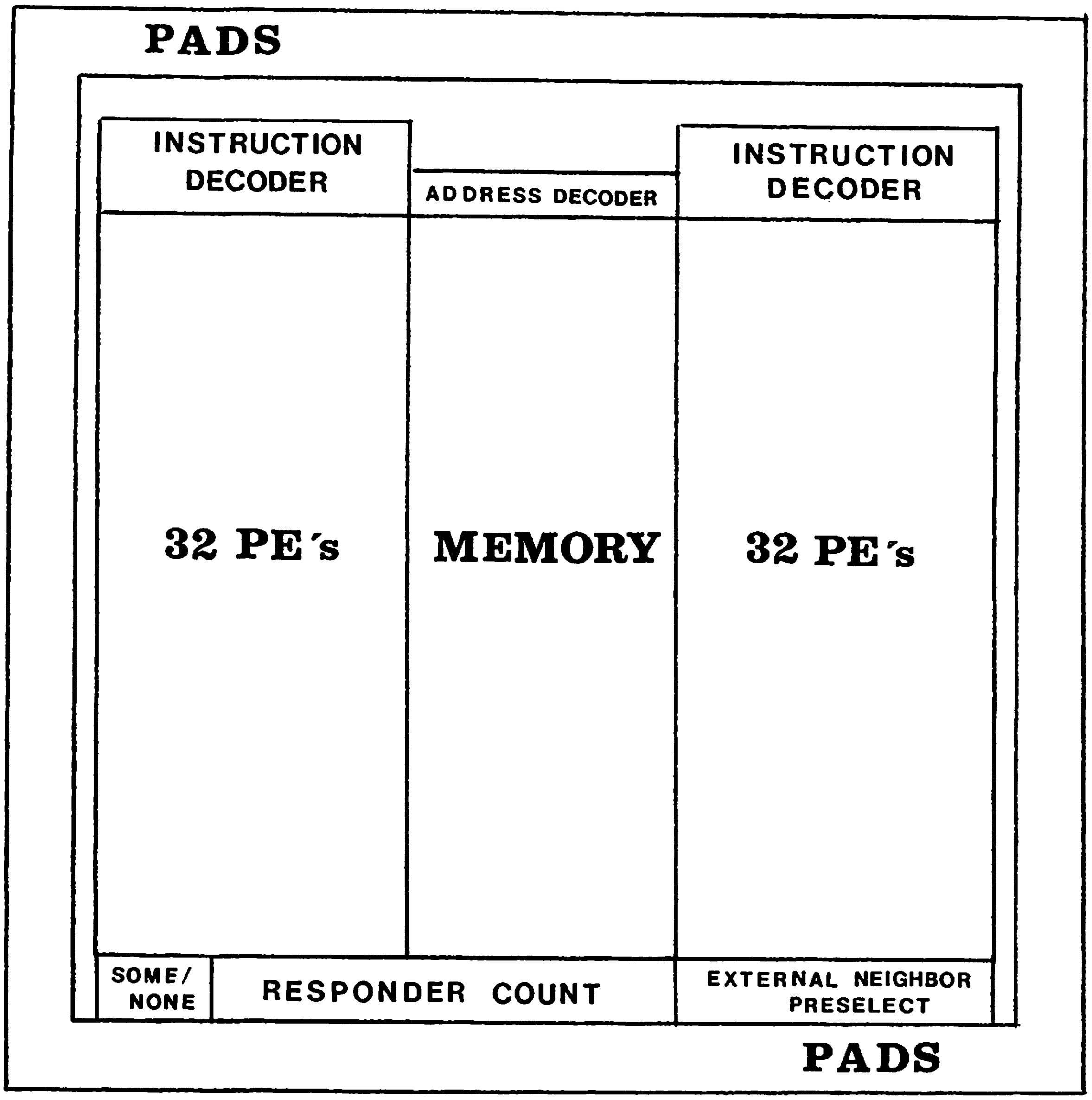
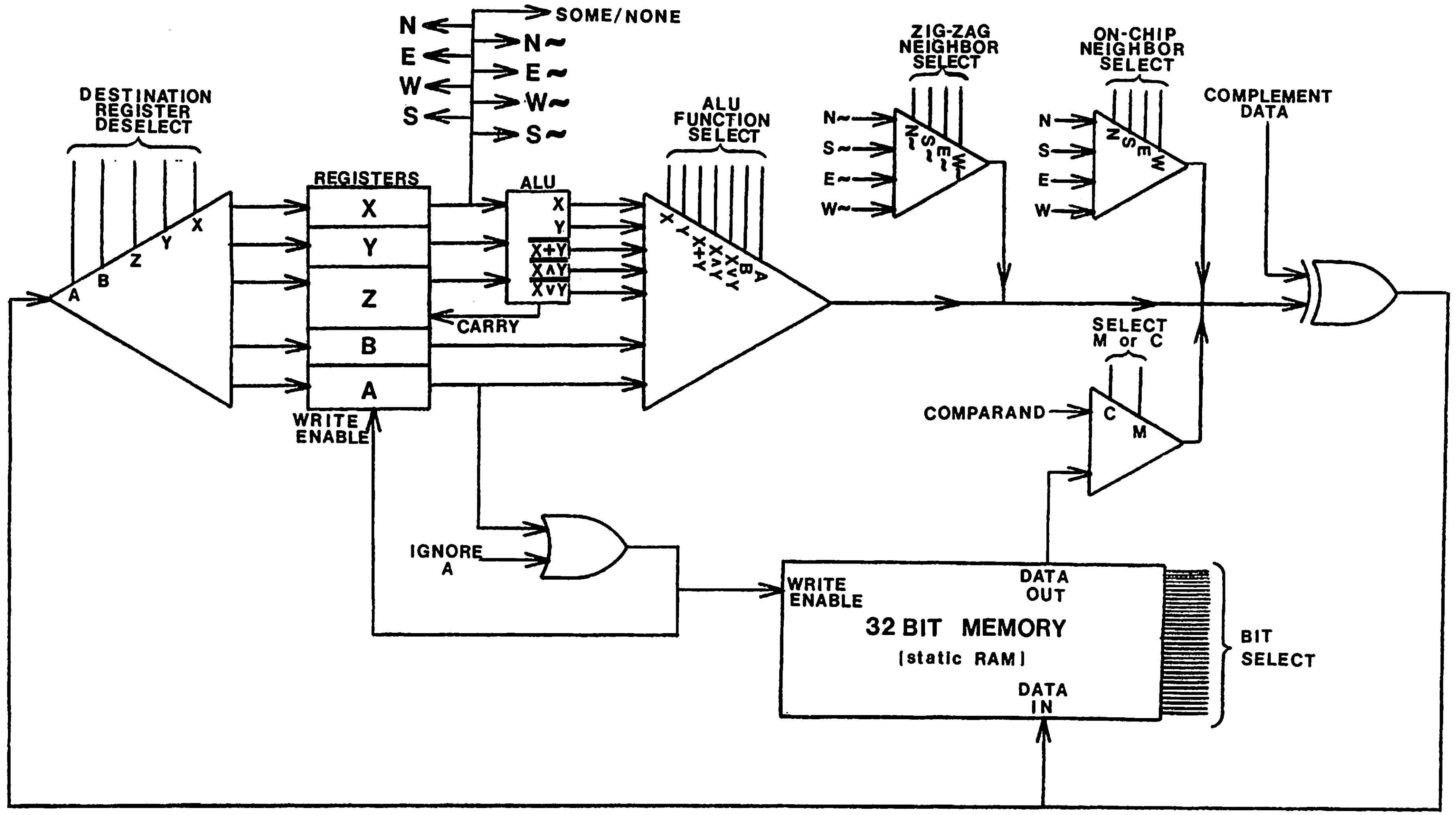
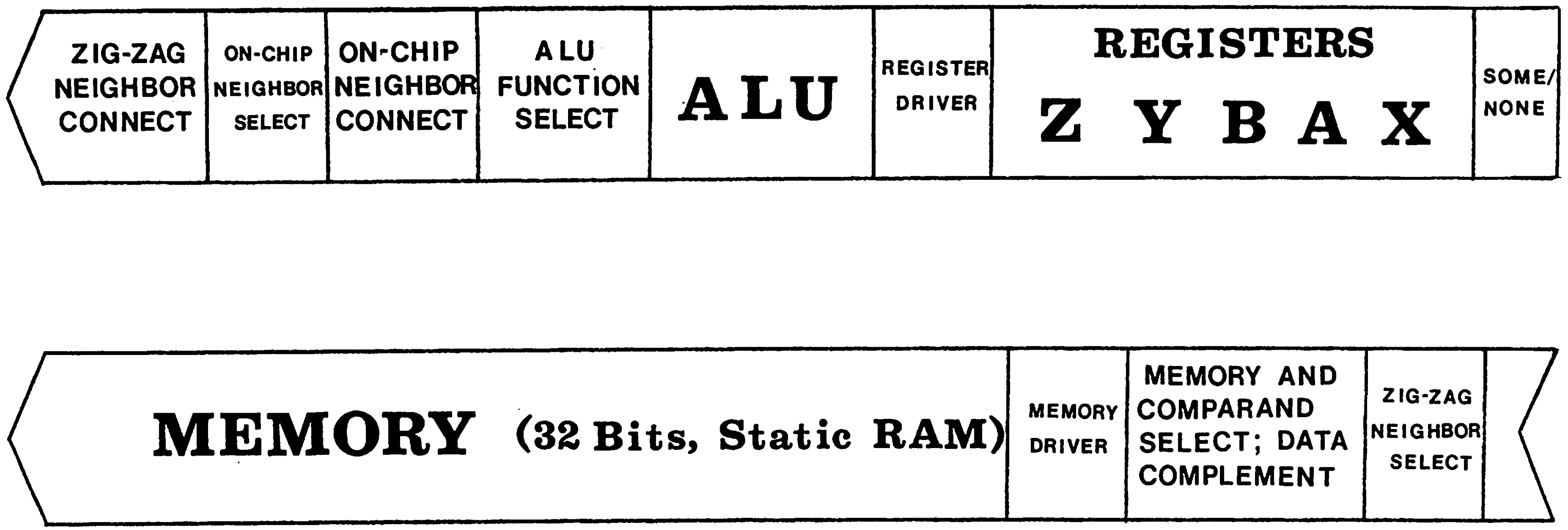

PADS
INSTRUCTION DECODER
ADDRESS DECODER
INSTRUCTION DECODER
32 PE's
MEMORY
32 PE's
SOME/ NONE
RESPONDER COUNT
EXTERNAL NEIGHBOR PRESELECT
PADS

Fig. 4



Organization of One PE

Fig. 5



Floorplan of One PE

Fig. 6

An Algorithm for a Simple Image Convolution on the Titanic Content Addressable Parallel Array Processor

Charles Weems

June 1982

Abstract

An algorithm is presented for the Titanic Content addressable Parallel Array Processor [1] which will cause it to perform a simple image convolution very quickly. It is further shown that this algorithm can be generalized to perform more complex convolutions with only a moderate reduction in speed.

Background

Our previous work on Conway's Game of Life implemented on a CAM [2] demonstrated that such a device could be effectively used to speed up algorithms which dealt with rectangular grids of cells and small neighborhoods about each of those cells. Because Conway's Game of Life actually involves performing a very simple image convolution, it was soon realized that the technique developed for Life could be applied to more general convolutions. This method was further refined with the Titanic design -- a content addressable parallel array processor.

· This paper is available separately as COINS Technical Report #83-07.

Basic Technique

One simple form of convolution involves each cell on a rectangular grid examining its immediate neighborhood and then updating its own contents based upon some function of that neighborhood. The update must, of course, be performed after all cells have finished examining their neighborhoods. On a parallel array processor this examination can be performed simultaneously by all of the cells on the grid, as can the update operation. Thus the algorithm for the convolution can be described as the actions of a single cell with the understanding that each action is performed simultaneously by all of the cells.

There are two different ways of approaching the problem of examining the neighborhood. The one that first comes to mind is that each cell "looks" at each cell in its neighborhood, gathering what information it needs to perform an update. In practice this involves moving data from each cell in the neighborhood into the "central" cell where some function is then applied to it and the result stored for the update phase of the convolution. The problem with this is that the data must often pass through other cells before it reaches the central cell. For example, when the neighborhood is 7x7 cells, data from the outer ring of cells must pass through at least two other cells before reaching the center cell. Because movement of data takes time, this "passing through" is rather inefficient. The solution is to have the data stored in the intermediate cells on its way to the center, thus taking advantage of the fact that those cells will also need to know the values in order to compute the function of their neighborhoods. Although this will work, the algorithm becomes rather messy since we must now consider the actions of several cells at once and how these relate to each other. It also becomes a complex problem to determine an optimal set of data collection paths as the neighborhood's diameter varies.

It turns out that the other approach to examining the neighborhood greatly simplifies these problems. This approach takes the opposite view of the collection process. Instead of each cell collecting all of the data from its neighborhood, each cell distributes its own data to every cell in the neighborhood. Because every other cell is also doing this, the end result is that the central cell (and hence all cells) gets the data it needs from all of the cells in the neighborhood. The problem of establishing an optimal distribution path is trivial for a square array of odd diameter: It is simply a rectangular spiral out from the center cell. For even diameter square neighborhoods the problem is only slightly more difficult because the center cell is actually half of a cell width off center in two diections. In this case it is simply required that the appropriate choice of initial direction and of clockwise or counter clockwise spiral be made to select the optimal path. The only other point that requires mentioning is that, because this is a distribution process rather than a collection process, the funtion mask for the convolution must be mirrored across the central cell. For example, when the cell's value is being stored in its north neighbor, the function applied to that value is the south neighbor function. The reason for this can be seen when it is realized that the central cell is actually the south neighbor of the cell to its north. The mirroring of the convolution function mask is actually quite easy to accomplish: we simply step through the mask in exactly the opposite direction that the distribution path takes.

Let's look at an example: A simple convolution for smoothing isolated cells of noise out of an image. We will use a 3x3 convolution mask in which the cell accumulates the sum of its neighbor's values, weighted inversely with distance away from the center. The sum will then be normalized. Define the mask to be an array $M^{i\ j}$:

$$M = \begin{array}{c|ccc} & j\ \ 0 & 1 & 2 \\ \hline i & & & \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 4 & 2 \\ 2 & 1 & 2 & 1 \end{array}$$

Where $M_{1,1}$ is the central cell. Then the following algorithm will perform the convolution (north is up, west is to the left, etc.):

```
i := 1
j := 1
sum := value *$M_{i\ j}$
move value north
i := i+1
sum := sum + value * $M_{ij}$
move value east
j := j+1
```

```
sum := sum + value * $M_{ij}$
move value south
i := i-1
sum := sum + value * $M_{ij}$
move value south
i := i-1
sum := sum + value * $M_{ij}$
move value west
j :+ j-1
sum := sum + value * $M_{ij}$
move value west
j := j-1
sum := sum - value * $M_{ij}$
move value north
i := i+1
sum := sum + value * $M_{ij}$
value := sum * normalizing factor
```

It should be noted that the time required to perform a convolution using the parallel processor is independent of the size of the image and only dependent upon the area of the convolution mask. Since the Titanic does cell level arithmetic bit-serially, the size of the data values also affects the speed of the algorithm.

Convolution on Titanic

The following algorithm gives the list of instructions required to make Titanic perform the convolution given in the above example. In this case we have taken advantage of special characteristics in the mask values to help direct the shift and add process of the required multiply operations. The algorithm is written for 8 bit data values and runs in an estimated time of 98 microseconds.

```
(* Initialize *)

A := 1!
Empty_Edges

(* Send to North *)
```

```
Z := 0
For Bit := 2 to 9 do
   X := M(Bit)
   Shift_X_North
   M(Bit+10) := X
   Y := M(Bit -1)
   Y := X+Y
   M(Bit - 1) := Y
End For
X := 0
For Bit := 9 to 11 do
   Y := M(Bit)
   Y := X+Y
   M(Bit) := Y
End For

(* Send to Northwest *)

Z := 0
For Bit := 12 to 19 do
   X := M(Bit)
   Shift_X_West
   M(Bit) := X
   Y := M(Bit - 12)
   Y := X+Y
   M(Bit - 12) := Y
End For
X := 0
For Bit := 8 to 11 do
   Y := M(Bit)
   Y := X+Y
   M(Bit) := Y
End For

(* Send to West *)

Z := 0
For Bit := 12 to 19 do
   X := M(Bit)
   Shift_X_South
   M(Bit) := X
   Y := M(Bit - 11)
   Y := X+Y
   M(Bit - 11) := Y
End For
X := 0
For Bit := 9 to 11 do
   Y := M(Bit)
   Y := X+Y
   M(Bit) := Y
End For

(* Send to Southwest *)

Z := 0
```

```
For Bit := 12 to 19 do
   X := M(Bit)
   Shift_X_South
   M(Bit) := X
   Y := M(Bit - 12)
   Y := X+Y
   M(Bit - 12) := Y
End For
X := 0
For Bit := 8 to 11 do
   Y := M(Bit)
   Y := X+Y
   M(Bit) := Y
End For

(* Send to South *)

Z := 0
For Bit := 12 to 19 do
   X := M(Bit)
   Shift_X_East
   M(Bit) := X
   Y := M(Bit - 11)
   Y := X+Y
   M(Bit - 11) := Y
End For
X := 0
For Bit := 9 to 11 do
   Y := M(Bit)
   Y := X+Y
   M(Bit) := Y
End For

(* Send to Southeast *)

Z := 0
For Bit := 12 to 19 do
   X := M(Bit)
   Shift_X_East
   M(Bit) := X
   Y := M(Bit - 12)
   Y := X+Y
   M(Bit - 12) := Y
End For
X := 0
For Bit := 8 to 11 do
   Y := M(Bit)
   Y := X+Y
   M(Bit) := Y
End For

(* Send to East *)

Z := 0
For Bit := 12 to 19 do
```

```
   X := M(Bit)
   Shift_X_North
   M(Bit) := X
   Y := M(Bit - 11)
   Y := X+Y
   M(Bit - 11) := Y
End For
X := 0
For Bit := 9 to 11 do
   Y := M(Bit)
   Y := X+Y
   M(Bit) := Y
End For

(* Send to Northeast *)

Z := 0
For Bit := 12 to 19 do
   X := M(Bit)
   Shift_X_North
   M(Bit) := X
   Y := M(Bit - 12)
   Y := X+Y
   M(Bit - 12) := Y
End For
X := 0
For Bit := 8 to 11 do
   Y := M(Bit)
   Y := X+Y
   M(Bit) := Y
End For

(* Scale Result *)

For Bit := 2 to 11 do
   X := M(Bit)
   M(Bit - 2) := X
End For
M(10) := 0
M(11) := 0


980 CAM Operations
98 uS per Convolution

340 Conv / Frame Time
10204 Conv / Sec
```

Convolutions with more general and/or larger masks will take longer. A very rough worst case estimate of the time required for such convolutions can be obtained from the formula:

$$T = P(.8N+.2M+.1) + .3M(N^2P+N+1)$$

where T = time in microseconds
N = number of bits in a pixel
M = number of bits in a mask value
P = number of pixels in the mask area

This is a worst case time which assumes that all of the bits in all of the mask values are ones (since this gives the slowest multiply speed). Under normal circumstances, T will be about half of the value obtained from the formula. This also assumes a totally general square mask where the values can change. If constants are to be used for the mask values, a significant speed increase can be obtained by optimizing the multiples for those values. Thus, for example, a convolution on 16 bit values with 8 bit mask values could be applied over at most a 7x7 mask in one video frame time with a worst case situation. For normal situations, it should be possible to convolve a 10x10 area. Given constant mask values, and depending upon the amount of optimization possible, even a 25x25 mask could be done in one video frame time.

As a final note, this method is not restricted to square masks and in fact should be readily generalizeable to any mask shape. All that is required for this is an algorithm for efficiently shifting the center cell's value so that it covers the mask area. Thus it should be possible to easily adapt it to such mask shapes as annuli and disjoint areas.

Conclusion

A method has been shown which can be used to program the Titanic content addressable parallel array processor to perform image convolutions simply and efficiently. Such a program, for a simple convolution, was shown which operates in ninety-eight microseconds. The time of the algorithm is independent of the size of the image and depends only upon the size of the mask and, for bit serial processing, upon the number of bits in the pixel and mask values. A formula was given for a worst case time estimate and a factor for estimating normal case time from this was discussed. It was also noted that the method could be applied to masks of other than square shapes.

References

1. C. Weems, S. Levitan, and C. Foster, "Titanic: A VLSI-Based Content Addressable Parallel Array Processor, Proceedings of IEEE International Conference on Circuits and Computers, September 1982, pp.236-239.

2. C. Weems, "Life is a CAM-Array Old Chum", unpublished paper, January 1980.

Finding a Center of Mass with a CAM*

by

Caxton C. Foster
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

ABSTRACT

Given a set of active points scattered about a plane, it is often of interest to discover the center of mass of these points. A method is presented for discovering the center of mass using a Content Addressable Memory.

Introduction

Suppose a new disease has been reported from a number of towns across the country. Suppose an area of a picture has been identified as belonging to a particular object. Suppose the intersections of many pairs of vectors have been found. In each of these cases it might be of interest to find the centroid of the points of interest. If the points vary in mass or in reliability, we might want to give the heavier points more weight in our averaging algorithm.

Let the set of points of interest, P_i, have mass m_i and x-y coordinates x_i, y_i. What we would like to discover is the first moment of the distributions:

$$\overline{X} = \frac{\sum_{i=1}^{n} x_i m_i}{\sum_{i=1}^{n} m_i}$$

* This work supported in part by a grant from the Army Research Office, [illegible]-71-G-0046

$$\bar{Y} = \frac{\sum_{i=1}^{n} y_i m_i}{\sum_{i=1}^{n} m_i}$$

If there are n points of interest then, in general, it will require a time of order n to discover the center of mass in a conventional coordinate addressed computer. Content Addressable Memories offer a method of finding the center of mass in a time independent of the number of points involved.

TITANIC

The machine we have called TITANIC has been described elsewhere in detail. For our purposes here it is sufficient to note that it is a content addressable memory with its cells arranged on a square grid. Each cell thus has four spatial neighbors with which it can communicate as well as a communication path to and from the central control unit.

A cell which matches some search criterion is called a "responder". This machine can do exact match searches, find the largest element or the smallest element of a set; it can locate the first responder or the leftmost or rightmost responder in each row or the topmost or bottommost of each column, and it can count number of responders.

Two algorithms in particular should be discussed because they are central to the operation.

Global Add.

Adding up a vector is relatively simple given the Count Responders instruction in TITANIC. Assume that in each cell of the memory the number we wish to add into the sum is stored in a field stretching from bit A (most significant) to bit B (least significant). We begin with the most significant bit and with the central variable SUM equal to zero. The algorithm proceeds as follows:

```
For I = A TO B
    Select those cells with bit I equal to 1
    X = count of the number of responders.
    SUM = 2*SUM + x
NEXT I.
```

As an example consider the following set of unsigned binary numbers:

```
                          1 0 1 1  =  11
                          0 0 0 1  =   1
                          0 1 0 0  =   4
                          1 1 0 0  =  12
                          0 1 1 1  =   7
number of ones
in column:                2 3 2 3
SUM = ((2*2+3)*2 + 2)*2 + 3 = 35
```

The Count Responders instruction takes about 20μseconds, so the sum of a vector of 8 bit numbers can be found in approximately 160μseconds independent of the number of elements in the vector.

For maximum speed global add will be micro-coded into the central controller of TITANIC.

FIND LOCATION

In a number of problems it is convenient if a cell can discover what row (or column) it is in. Storing this information permanently in each cell is wasteful of bits and serially writing the information in each of the 512 rows (or columns) is slow. Since we wish to make the chips (and circuit cards) of TITANIC interchangeable, we do not wish to put the location information on board a chip or wire it into each card. The following algorithm takes advantage of the way TITANIC is designed to tell each cell its row (or column) address very rapidly.

The chips which make up TITANIC have sixty-four words arranged in an 8x8 square. It takes .9 μseconds to select the topmost row of cells on every chip and .1 μseconds to select successive rows thereafter. Writing to many cells simultaneously (multiwrite) is possible in TITANIC. It is done bit serial-word parallel at a rate of ten bits per μsecond.

We set aside a nine bit field in each cell to hold its row address. We can select the first row of cells on every chip of the memory and write 000 in the low order three bits of their address fields. This will require .9 + .3 = 1.2 μseconds. We then select the next row of cells on every chip and write 001. This will require .1 + .3 = .4 μseconds. We proceed through all eight rows on the chips writing 111 in the last row for a total of 1.2 + .4x7 = 4.0 μseconds.

Sixty-four chips are arranged on each printed circuit card in eight rows and eight columns. Circuits on the cards allow the central control unit to select any row or rows of chips for participation in an operation. The row in which a chip lies on a card will determine the middle three bits of the "row address" of that chip's cells (see Figure 1). Selecting a row of chips on each card requires .1 μseconds, and writing three bits into the address fields of all the cells on the selected chips requires .3 μseconds. Thus, in 8 x .4 = 3.2 μseconds the central control unit can scan down all eight rows of chips and deposit the appropriate patterns for each row.

In a similar way, the sixty-four cards that constitute TITANIC's memory are arranged in eight rows and eight columns, and the central control unit

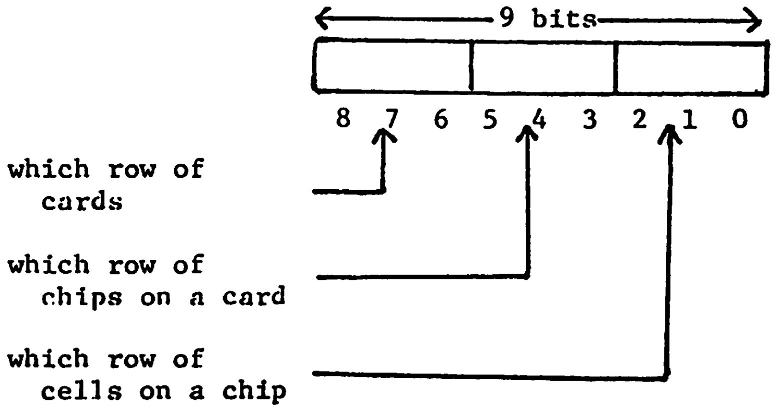


Figure 1. The constitution of a cell's "row address"

can select any desired combination of rows and columns of cards to participate in an operation. We select all the cells of all the chips on the first row of cards and write 000 into the high order three bits of the address fields of these cells. This takes .1 + .3 = .4 μseconds. We repeat on each successive row of cards and finish in .4 x 8 = 3.2 μseconds. When we finish the last row of cards, each cell will have all nine bits of its row address field filled with patterns ranging from 000 000 000 to 111 111 111. The total time involved is 4 + 3.2 + 3.2 = 10.4 μseconds.

A similar algorithm can be described that will insert the "column address" in the cells. Both of these algorithms will be micro coded in TITANIC's central control unit.

CAM ALGORITHM

Suppose that the points of interest are represented by CAM cells which are responders to a search. The location of the cell on the machine grid is the analogue of the X Y coordinates we wish to average. Such a situation might readily arise if a picture has been mapped onto the CAM with each pixel occupying one memory cell.

To find the value of $\ddot{X}$ we first find the total "mass". We begin by setting the "activity bit" (A) of those cells that contain points of interest. The design of TITANIC is such that only those cells with A = 1 will participate in operations.

1. We perform "Global Sum" on the MASS fields of the participants. Let this sum equal M.
2. We execute a FIND ROW LOCATION, putting the row address in a cell field called ADDR.

3. For all participants in parallel we multiply MASS times ADDR and put the product in P.

   for all i: $P_i \leftarrow MASS_i * ADDR_i$

   This cell parallel operation is performed bit serially by shift and add. Each bit addition takes .4 μseconds, so the product of two eight bit fields can be calculated in 64x.4 = 25.6 μseconds.

4. We do a "Global Sum" of the products. Call this P.

5. The center of mass will be located on the row given by P/M.

6. Repeat steps two through five for the column addresses and we will have the column on which the center of mass is located.

The timing for this algorithm is as follows.

Find ΣM	160	x 1	=	160
Find Row	10.4	x 2	=	20.8
Multiply	25.6	x 2	=	51.2
Find ΣP (16 bit products)	320	x 2	=	640
Total time in micro seconds			=	872.0

In a conventional von Neumann machine we must fetch each cell, decide whether it is a participant and the perform a multiplication and an addition. Letting R=1 for participants and 0 otherwise, then the program below will discover the center of mass:

```
1  FOR I = 1 TO 512
2     FOR J = 1 TO 512
3        IF R(I,J)=0 GOTO 7
4        MASS = MASS + M(I,J)
5        XP = XP+I*M(I,J)
6        YP = YP+J*M(I,J)
7        NEXT J
8     NEXT I
```

Given the most efficient compiler, this will require $3x2^{18}$ additions and $2*2^{18}$ multiplications. The reader is welcome to make his own assumptions about the time required for an add and for a multiply, and to calculate a time to find the center of mass. Assuming all cells are active and one μsecond for each operation brings the total time to roughly $5x2^{18}$ or one and one-quarter seconds.

CONCLUSIONS

The TITANIC has been shown to be about 1,400 times faster than a RAM in computing the center of mass given reasonable assumptions. The reason it is not 2^{18} times as fast is that the individual operations in a CAM takes many times as long as they do in a RAM because they are performed bit serially. It is interesting to note that the time required for a CAM is the same no matter how many active cells are involved. In a RAM storing data in compressed vectors will allow the time to depend linearly on the number of active cells. Again assuming 1 μsecond adds and multiplies, it will take 5 μseconds to process each active cell. If there are fewer than two hundred active cells and if the data can indeed be stored properly, the RAM will be faster than the CAM.

DETERMINATION OF THE ROTATIONAL AND TRANSLATIONAL COMPONENTS OF A FLOW FIELD USING A CONTENT ADDRESSABLE PARALLEL PROCESSOR

M. E. Steenstrup, D. T. Lawton, C. Weems

Department of Computer and Information Science[1]
University of Massachusetts at Amherst

Abstract

The realization of motion perception in artificial systems will require highly parallel architectures. Here we demonstrate the use of a Content Addressable Parallel Processor (CAPP) [1,2] as an effective means of quickly and accurately decomposing a flow field into its rotational and translational components [3] to recover the parameters of sensor motion.

Organization of the CAPP

The CAPP is a VLSI-based Single Instruction Multiple Data (SIMD) machine designed at the University of Massachusetts [4]. It consists of a parallel processor containing 512x512 cells and a central controller. The central controller issues instructions to the parallel processor, controls loading and unloading of data in the parallel processor, and serves as an interface to the host computer and to secondary storage devices. It broadcasts data to the parallel processor bit serially, and the entire memory may be bulk-loaded in one video frame time (1/30 second). The central controller contains a set of micro-coded subroutines in ROM for performing high-level CAPP routines and a writeable control store for adding microcode.

The parallel processor consists of an 8x8 array of processing circuit boards and a set of boards which control CAPP edge treatment. Each processor board, in turn, consists of an 8x8 array of special purpose CAPP IC chips plus random buffer logic. Each chip then contains 64 cells, an instruction decoder, and some miscellaneous logic. There are eight basic instruction types recognized by the chip, each performed in parallel by the constituent cells. Most instructions take one minor cycle time (100 nanoseconds) to execute. Inter-cell communication is bit serial and is accomplished by a four-way (N, S, E, W) cell interconnect network, allowing for three types of edge treatments: dead-edging, circular wrap, and zig-zag wrap.

Each unit cell consists of 64 bits of fully static memory, four one-bit static "tag" registers A, B, X, and Y, a static carry bit register Z, and an ALU which continuously generates X NAND Y, X NOR Y, and X + Y + Z. Also, each cell contains logic for selecting a data source (a register (excluding Z), memory, an ALU function, broadcast data, or a neighboring cell (N, S, E, or W)), possibly inverting the selected signal, and storing it in a destination (a register or memory). The X register is the main tag register. Its output is connected to Some/None logic, indicating cell response, and to the neighbor communication network. The A register controls whether or not a cell is active. An inactive cell ignores all but a small set of instructions broadcast by the central controller. The Y register provides a secondary store for tag bits, while the B register provides a secondary store for activity bits.

Flow Field Decomposition Procedure

Our algorithm is an exhaustive search procedure which uses a set of rotational and translational flow field templates to find a component pair which can account for the motion depicted in a given flow field. Currently, 1000 rotational templates and 200 translational templates are used. These are generated from 100 axes which are uniformly distributed with respect to a unit hemisphere, and all pass through the origin of the sensor coordinate system. Each flow field consists of 16x16 vectors and is stored on a 2x2 square of chips consisting of 256 cells. The 2x2 chip arrangement facilitates flow field addressing. Each cell contains the horizontal and vertical components of a flow vector, each specified with 10 bits of precision.

The algorithm consists of four basic steps.

(0) The rotational templates are loaded into the CAPP, one template for each flow field location. Each flow field location corresponds to one of the squares in the CAPP diagrams shown in Figures 2a, 2b, and 2c. The rotational templates need only be loaded once since they are used in determining any flow field decomposition.

(1) A copy of the input flow field is loaded into each flow field location in the CAPP. Figure 1a and 1b show two sample input fields, both produced by the same motion and environment, except that Figure 1b was produced by adding random spike noise to Figure 1a.

1. This research was supported by DARPA under Grant N00014-82-K-0464.

This paper originally appeared in the proceedings of the 1983 International Conference on Parallel Processing, August 23-26, 1983

(2) A set of *difference fields* is formed by subtracting each rotational template from the copy of the input flow field stored with it. For each resulting difference field, the slope of each difference vector is computed by dividing the vertical component by the horizontal component. These subtraction and division procedures are performed in parallel across all flow fields represented in the CAPP.

(3) The similarity between the difference fields and each of the translational templates is evaluated, proceeding sequentially through all the translational templates. For a given translational template, this comparison is done in parallel with all difference fields stored in the CAPP and consists of the following steps:

(3a) The slope of each component vector of the translational template is loaded into the corresponding vector location of each difference field. The sign of the slope of each difference vector is compared with the sign of the slope of the corresponding translational template vector. If the signs agree, the absolute value of the difference between the slope of the difference vector and the slope of the translational template vector is computed, and then scaled according to the absolute value of both slopes. If the scaled slope difference does not exceed a predetermined maximum error value, then a vector match is designated at that position. The quantity of error permitted here allows the algorithm to be resistant to uniformly distributed Gaussian noise of low variance present in the original flow field.

(3b) For each difference field the number of vector slope matches is counted. If this sum exceeds a predetermined minimum number of matches (in our implementation, 75% of the field size), then the associated rotational and translational templates become a candidate pair for the flow field decomposition. Utilization of a minimum number of required matches ensures that only templates which are reasonably close to the actual motion will be chosen and permits some resistance to random spike noise. Figure 2a shows, for difference fields resulting from the input field in Figure 1a, the CAPP response to the translational template which is closest to the actual translational motion. Each black dot within a square represents a position in a difference field at which the slope of the difference vector matches the slope of the translational template. Figure 2b shows, for difference fields resulting from the input field in Figure 1b, the CAPP response to the translational template which is closest to the actual translational motion. Figure 2c shows the CAPP response to a translational template which is not close to the actual translational motion. This translational template is shown in Figure 3.

(3c) For all difference fields yielding at least the required minimum number of matches, the variance of the scaled slope difference is computed, and the difference field with the minimum variance is determined. This value is compared to the minimum variance found from processing the preceding translational templates. If this value is less than the preceding minimum, it becomes the new global minimum, and the rotational template associated with the difference field together with the current translational template become the current best candidate pair for the flow field decomposition.

Steps 3a, 3b, and 3c are performed for each translational template.

(4) The flow field decomposition considered to be the best is the rotational and translational template pair resulting in the difference field yielding at least the required minimum number of matches and the least slope difference variance. Utilizing minimum variance instead of the maximum number of matches, the algorithm has achieved better results, particularly for motions whose component parts lie between sets of templates. Figures 4a and 4b show the rotational and translational templates selected by the algorithm in the presence of and in the absence of noise, for the input fields in Figures 1a and 1b. These templates are the closest ones to the actual motions. Figures 5a and 5b show the difference fields resulting from subtracting the rotational motion in 4a from the original fields in Figures 1a and 1b respectively.

Experiments

Experiments have been performed with a CAPP simulator on a VAX 11/780 using a wide variety of motions and simulated environments. In all cases examined, the translational template closest to the actual translational motion was selected. The rotational template was always close to the actual rotational motion, but was sometimes not the closest template. The procedure proved to be resistant to limited Gaussian noise as well as to limited random spike noise in the original flow field. Applying motion to points at random depths produced results similar to those obtained in the noise experiments. The algorithm's performance degraded slightly if each flow vector component was specified by eight bits of precision instead of by ten.

The CAPP timing calculations revealed that the algorithm could perform the rotational-translational decomposition in slightly more than 1/4 second. If two CAPPs are used in parallel, then the time can be reduced to less than 1/5 second, since only half of the translational templates need be tested on each CAPP. Given fabrication techniques available in the immediate future, we expect execution times to be significantly improved. We also suspect that performance will improve by increasing both the number and size of the rotational and translational templates. This amounts to utilizing more CAPPs in parallel.

References

[1] Foster, Caxton C. *Content Addressable Parallel Processors*. Van Nostrand Reinhold, New York, 1976.

[2] Lawton, D.T., Steenstrup, M.E., Weems, C. "Determination of Rotational and Translational Components of a Flow Field using a Content Addressable Parallel Processor", COINS Technical Report, Computer and Information Science Department, University of Massachusetts, February, 1983.

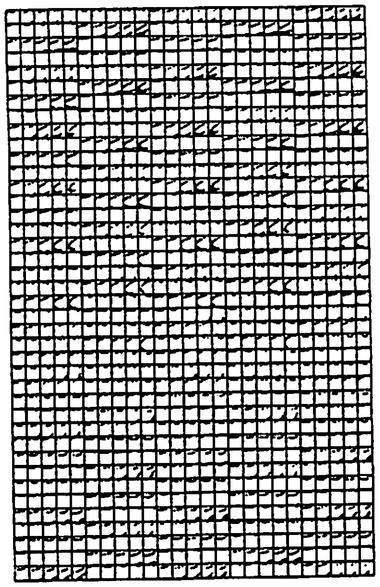


Figure 2c

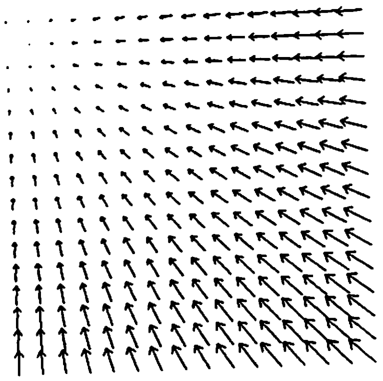


Figure 3

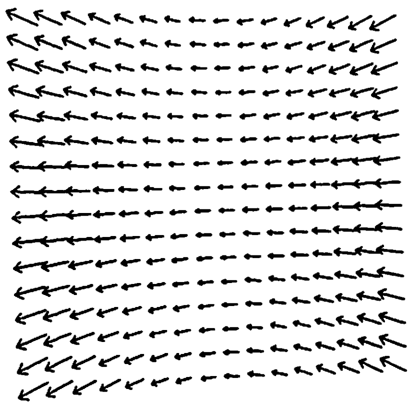


Figure 4a

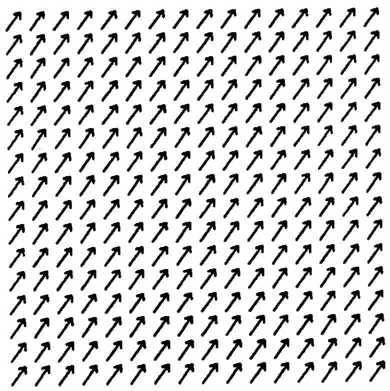


Figure 4b

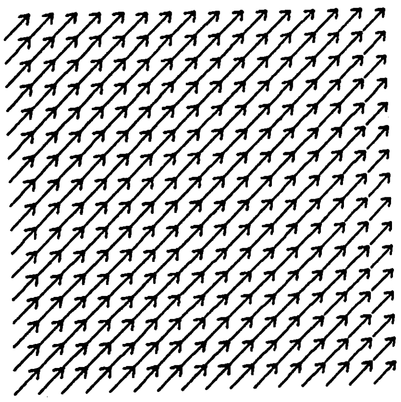


Figure 5a

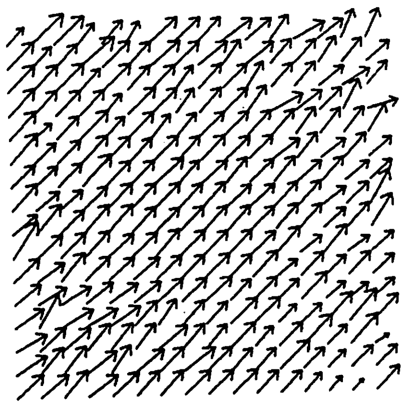


Figure 5b

[3] Prazdny, K. "Determining the Instantaneous Direction of Motion from Optical Flow Generated by a Curvilinearly Moving Observer." Proc. of the Pattern Recognition and Image Processing Conference. Dallas, Texas, August 1981, pp. 109-114.

[4] Weems, C., Levitan, S., and Foster, C. "Titanic: A Content Addressable Parallel Array Processor for Image Processing." IEEE International Conference on Circuits and Computers. New York, September 1982.

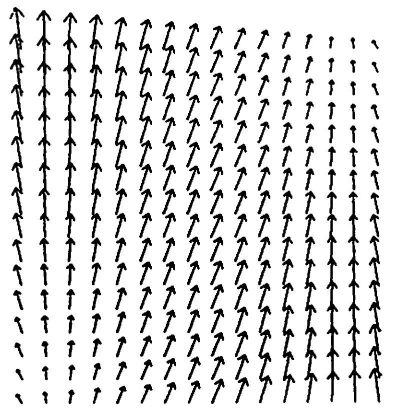


Figure 1a

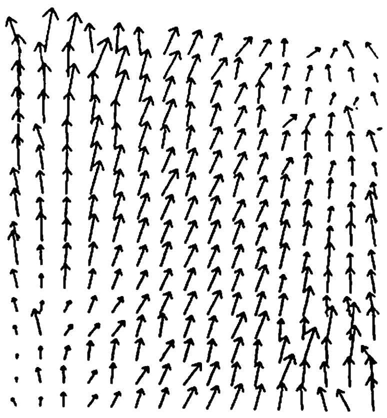


Figure 1b

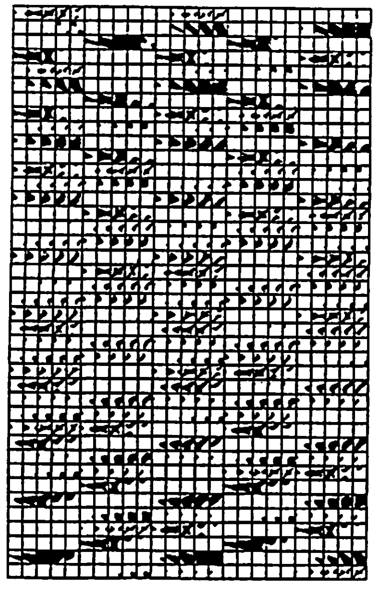


Figure 2a

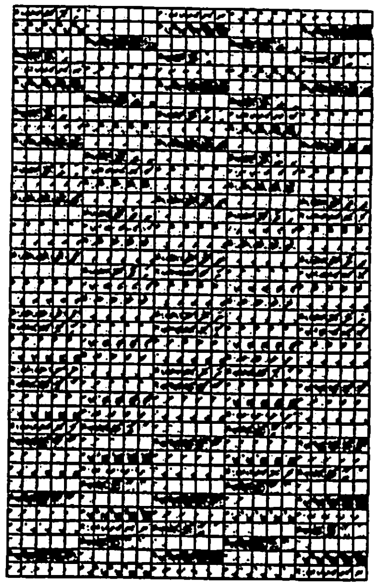


Figure 2b

Extracted from: "Incorporating content addressable array processors into computer vision systems"; 27th Annual International Technical Symposium & Instrument Display, August 21-26, 1983, San Diego, California; Charles Weems and Daryl T. Lawton

Associating symbolic descriptions with segmentations

A basic step in the functioning of autonomous, general purpose vision systems is the association of low level, spatially organized, symbolic descriptions with the results of segmentation, region and edge extraction processes (for example, the Primal Sketch [1] and the Regions-Segments-Vertices RSV representation of the VISIONS system [2]). Such a representation acts as a data base which is accessed by various recognition processes to determine the relations between different image strucutres. We have analyzed the implementation of some simple segmentation procedures using the CAAPP, such as zero-crossing extraction after convolution with a Gaussian-Laplacian mask [1] and histogram-guided segmentation [3]. Both these procedures are very rapid and are selectively sensitive to image information at different spatial frequencies and contrasts. We have found that associating symbolic information with the results of these segmentation procedures is most effective when the CAAPP is used in two different ways. In one of these, the symbolic labelling takes place in the same memory locations where the segmentation is, in parallel accross the CAAPP. In the second, a test is performed at a specified location for a particular type of image structure. If the test is successful at that location, the occurrance of the resulting image structure is stored in a network residing in another CAAPP. Once the symbolic data base is stored in a CAAPP, its associative character is used to make the extraction of complex structural relations possible [4].

The particular representation we have been developing is a version of the RSV representation of the VISIONS system [2]. In this the basic entities are Regions (connected sets of pixels); Segments (portions of the contours surrounding regions); and Vertices (selected points along contours). Each of these entities has specific attributes (such as area and extent for regions; length and orientation for segments). There are also specific relations between these entities (such as adjacencies between regions and edges). The process for associating a RSV representation with a segmentation present in the CAAPP is somewhat involved. It can also be made sensitive to different types of image structures by setting specific parameters. The first step labels the contour pixels of the regions determined by the segmentation. The second step extracts interesting points along these contours. This is done by a version of the interest operator described in [5]. The extracted interesting points become potential vertices since they tend to correspond to points of high curvature along the contour. The next series of steps propogate the labels of the selected interesting points along contours and into the interiors of regions. The values of different attributes are determined while this propogation occurrs. For example, for labels propogated along contours, information is stored concerning length, direction, and changes in direction. The attributes of a segment are then determined by the state of this information upon collision with another extracted interesting point. We are currently evaluating several different ways of dealing with collisions of labels in the interior of regions effectively.

References

[1] Marr, D. Vision, W.H. Freeman, San Francisco, 1982.

[2] Hanson, A. R. and Riseman, E. M., Computer Vision Systems, Academic Press, 1978.

[3] Kohler, R. "A Segmentation System Based on Thresholding", Computer Graphics and Image Processing, vol. 15, 319-338, 1981.

[4] Bonar, J. aand Levitan, S. "Real-Time LISP Using Content Addressable Memory". 1981 International Conference on Parallel Processing, Bellaire, Michigan, August 25-28, 1981.

[5] Lawton, D. T., "Processing Translational Motion Sequences", Computer Vision, Graphics, and Image Processing, vol. 22, 116-144, 1983.

REAL-TIME LISP USING CONTENT ADDRESSABLE MEMORY *

Jeffrey G. Bonar and Steven P. Levitan
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract -- The dynamic data structures of LISP require periodic garbage collection, prohibiting the use of most LISP implementations for real-time applications. We propose a scheme for implementing a real-time LISP system which uses Content Addressable Memory (CAM) to allow incremental garbage collection. In our scheme, all basic LISP operations, notably including retrieving a free cell for CONS, the list building function, and retrieving a current name-value binding, can be implemented with four or fewer CAM searches and very little other computation. Furthermore, CAMs are well suited for sufficiently inexpensive implementation with VLSI technology. Our system is not suitable if a virtual memory environment is needed, and becomes considerably more complex with CDR-coding. We are currently implementing a version of our scheme on a microcomputer.

Introduction

There are many real-time tasks which lend themselves to Artificial Intelligence (AI) solutions. Examples include assembly line robots, rapid transit system controllers, many complex scheduling tasks, and intelligent assistants for interactive devices. Such systems will most likely be designed and tested in LISP. The flexibility and expressibility of LISP have made it the "work-horse" language of the AI community. Can the prototype systems, still written in LISP, then be transferred to the final "production model"? We feel they can, but not with a standard LISP implementation.

The dynamic data structures of LISP require the use of "garbage collection" to reclaim memory as the data structures of the program grow and shrink. Garbage collection is typically done in a two phase process of first tracing and marking all active data, and then collecting all unmarked data. Depending on the size of the memory this operation can cause serious delays in processing. These delays can occur any time the program needs a new free cell. In particular they could occur during time-critical applications. An alternative space management scheme, reference counting, is unacceptable because it allows unbounded delays whenever a cell is released to the free list. This is because all successors of the released cell could become garbage and would have to be put on the free list at the same time. For these reasons a standard LISP implementation is not considered acceptable for real-time environments.

In this paper we discuss a real-time LISP implementation. Various LISP machines (e.g. Greenblatt [7] and Deutsch [4]) -- although usually presented as personal computing tools -- have shown that special purpose processors can vastly increase the speed and utility of LISP programs. Our paper shows how special purpose associative memory can be used to gain additional benefits.

Following Baker [2] we define a real-time list processing system as having "the property that the time required by each of the elementary operations is bounded by a constant independent of the number of cells in use". Baker's real-time LISP system involves incrementally compactifying and linearizing active cells by moving them between two memory partitions while leaving the garbage behind. Wadler [11] analyzes and summarizes a real-time scheme involving two processes running in parallel: the mutator is the application program while the collector keeps the free-list from becoming empty.

Our scheme uses specialized hardware, Content Addressable Memory (CAM), to create a very fast real-time LISP system, using a very simple set of algorithms. This speed and simplicity, which are the advantages of our scheme, are due directly to our use of CAM to examine all cells in memory in parallel.

We begin with a discussion of CAM. After presenting our real-time LISP scheme, its limitations are discussed. Finally, we discuss our implementation of this scheme.

Content Addressable Memory

General Description

Content Addressable Memory (CAM) is memory organized such that each word can compare its contents, rather than its address as in random access memory (RAM), with a value broadcast by the central processor [5]. This comparison process is done by all CAM words simultaneously. The processor can then interrogate the CAM to discover which words, if any, match the broadcast value.

Each word of a CAM memory has an associated responder bit (see figure 1). This single bit is reset if the contents of the word do not match the broadcast value, held in a register called the

* Support for this work was partially provided by the Army Research Office under grant DAAG29-79-G-0046.

This paper originally appeared in the proceedings of the 1981 International Conference on Parallel Processing, Bellaire, Michigan, August 25-28, 1981.

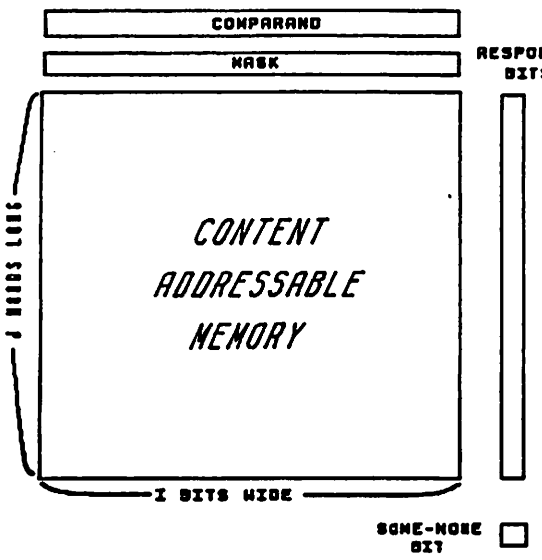


Figure 1. CAM Organization

comparand. All responder bits are typically OR'ed together and their disjunction is available to the processor as the signal SOME-NONE. Using SOME-NONE the processor can determine if there are any words that match the comparand. Additionally, a function to count the number of responders is often provided.

Another function the responder bits provide to the processor is to allow it to select a single responder if more than one exists. This is done by daisy-chaining the responder bits such that when the signal SELECT-FIRST is generated by the processor only the first responder in the chain remains set and all the others are reset. The processor can also perform the function SET-ALL which sets all the responder bits true. This is usually done before the comparand is broadcast to the memory.

Along with the comparand the processor also broadcasts a mask value. This is used by the words of the CAM to determine which bits of the word are to participate. For bits in the word where the mask bit is not set, no comparison takes place. The full operation is:

```
for all Words J
  for all Bits I in Word J
      Responder_bit[J] <-
          Responder_bit[J]
      and
        ( ( Mask_bit[I]
            and
            CAM_bit[I,J] = Comparand_bit[I]
          )
        or not Mask_bit[I]
        )
```

Note that this operation takes place in all words in parallel.

The processor can also perform the operations READ-RESPONDERS and WRITE-RESPONDERS. These allow the processor to read the contents of and change the contents of all words whose responder bits are set. This operation is often implemented to be under the control of the mask. Finally it is often convenient to allow the processor to access the CAM as a regular RAM and allow reading and writing of single words.

Suitability For Very Large Scale Integration (VLSI)

CAM is well suited to VLSI implementation. Foster [6] and Mead and Conway [10] both discuss the practical design of a VLSI CAM circuit. Two of the most important criteria for determining if a circuit can be implemented efficiently in VLSI are the regularity of circuit components and the number of input/output pins necessary [10]. CAM, like RAM, has an inherently regular sub-structure: the word.

To minimize the pinout (the number of input/output pins needed) several techniques can be used. First both the comparand and the mask values can be broadcast to the CAM in a bit serial protocol. This would mean that comparisons are done one bit at a time across all words in parallel. Bit serial operation would slow down the comparisons somewhat, but only on the order of the number of bits in a word. (a)

To minimize pinout further, the data in, data out, and address lines of the circuit can be multiplexed onto the same pins of the package. This technique has been used successfully for other types of VLSI circuits, for example, the Zilog Z8000 microprocessor. Minimizing the number of pins (and output drivers) would significantly reduce the cost of the circuit and increase the area available for storage.

The cost of CAM has been estimated to be 1.5 to 3 times the cost of an equivalent size RAM [6]. Memory sizes up to 64k of 32 bit words per circuit are not inconceivable [10]. Printed circuit cards containing 4k bytes of CAM have been on the market since 1978 [8].

Finally, CAM architectures lend themselves to a solution of the yield problem for VLSI. The problem is that a single flaw in one place of a VLSI circuit will cause the whole circuit to be unusable. As the physical area of VLSI circuits increases, so does the the probability of a flaw ruining a given circuit [10]. Since CAM operations, unlike RAM operations, do not depend upon where in memory a particular value is stored, it would be possible to disable flawed words of a CAM circuit, after testing, and still use the resulting (smaller) memory.

(a) The time per bit would be on the order of 10 nano-seconds. Therefore, even with bit serial operation, with reasonable word lengths, the time for a CAM operation would be on the order of the time for a machine instruction.

The Ideal CAM for LISP

For most applications CAM words are quite long. The Semionics CAM, for example, has 256 bytes (2048 bits) per word [8]. This allows entire records of data to fit in one word. A record might contain an employee's name, address, telephone number, pay rate, regular hours, overtime hours, etc. This would allow searching on any field of the record to retrieve it. Although there are standard techniques for spreading records across two or more CAM words, this slows the search considerably [6].

An ideal CAM for LISP has much shorter words since it is desirable to have only one LISP cell per CAM word. We discuss several types of LISP cells below. Here we concentrate our discussion on list cells which have seven fields: Flags, Garbage, Cell_type, Left, Left_type, Right and Right_type.

The Flags field is used for complex CAM searches involving logical disjunction and conjunction of different match criteria [6]. The bits in the Flags field are used as "temporary storage" for the responder bit of each word. The Flags field could be replaced by several auxiliary responder bits for each word and CAM operations to logically combine them [6] [8].

The Garbage field need be only one bit, indicating if the cell were "free". Using this bit we completely dispense with the Free list found in most LISP implementations.

The Cell_type field indicates if the cell is a list cell, a string cell, or any one of a number of other types. We discuss this in detail later. The Cell_type will facilitate any desired strength of typing and also allow cells of different types to share the same memory space (without partitioning) and the same garbage collecting scheme.

The Left_type and Right_type fields will also enforce typing. They allow us to pack short integers, bit strings, and pointers to machine language code into the cell. In addition they simplify the garbage collect process by allowing us to test whether a given Left or Right is a pointer.

The Left and Right fields would, as usual, be large enough to point to any other cell in memory. That is, a memory with 2**n CAM words (cells) would require n-bit Left and Right fields.

The CAM operations that need to be supported are SET-ALL, MATCH, SELECT-FIRST, SOME-NONE, READ-RESPONDERS, WRITE-RESPONDERS, READ, and WRITE as outlined above. The COUNT-RESPONDERS is not necessary. Additionally, for the name-value binding scheme outlined below, a FIND-GREATEST function would be helpful.

The Algorithm on a Simplified LISP CAM

We begin the description of our algorithm using a CAM in which each word contains one simplified LISP cell with only three fields: Left (CAR) and Right (CDR), which both point to another LISP cell, and a Garbage bit (see figure 2).

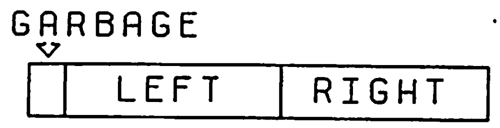


Figure 2. Simplified CAM LISP Cell

The key observation about garbage collection with such a cell is that we can find if there are any pointers to a given cell with two CAM operations: a CAM search of the Left fields and a CAM search of the Right fields, of all cells in memory.

Any practical implementation would use CAM words to hold several different kinds of cells. In particular, our implementation uses special cell types to allow garbage collection of strings, name-value bindings, and the primitives of the GRASPER graph processing language [9]. We discuss how these special cells are handled after presenting the simplified one cell type algorithm.

When a free cell is needed, a CAM search is done for a cell whose Garbage bit is set. This is done by the Supply_free_cell routine in figure 3 (which appears at the end of the paper). One of these cells is selected with the SELECT-FIRST operation. This cell, call it C, is returned as the needed free cell. It is still necessary, however, to propagate "garbageness" to the sub-structures of this cell. This is done by the Potentially_make_garbage routine in figure 3. We do this by first CAM searching the Left and Right fields of all other cells for equality to C.Left. If there are no responders to this search (SOME-NONE has value NONE), then the cell pointed to by C.Left is garbage and we set its garbage bit. If C.Left = NIL, then the search need not be done. We handle C.Right in an identical way. The algorithm requires that all cells be initialized with their Garbage bits set and their Left and Right fields set to NIL.

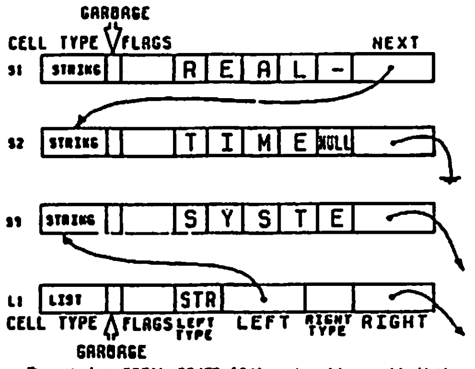
A piece of list structure potentially becomes garbage when one of possibly many pointers to it is deleted. This can occur in several ways during the execution of a program. The functions REPLACA and REPLACD explicitly delete pointers from the left (CAR) and right (CDR) fields of list cells. The function SET (assignment) also deletes the pointer to a variable's old value. These functions all call the routine Potentially_make_garbage on the the pointer they are deleting. This routine determines whether to set the Garbage bit of the head cell of the structure pointed to. All sub-structure will be handled if that head cell is made garbage and when it is actually reused.

Circular lists cannot be garbage collected in our regular scheme because there is always a pointer to any cell in the circle. They can be accommodated, however, either by requiring the user to release them explicitly, or by simulating them with a "lazy evaluation" scheme (see Allen [1] for details on lazy evaluation).

Extensions For Other Cell Types

Our scheme is easily adapted to other kinds of dynamic data structures. Here we will discuss an implementation for strings. Remember that, as discussed earlier, our LISP list cell actually has seven fields. The simplified cell is augmented with a Type field for the cell and for the Left and Right fields. These fields are necessary for the algorithm, but also allow us to enforce typing. Typically, typing is done by putting all of one kind of data together so that address alone can be used to determine type. In our scheme, if a field is of type T, it may only point to a cell of type T.

Strings are made up of linked lists of cells (see figure 4). String cells, like any other cell type, must be fit into the existing size CAM word and must have Type, Garbage, and Flags fields. They also have several bytes of character data and also Next, a Cell_ptr implicitly of type string. The implicit typing saves space in the cell and it does not cause a problem, since string cells can point only to string cells.



The string "REAL-TIME" (S1) and a list cell (L1) whose CAR points to a string beginning "SYSTE".

Figure 4. Example of CAM LISP Strings

Unlike a list, when the head of a string becomes garbage, the entire string is known to be garbage. Potential "garbageness" need only be propagated down the Next field link and the Other_ptrs to operation need not be done.

For example, in figure 4, assume that cell L1 is made garbage. When the cell is chosen to be reused, we attempt to propagate "garbageness" to L1.Left. If there are no other pointers to cell S3 the string "SYSTE..." becomes garbage. S3 is marked garbage and when it is reused no other CAM searches need be done.

Atoms are also implemented as special cells. In addition to the Flag, Cell_type, and Garbage fields, atoms have a Value field and Value_type field, pointing to the atom's static binding, and a Print_name field implicitly of type string (that is, pointing to a cell of type string).

A Truly Associative "A-List"

In LISP each function call creates a set of name-value bindings which exist during the execution of the function and disappear at its completion. This is roughly equivalent to the formal to actual parameter bindings in other programming languages. Traditional binding schemes use one or more lists to associate names with values. A list used this way is called an A-List for Association-List (see Allen [1] for more details).

In our scheme the A-list, like the Free list, does not exist. Instead the bindings are held in a set of distinguished cells, existing anywhere in CAM. When entering a new environment, we increment an environment counter and create a set of CAM cells to hold the names bound in that environment, their values, and the new environment number. Now we can ask the question above as a single compound CAM search for a name-value binding within an environment, and retrieve the current binding directly. Since the current value of a name might not be in the most current environment, we need to search for the greatest environment number for that name.

When an environment is exited, a pair of CAM operations is executed. First a search for all environment cells with the current environment number, followed by a WRITE-RESPONDERS operation to make all these cells garbage. Since no other cell will point to these binding cells, even if some do point to their descendants, they can all be turned into garbage in one operation.

Figure 5 summarizes all the cell types discussed in this section.

Garbage, Flags, and Cell_type fields occur in each cell.

List	Left, Left_type, Right, Right_type
Atom	Print_name (implicitly of type string) Value, Value_type
String	Character_1,...,Character_n, Next (implicitly of type string)
Environment	Environment_number, Name (implicitly of type atom), Value, Value_type

Figure 5. Summary of Cell Types

Other Issues

CDR-Coding

Many recent LISP implementations use CDR-coding, compact encodings of list representations which take advantage of statistical regularity in list structures (see Bobrow and Clark [3] for a summary and discussion of these schemes). A CAM augmented LISP with CDR-coded cells is easy to imagine, though it would require considerable extra time and complexity in the implementation of the basic LISP operations. Finding all pointers to a given cell would, in general, require a CAM search for each possible interpretation of a cell pointer field.

Given decreasing hardware costs, we did not feel it necessary to compromise the simplicity and speed of our algorithms. In particular, CDR-coding offers no solutions to our primary goal of real-time operation since it reduces space rather than time needs.

Virtual Memory

Our scheme does not support virtual memory. In general, it would be impossible to perform the test Other_ptrs_to on a given cell without paging every active page of the virtual memory into CAM. The application programs we envision for our system can always be tested in advance to determine their space needs. More CAM cells can always be added without a time penalty.

Our Implementation

We are currently implementing the LISP system discussed above using a Z80-based microcomputer and 80K bytes of CAM. The CAM, Semionics Recognition Memory (REM) [8], is organized as 320 256-byte words (called "super words" in the company literature). We do not need such long words and have cut the memory into vertical slices, yielding 32 LISP cells per word. Although this means that many of our CAM operations will have to be repeated 32 times in the worst case (once for each vertical slice), the system runs at an acceptable speed. The real-time properties of our system remain intact.

The project is a pilot study to examine two issues. First we wish to show that even with relatively slow CAM (bit serial searches on the order of 1 micro-second per bit) which is not organized to our needs, we can build a real-time, self-contained LISP system.

Second, the graph processing language GRASPER uses many associative operations which can be supported by CAM. (b) GRASPER objects have the same dynamic allocation needs as other LISP objects. We will embed a subset of the GRASPER language into our LISP system using the cell typing conventions already discussed. We expect to show the advantages of a CAM based GRASPER system as part of a feasibility study for the design and implementation of a state of the art CAM on our VAX 11/780.

(b) GRASPER is used to represent and operate on semantic nets, augmented transition networks (ATNs), HEARSAY-II style blackboards, and other associative data structures used by AI projects at the University of Massachusetts.

Conclusions

We have presented a scheme for implementing a real-time LISP system by using Content Addressable Memories for storage of the basic LISP cells. Not only does our scheme perform all elementary operations in real-time, it also has the following other advantages:

1. All cells are available for use, in contrast to other real-time schemes.
2. Retrieving the correct value for a name can be be done truly associatively, always requiring only two CAM operations.
3. Strings and other dynamic data types can be elegantly and efficiently integrated into the basic scheme without partitioning memory.
4. CAM is eminently suited to modern VLSI implementation techniques.

Our scheme does have limitations, however:

1. Circular lists cannot easily be garbage collected.
2. Our scheme does not lend itself to a virtual memory environment.

We believe that even given the above limitations, our scheme is an attractive alternative for self-contained, dedicated systems. It is usable in a real-time environment and all basic LISP operations perform extremely quickly. We believe that tested AI systems written in LISP could be transferred to a CAM-augmented LISP machine without costly redesign and without recoding in a standard systems programming language (e.g. assembly language or Ada). In this way we hope our scheme will aid in the creation of simpler yet more powerful computer-controlled systems.

Acknowledgements

We would like to thank Caxton Foster for his wealth of knowledge about CAMs and "hardware solutions". Raj Wall is working with us to implement the system. Our interactions with him have been very valuable. John Lowrance provided us with much information about the basic implementation and operation of LISP and GRASPER. We would also like to thank Jeff Conklin, Dan Corkill, Elliot Soloway, Bev Woolf, and the referees for their careful reading of earlier drafts.

References

[1] John Allen, Anatomy of LISP, McGraw-Hill Book Company, (1978), pp. 149-153.

[2] Henry G. Baker, "List Processing in Real Time on a Serial Computer," Communications of the ACM, (April, 1978), pp. 280-294.

[3] Daniel G. Bobrow, and Douglas W. Clark, "Compact Encodings of List Structure," ACM Transactions on Programming Languages and Systems, (October, 1979), pp.266-286.

[4] L.P. Deutsch, "A LISP Machine With Very Compact Programs," Proceedings 3rd IJCAI, Stanford, California, (1973), pp. 697-703.

[5] Caxton C. Foster, Computer Architecture, second edition, Van Nostrand Reinhold Co., (1976).

[6] Caxton C. Foster, Content Addressable Parallel Processors, Van Nostrand Reinhold Co., (1976).

[7] R. Greenblatt, LISP Machine Progress Report, AI Lab. M.I.T., Cambridge, Massachusetts, memo 444, (August,1977).

[8] Sydney Lamb, "An Add-In Recognition Memory For S-100 Bus Microcomputers-Parts 1,2, and 3," Computer Design,(August-October, 1978).

[9] John D. Lowrance, GRASPER 1.0 Reference Manual, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, Report 78-20, (December,1978).

[10] Carver Mead, and Lynn Conway, Introduction to VLSI Systems, Addison-Wesley Publishing Co., (1980).

[11] Philip L. Wadler, "An Analysis of an Algorithm for Real Time Garbage Collection," Communications of the ACM, (September, 1976), pp. 491-500.

Figure 3. Algorithm for CAM Augmented LISP Garbage Collection

```
function Supply_free_cell : Cell_ptr;
    (* called by CONS to find a cell it can use
       to build a list structure with. In addition
       this function does the incremental garbage collect *)
    var Free_cell, Temp : Cell_ptr;
    begin
    search for first Free_cell from Cell
        where Cell[Free_cell].Garbage
        do begin
           if Cell[Free_cell].Left <> Nil_ptr
               then begin
                    Temp := Cell[Free_cell].Left;
                    Cell[Free_cell].Left := Nil_ptr;
                        (* These two make sure a check for other
                           pointers = Cell[Free_cell].Left will
                           not respond to that field itself *)
                    Potentially_make_garbage (Temp)
                        (* propagate "garbageness" *)
                    end;

           if Cell[Free_cell].Right <> Nil_ptr
              then begin
                   Temp := Cell[Free_cell].Right;
                   Cell[Free_cell].Right := Nil_ptr;
                       (* These two make sure a check for other
                          pointers = Cell[Free_cell].Right will
                          not respond to that field itself *)
                   Potentially_make_garbage (Temp)
                       (* propagate "garbageness" *)
                   end;
           return Free_cell
           end

        else System_error ("Cell space full")
    end;


procedure Potentially_make_garbage (C : Cell_ptr);
    begin
    Cell[C].Garbage := not Other_ptrs_to (C)
    end;


function Other_ptrs_to (C : Cell_ptr) : boolean;
    var Responder : Cellptr;

    begin
    search for Responder from Cell
        where         not Cell[Responder].Garbage
                  and Cell[Responder].Left = C
        do return true
        else search for Responder from Cell
                    where        not Cell[Responder].Garbage
                             and Cell[Responder].Right = C
                    do return true
                    else return false
    end;
```

(continues on the next page...)

(...figure 3 continued)

```
procedure Init_CAM;

   var Responder : Cell_ptr;
   begin
   search for Responder from Cell
      where true
      do begin
         Cell[Responder].Garbage := true;
         Cell[Responder].Left := Nil_ptr;
         Cell[Responder].Right := Nil_ptr
         end
   end;
```

Notational Conventions

The CAM is seen as an "associative" array of records, where each record represents the data in one CAM cell. Standard indexing into the array allows us to treat the CAM as RAM. From the above we have two data types:

```
Cell_ptr = 1..Num_cells;


Cell = associative array [Cell_ptr]
          of record
             Garbage : boolean;
             Left, Right : Cell_ptr
            end
```

The basic CAM operation is:

```
search for [ first ] <index variable into CAM>
   from <CAM array name>
   where <boolean expression>
   do <statements>
   else <statements>
```

The <index variable> is available within the do <statements> to syntactically represent all cells that meet the search criteria. This <index variable> is a free variable ranging over all possible values, that is, indexing all cells in the CAM array. For each CAM cell where the <boolean expression> is satisfied, the do <statements> are executed. The do <statements> are performed in parallel for these cells. In the case of "search for first", the index variable gets set to the value of the first responder. In the case that no cells satisfy the <boolean expression>, the else <statements> are executed. Typical CAMS do not support the generality implied by this construct. In particular, arbitrarily complex <boolean expressions> will take N CAM searches, where N is the number of disjuncts in a disjunctive-normal-form version of the <boolean expression>, and do <statements> are limited to assignments to the cells indexed by the <index variable>. Other operations can be supported either by more intelligent CAM cells or by a micro-coded CAM controller. Our algorithms use the construct in ways easily implemented in CAM.

DECRYPTION OF SIMPLE SUBSTITUTION CYPHERS WITH WORD DIVISIONS USING A CONTENT ADDRESSABLE MEMORY*

Rajendra S. Wall

Abstract: The problem of decrypting simple substitution cyphers can readily be solved by pencil and paper. It can also easily be attacked by various computerized approaches. This paper shows the results of a table look up approach aided by simulated content addressable memory hardware.

Keywords: Content Addressable Memory, Parallel Search, Table Look-up Search, Simple Substitution Cyphers.

Introduction.

The simple substitution cypher with word divisions, also known as the Aristocrat cypher, is encrypted by creating a one to one mapping of the alphabet onto itself and applying that mapping to the plaintext to be encoded. Various methods have been proposed for decryption by humans with pencil, paper and patience [4, 9, 6]. Human and computer abilities have been used together to solve the problem [1]. In addition, artificial intelligence approaches, attempting to simulate human methods with computerized heuristics, have been used [8], as well as abstract image processing relaxation procedures [7].

A more straightforward and less heuristic approach involves table look up. Each time a crypt word is partially decyphered a dictionary of possible words is searched to see if any words in the dictionary could be replacement words to fit and fill in the partial word. When using a general purpose computer with standard memory architecture this approach is tedious and impractical. As the dictionary of possible words grows to a useful size the amount of time spent looking up partially decyphered words quickly becomes unacceptable.

Standard computer memory architecture can be thought of as a series of numbered boxes in each of which we place one word of the possible word dictionary. Standard memory access in a general purpose computer involves asking: "What is in box one?" or "What is in box four hundred and thirty two?" Performing the search of the dictionary is then asking: "What is in box one? Does it look like this partially decyphered word?" What is in box two? Does it look like this partially decyphered word?" and so on, for as many words as are in the dictionary. When the dictionary is large, this could take a long time to accomplish.

Content addressable memory (CAM) architecture allows a number of memory access options that are unavailable in standard memory acrhitecture [2]. These extra options involve more hardware and consequently are more expensive than standard memories. Recent advances in silicon chip technology have made content addressable memories more affordable. Companies such as Semionics offer a memory board for micorprocessors that they call "recognition memory." Though not full content addressable memories these boards do have some of the needed features.

*This research was supported in part by a grant from the Army Research Office Number DAAG 29-79-G-0046.

A content addressable memory begins with a set of numbered boxes similar to standard memory. To each box is added comparison circuitry that allows direct comparisons between a central data item and each memory box. Since each memory box has this circuitry all the memory boxes may be interrogated at the same time. Circuitry is also added to allow the computer to know which boxes have comparison results to return and what those results are. Thus the search of a dictionary with a CAM begins with loading the dictionary into the memory one word per box as in the standard memory search. But to perform the search we only need ask once: "What boxes have words in them that look like this partially decyphered word?" Since all boxes have circuitry to compare the partially decyphered word against their own contents they can all perform this comparison in parallel. The comparison takes the same amount of time no matter how many words are in the dictionary. Through the response circuitry it can be determined how many boxes had matches as well as which specific boxes had matches. Words that signalled matches to the partially decyphered word are considered as candidates for being possible replacement words for that partially decyphered word.

The Search Algorithm.

The system used a depth first algorithm to perform a tree search in order to implement the decryption process (Winston). A tree is a data structure consiting of nodes and links between those nodes [5]. One node is the root node. All other nodes are either descendants of that root node or descendants of descendents. An example of this type of structure is seen in a family tree: The head of the family is the root node and all the children, the children's children and so on are the descendant nodes. Links are thus made between parent nodes and descendant nodes. A depth first search of a tree involves examining the nodes of the tree by first following a particular branch or path for as long as possible before finally having to back up and try a new path when the old one reaches a dead end. For example, in the family tree a depth first search might involve following the line of first born males. The search would follow the path of first born males through generation after generation until it reached a node where there was no first born male (perhaps the first born was female). At this point it might back up to the previous generation and examine the children of the second born male, again following the path of first born male. If the second born male did not have a first born male child then the search would move to the third born male. If there was no third born male then the search would have to back up another generation and look at that generation's second born male, and so on.

In the decryption problem, each node is a guess about some partially decyphered crypt word. The guesses are drawn from the set of all words in the dictionary of the proper length and having A's where the partially decyphered word has A's, B's where the partially decyphered word has B's and so on and anything where the partially decyphered word has blanks. Blanks represent unknown, unconstrained positions in the partially decyphered crypt word. Since the dictionary is ordered by frequency the first dictionary word that matches the partially decyphered crypt word is the most frequent word that fits, and is hopefully the best choice. The search thus always assumes that the crypt was made up of words that occur often in usage before it is forced to look at less frequent choices. A node's descendants are guesses about other crypt words under the assumption that the guess in the parent node was correct. Any given node could then have any number of descendant nodes but only one parent node. The descendant nodes are grouped by which crypt word they are replacements for.

A search of the tree starts by choosing a crypt word to make guesses about. In general this initial choice is arbitrary. A replacement word is chosen from the dictionary to be the guess for this crypt word and is placed in the root or top most node of the tree. The descendants of that node are then generated. This generation is done by taking the letter replacements suggested by this guess/crypt word pair and inserting those replacement letters into the other crypt words, creating partially decyphered words. These partially decyphered words can then be used to interrogate the dictionary. At each level farther down into the tree there are fewer and fewer crypt words left to make guesses about. Eventually the process will stop when either every crypt word has a replacement word from the dictionary as its guess (this path through the tree is then called a solution), or it is discovered that no solution exists. When no complete solution is found the partial solution of the crypt at the deepest penetration of the search tree -- thus the most complete but not necessarily the most correct -- is returned.

The depth first tree search algorithm specifies the selection process of which crypt word to make guesses about and which dictionary replacement word to choose to be that guess. The decision of which crypt word to pursue next is made by looking at all remaining partially decyphered crypt words and by interrogating the dictionary noting how many responders and thus how many possible replacement words each partially decyphered word has. The crypt word that has the fewest number of replacement words is the one chosen. This forces the search to take the easiest path to the solution first. The choice of which dictionary replacement word is to be used as the guess for this crypt word is done by choosing the most likely word that would show up under the constraints of previous guesses if this crypt conformed to standard English text frequencies. This is done by ordering the dictionary based on frequency of usage and then picking the first replacement word through the ability of the CAM to identify the first responder.

A crypt word that has no possible replacements at all will obviously be chosen as the one with the fewest replacements. When this happens it means that there is no way to solve the rest of the crypt if the previous guesses are left unchanged. This is a dead end for the search. Since during the first pass of the search it is assumed that there does exist a solution and that all words in the crypt are in the dictionary the only answer is that one of the previous guesses was incorrect. The first candidate for correction is the parent node. This node contains a guess for a particular crypt word. Since this guess is incorrect the next most frequent alternative replacement word from the ordered dictionary is chosen to be the guess for that crypt word. The search then continues downward assuming now that this new guess is correct. If there were no more alternative dictionary replacement words then the search has again been taken to an untennable position and the parent node of the node with no alternatives is deemed incorrect, initiating the above procedures for handling incorrect guesses.

If it is found that the root node has no more alternative dictionary replacement words then the search is terminated with only a partial solution to the crypt. Up to now the search has assumed that all the words are in the dictionary and that a partial word that is not found in the dictionary is in error. If the first search of the tree is found to have no solution under this assumption then the search may be reinitiated at all its previous dead ends with a new assumption, that there may be at most one word that is not in the dictionary. Any solution found now will only be a partial solution. If this second search again produces no solution then at most two words are assumed to be missing from the dictionary, and so on. Appendix 1 shows an example of the system.

Implementation and Test Results

Implementation was made on the University's Cyber 170 using the APLUM version of APL [10]. Simulation of the content addressable memory was done with the APL inner product operator. The dictionary was built from the Brown sample of edited American English [3]. The full dictionary contains over thirty six thousand words. For this problem, reduced dictionaries were used to collect statistics. The first dictionary included the top five hundred words of each word length from one to ten , giving a total dictionary size of slightly more than four thousand words. The second contained the top one thousand words of each word length giving a total dictionary size of around eight thousand words. The third and final dictionary contained all thirty six thousand words. In each search no unsolved words were allowed.

The system worked well in those cases where all cryptanalytic systems, human or machine, work well: long messages, plaintext words of high frequency and repeated text fragments. On those samples of short messages, words of low frequency (but still in the dictionary) or no repeated text it would have some trouble but would not have a lot of backtracking if the crypt words were of unique patterns. It worked worse on nonsense phrases such as "Grown under the quick jazz film display box" that were short, had low frequency words, little or no repeated text, and no distinguishing patterns. It obviously worked worst on crypts that contained words that were not in its dictionary, since such crypts necessitated only partial solutions.

Using the APLUM system it is possible to obtain statistics of processor time spent executing statements or routines. This allowed examination of the time spent in the content addressable memory simulation routine versus the time spent in all the other routines combined. Appendix 2 shows the results of the system.

The first table is a list of five crypts made of words from the four thousand word dictionary. The system was run three times against each crypt, once per dictionary size. Below the list is the average amount of computer time spent performing the non-CAM related parts of the search. This shows the time that would be spent decrypting the cyphers if a CAM was available that could do the dictionary look up in parallel. The results showed that the major drawback to table look up driven procedures, that of incredible delays due to dictionary examination, are eliminated by the use of content addressable memories. The decryption time becomes a function of the difficulty of the message and is unrelated to the size of the dictionary.

The second table shows the results of putting a nonsense word "abcd" (encrypted as "OJPM") at the end of the first crypt and then making four more runs, one with "abcd" as the five hundredth four letter word in the four letter word dictionary, one with "abcd" as the one thousandth four letter word, once as the last four letter word and once with it not in the dictionary at all. These trials were made to see if the system would be slowed down by having to dig through erroneous guesses about the crypt word "OJPM" as it becomes partially decrypted. This would be expected since the CAM returns all words that fit the pattern of a partially decyphered crypt word in order of decreasing frequency and if the word you desire is at the bottom of the dictionary it may not be the first word in the returned list. It turned out this was not the case. The computer time did not increase as much as would be expected when the sought word was placed deeper and deeper in the dictionary. This was due to the nature of the search procedure, rather than the CAM. By the time the system decided to pursue the crypt word "OJPM" the result was so constrained that only "abcd" was returned. When the word was not in the dictionary at all the entire decryption tree had to be searched and only a partial solution was returned. Thus the much higher amount of computer time spent in that run.

Conclusion

This paper has shown the difference between standard and content addressable memory achitectures in terms of solving a particular table look up problem, decryption of simple substitution cyphers with word divisions. We have also shown how a content addressable memory can eliminate the problems caused by searching a large dictionary.

APPENDIX 1

This execution example is of a crypt taken from the January-February 1978 issue of the American Cryptogram Association publication *The Cryptogram*. The dictionary size was the full thirty six thousand words. The original crypt was:

```
E SET ANU MEOTX QTUARKJMK ETJ JUKX TUY HXK OY OX ROQK E LEWSKW ANU
VRUAX NOX LOKRJ IHY JUKXTY XUA OY
```

The first thing the system does is break the crypt into individual words. It then looks to see where would be the easiest place to start work. Since it has no constraints, no letters are known yet, all it can do is choose the word with the shortest dictionary length. In this case the one letter word 'E' can have at most 26 possible replacements, 'a' through 'z', so it is chosen as the first word to search on. The system takes the first choice from the list of one letter words, which happens, by the fact that the list is in frequency order, to be the word 'a', and substitutes that choice into the crypt wherever 'E' is found:

```
E SET ANU MEOTX QTUARKJMK ETJ JUKX TUY HXK OY OX ROQK E LEWSKW ANU
a  a       a              a                           a  a

VRUAX NOX LOKRJ IHY JUKXTY XUA OY
```

The system then looks for the word that is now the most constrained. The obvious choice is the second occurrence of 'E', since given the choice made for the first occurrence of 'E' it must also be 'a'. After making that substitution into the crypt (which didn't change anything, but the system doesn't know that) it again looks for the most constrained word, the word with the fewest replacement choices. In order to do this it must query the content addressable dictionary with each partially decyphered crypt word, e.g., ' a ' for 'SET', ' a ' for 'MEOTX', and so on. When it is finished, it finds that the crypt word 'ETJ' has only 66 choices making that the fewest number of choices of any of the other partial words. Since 'ETJ' is a three letter word the system takes the first three letter word from the list of the 66 possible three letter words that begin with 'a', the word 'and', and substitutes it into the crypt:

```
E SET ANU MEOTX QTUARKJMK ETJ JUKX TUY HXK OY OX ROQK E LEWSKW ANU
a  an      a n   n    d   and n    n                  a  a

VRUAX NOX LOKRJ IHY JUKXTY XUA OY
              d     d   n
```

This process of finding the most constrained word and substituting the first dictionary word that fits that most constrained crypt word into the crypt, giving more constraints so that other words can be looked up, and so on, continues through the choices of 'knowledge' for 'QTUARKJMK', 'does' for 'JUKX', 'gains' for 'MEOTX', 'is' for 'OX', 'like' for 'ROQK', 'sow' for 'XUA', 'use' for 'HXK', 'doesnt' for 'JUKXTY', 'not' for 'TUY', 'it' for 'OY' twice, 'who' for 'ANU' twice, 'his' for 'NOX', 'field' for 'LOKRJ', and 'blows' for 'VRUAX'. The crypt now look like:

```
E SET ANU MEOTX QTUARKJMK ETJ JUKX TUY HXK OY OX ROQK E LEWSKW ANU
a  an who gains knowledge and does not use it is like a fa  e  who

VRUAX NOX LOKRJ IHY JUKXTY XUA OY
blows his field  ut doesnt sow it
```

It can be seen that the choice of 'blows' for 'VRUAX' might cause problems. When the system examines its next choice for a most constrained word, 'but' for 'IHY', it discovers a UNIQUE SUBSTITUTION FAILURE. This means that the system has realized that it cannot have both 'V' and 'I' be replaced by 'b'. So it throws out 'but' and tries the next dictionary word in the list of three letter words that meet the constraints of ' ut'. This next word is 'out', which also causes a unique substitution failure. The third word on the list is 'put', which fits. It then goes to the crypt word 'SET', which is now the most constrained. The first choice for a dictionary word that fits ' an' is 'can'. But when the search is resumed assuming that the substitution 'can' for 'SET' is correct it is found that there is no solution to the crypt word 'LEWSKW". The system then must back up to the previous level which it assumes was in error, and removes the choice of 'can' for 'SET'. It then tries the next word on its list of words that fit ' an' and uses it as a replacement for 'SET'. This new replacement is the word 'man' which allows the crypt word "LEWSKW' to be solved as 'farmer'. Since there are no more words to work on the system ends the search with its final solution:

```
E SET ANU MEOTX QTUARKJMK ETJ JUKX TUY HXK OY OX ROQK E LEWSKW ANU
a man who gains knowledge and does not use it is like a farmer who

VRUAX NOX LOKRJ IHY JUKXTY XUA OY
blows his field put doesnt sow it
```

Notice that although to a trained speaker of English this solution can be seen to be incorrect, at the level at which the system operates the solution is perfectly legitimate. Since both 'V' and 'I' only occurred once there is no way that from just the rest of unrelated crypt text the system can tell which should be 'p' and which should be 'b', since both 'plows' and 'blows' as well as 'put' and 'but' are words in the system's dictionary.

APPENDIX 2

Table 1

Five Crypts from the Four Thousand Word Dictionary

(1) OXAQF AEQDF GQZQFOH MDRPLRRDCZ EAQ PCWWDAAQQ SOR QKIQPAQM AC ORN XCF
WCFQ WCZQT XCF RFDQZADXDP FQRQOFPE

(2) ORW VWDWXGW HOOPCXWZ BUWGOMPXWV ORW LP JWCXIWXO HLWXOG OWGOMIPXZ ORHO
ORW FPXGKMCHFZ MXFSUVWV IWIAWCG PD ORW FPIIUXMGO KHCOZ

(3) KXTC UZOCNWUH QTENHOEE ZUHUFOZOHS EWRXXDE OPBOWSOY SRO JXQ ZUCLOS SX
WXHSNHTO SX NZBCXAO YOEBNSO SRO ESUSO XK SRO OWXHXZG

(4) UPB CEDUBK RUWUBR UOWKB WJNWRRWKLO RWDK UPB FWTWEBRB ZDAA TCOSPWRB
UPOBB ECSABWO RCNJWODEBR HLO CRB WQWDERU OCRRDWE ZWO RPDTR

(5) FAR BRHMFVY BMQK JR JRYR NSEHRYMDER FV M GQYBF BFYQOR QG FAR BVNQRF SHQVH KRNREVWRK M KRGRHBR MXMQHBF VSY BRPVHK BFYQOR PMWMDQEQFI

Dictionary Size	Average Non CAM CPU Time in msec
4224	518.2
8147	551.8
36803	559.4

Table 2

Time Spent Searching for Word at Various Depths

Depth of Nonsense word "abcd"	Non CAM CPU Time in msec
500	170
1000	182
7000	194
not in dictionary	7192

REFERENCES

1. Edwards, D. J. 1966. *OCAS: On-line cryptanalysis aid system.* MIT Project MAC Technical Report TR-27. May.
2. Foster, C. C. 1976. *Content addressable parallel processors.* New York: Van Nostrand Reinhold.
3. Francis, W. N. 1964. *A standard sample of present day edited American English, for use with digital computers.* Providence: Brown University.
4. Gaines, H. F. 1939. *Cryptanalysis.* New York: Dover.
5. Knuth, D. E. 1975. *The art of computer programming.* Reading, Massachusetts: Addison-Wesley.
6. Ohaver, M. E. 1973. *Cryptogram solving.* Columbus, Ohio: Etcetera Press.
7. Peles, S., and A. Rosenfeld. 1979. *Breaking substitution cyphers using a relaxation algorithm.* University of Maryland Technical Report TR-721. January.
8. Schatz, B. R. 1977. Automated analysis of cryptograms. *Cryptologia.* 1: 116-142.
9. Sinkov, A. 1966. *Elementary cryptanalysis.* New York: Mathematical Association of America.
10. Wiedmann, E. 1975. *APLUM reference manual.* Amherst, Massachusetts: Newell.
11. Winston, P. H. 1977. *Artificial intelligence.* Reading, Massachusetts: Addison-Wesley.