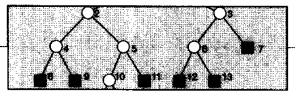


A General-Purpose CMOS Associative Processor IC and System

An associative processor architecture integrates the functionality of content-addressable memory, functional memory, and associative parallel processors in a single-chip architecture. We combined a set of 16 such chips to form the Coherent Processor, which interfaces to an IBM PS/2 computer. The processor's writable control store permits quick execution of application-specific microcoded operations.

Charles D. Stormon

Nikos B. Troullinos

Edward M. Saleh

Abhijeet V. Chavan

Mark R. Brule

Coherent Research

John V. Oldfield

Syracuse University

Association has been a long-standing alternative to addressing in computer organization, but its efficient and economic realization has been slow to materialize. Bush originated the concept in his 1945 article,[1] which anticipated many developments we now take for granted. There were flurries of interest in the 1950s and 1960s, as new memory technologies were explored,[2] but cost and, until recently, size have been formidable barriers to effective application. Unlike random-access memory, content-addressable or associative memory is complicated by word length and multiple-hit considerations. Until now, no general-purpose CAM architecture has emerged, although there are specific application niches such as memory management units for fast processors.

We broadly classify associative processors into three types. The simplest and most familiar variety is content-addressable memory. CAM acts as a directory memory: Data stored in CAM is compared in word-parallel fashion to a comparand, which may have certain bits masked. The result is a response vector that indicates whether each word in CAM matched the masked comparand. Typically, the response vector addresses a RAM, which contains the information to be looked up.

The second type of associative processor is functional memory. An FM is similar to CAM except that each cell of the FM can store a third state called "don't care."[3] When a don't-care bit is stored in a cell, it will match either a 1 or a 0 in the comparand. Each word in the FM can store a conjunction of n Boolean variables, where n is the number of bits in each word. Each variable is positional, so if it does not appear in the conjunction, a don't care is stored in that position. The desired output is stored along with the conjunction, to be read out if a match is found. We can form disjunctions by replicating the output at each term in the disjunction. In this way, we can efficiently implement a Boolean function of n variables using the FM, similar to the function of a programmable logic array, except we can selectively change the functions simply by altering the FM contents.

The final variation of associative processing is the associative parallel processor. An APP is a single-instruction, multiple-data (SIMD) parallel computer with a linear interconnection network (bus). Operations are performed on local data based on a CAM-performed selection. To implement an APP, CAM bits of a specified column must be individually writable. We can add an arithmetic logic unit to each CAM word to improve performance, but this is not essential.

The APP allows arithmetic, Boolean, and other functions to be performed on the data in CAM, in every selected word in parallel. Both operands may be stored in the CAM word, or one may be in the CAM word and one on the comparand bus. The result is stored in the corresponding

0272-1732/92/1200-0068$03.00 © 1992 IEEE

CAM word. Because of the limitation of the linear interconnection network, the APP cannot perform functions in parallel when the operands are in different CAM words (such as in summation). These must be performed sequentially.

Searching tables and trees

We can easily determine whether an associative solution will be effective in a given application. If a program spends a large proportion of its time searching data structures, managing tables, following index pointers, or performing the same operations on lists or tables of data, an associative solution will probably increase its performance significantly.

CAM has traditionally been used to resolve page table references in virtual memory systems. An associative parallel processor is useful in many other areas. In RAM, we can organize data to optimize one particular access method. In an associative processor, we can organize data to efficiently serve many access methods simultaneously. Consider the problem of storing and operating on binary trees. To store any part of the complete binary tree in CAM, we number the nodes according to the following rules:

- Root node is number 1.
- Left child of node number n is $2*n$.
- Right child of node number n is $2*n+1$.

Following these rules gives a unique number (address) to every node in the tree. When the user program must access the data at a particular node, it accesses the data in one operation by using the address as a search key. A modification to this technique allows more flexible access to the data in the tree. In a fixed-size word, we left-justify the node number, padding to the right with don't cares. For example, Table 1 shows how we would encode the leaf nodes in the tree shown in Figure 1 in an 8-bit field.

The addresses encoded are stored in FM. To access a specific node, we encode the node number in the same manner, left-justifying and padding with don't cares. If the node address is present in CAM, that node will match. Any node that is an ancestor or descendant of the presented node will also respond. For example, for the tree shown in Figure 1, if 2 is presented as the search key (10******), both nodes 4 (100*****) and 5 (101*****) respond. This subtree has node 4 as its root.

Finding the sibling of a particular node is also simplified. Node number 20 is encoded as (10100***). We access its sibling node 21 (10101***) by inverting the least significant bit in the node number. In RAM, this requires either a traversal of the tree or the allocation of more storage to hold pointers to each node's siblings. Graphics and image processing applications use quadtrees (trees with a branching factor of 4) quite frequently. Complex pointer schemes permit efficient access to these trees for a single purpose. By encoding the quadtree nodes (as in the example just discussed), we can perform many operations on the tree equally.

Table 1. Node numbers representing tree nodes.

Node number	Encoded
4	100*****
5	101*****
12	1100****
13	1101****
7	111*****

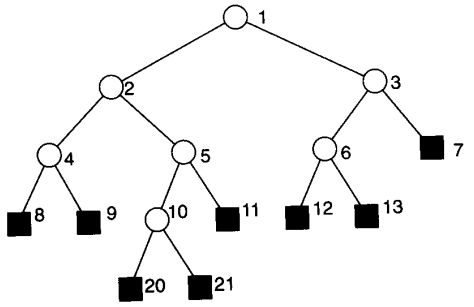


Figure 1. Binary tree.

CAM functions required

The CRC32256 device integrates the functionality of content-addressable memory, functional memory, and an associative parallel processor in a single-chip architecture with 256 processing elements, each having 32 bits of CAM and 4 bits of tag (CAM that is individually bit-writable). We developed the architecture of this device by looking at the requirements of a set of applications and selecting the functions required to support them. The principle of address independence is fundamental to associative processing. Data in the CAM array is accessed without regard to its absolute position in the array. The only position information used is the order that the priority encoder assigns to the words. If a match operation selects a set of words, the encoder provides access to the words in the same order every time.

CAM applications must perform the match operation on a runtime-selectable bit field. Bits in the search argument can be specified as don't cares. Requirements for writing into the memory fall into two categories. The processor must be able to select a subset of words in the memory and alter only those rows. Also, we must provide some number of bits in each word that can be altered without affecting the other bits in those rows.

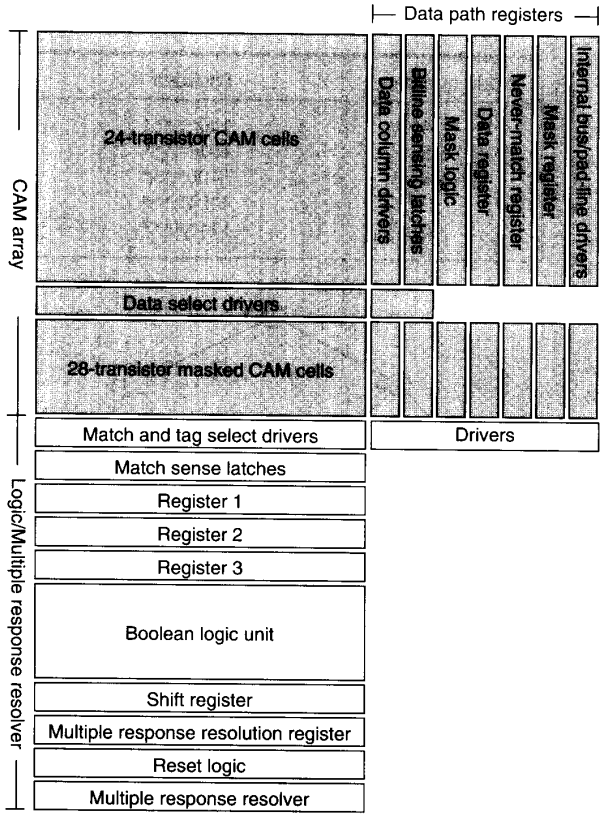


Figure 2. Top-level block diagram of the CRC32256.

The ability to store don't-care values is critical in some applications (such as those involving tree-structured data) but is not needed in all applications. Therefore, the device should store data either with or without don't-care values, and this flexibility should be provided with a minimum impact on storage capacity.

Finally, the device needs a facility for cascading the words horizontally. Some applications fit very nicely into 32-bit words, but many require some other size to store a record of data. Since we intended to integrate the CRC32256 into a system with a 32-bit bus, each memory transaction takes place 32 bits at a time. The user can group multiple 32-bit words together to vary the logical record size.

CAM architecture description

The basis of the CRC32256's VLSI implementation is a modular segment. We used this modular approach partly for the capability to migrate the design to higher densities with more advanced VLSI technologies. The CRC32256 device has a total of two segments, which together form the 256×36 array in a way that is transparent outside the device. As Figure 2 shows, a segment consists of an array of 128 words × 36-bit CAM, the data-path registers, 128 logic rows, a multiple response resolver (MRR) circuit, and all the necessary driving and sensing circuitry. The pitches of the custom-designed cells are all matched in both dimensions. Cascading of the two segments involves four separate mechanisms:

- Enabling for output the data-path drivers of only the active segment during a read operation. A segment is active if it has a word selected.
- Connecting the two halves of the first response register (register 1) through the multiword cascading logic.
- Connecting the two halves of the shift register.
- Completing the MRR tree.

Cascading of multiple devices involves the same mechanisms. When connected in an array, multiple devices form a contiguous area of CAM with associated logic and MRR. The array does not use absolute addressing. On the other hand, adjacency of given words is significant because it provides logical words whose width is a multiple of the physical word width.

The parallel data-path section consists of the data, mask, and "never-match" registers; the read sense amplifier; and the mask-generation logic. All these registers reside on a separate internal data bus that can be selectively coupled to the device data pins. When the mask-enable signal is active, the contents of the data, mask, and never-match registers generate the search argument. This way, the device can store a masking combination, and we can freely intersperse operations that use it with operations that don't, without having to reestablish the masking combination.

Each of the 256 words in the chip has an associated logic row. Two words of CAM are combined and provide three match-result signals and accept two word-select signals. The logic array is organized as a SIMD processor. As Figure 3 shows, each logic row consists of several different registers, switches, and combinational logic blocks. Specifically, a single logic row consists of five registers, three switches, two buffering blocks, two combinational logic blocks, and one Boolean logic unit. We can select any one of the three response registers (registers 1, 2, and 3) to store a match result. The MRR register provides the input to the first stage of the MRR tree. Responses from a match operation must be moved into the MRR register to perform the MRR function on those results. The shift register can be loaded from the MRR register

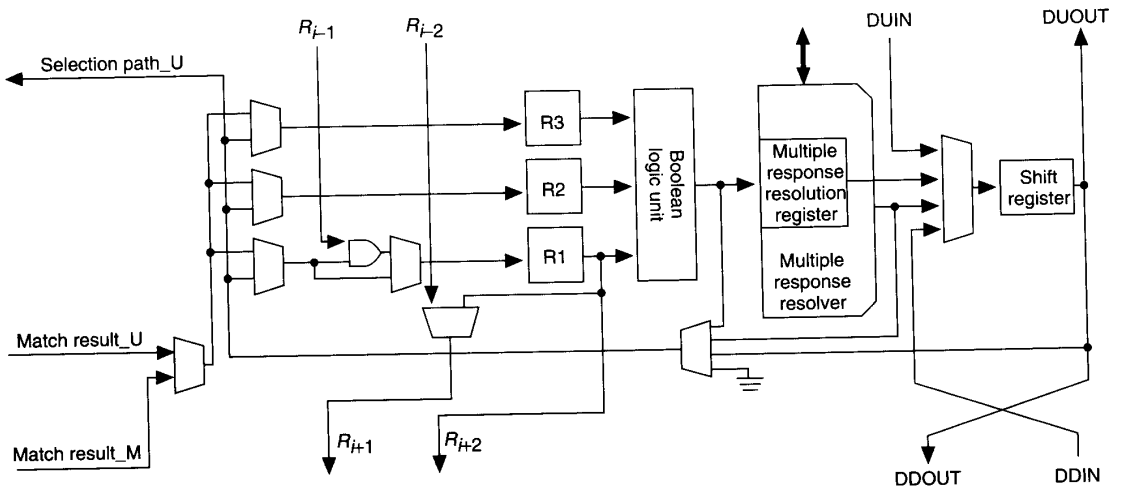


Figure 3. Organization of the logic row.

or from the output of the MRR function. Also, during a shift operation, this register is loaded with the value from the shift register of the word above (shift down) or below (shift up).

Register 1 must be selected as the response register when performing multiple-word matches. As we said, data records can span multiple CAM words. When the application program searches the records, it matches the first word of the comparand against the first word of the CAM records. It then matches subsequent words of the comparand against the corresponding fields within the CAM. As the multiword record matches with a multiword CAM entry, the response in register 1 is propagated through the physical words of the CAM entry. For example, if a data record is four CAM words long, four matches would be performed (one match, followed by three "match-next" operations). Those records that matched would have a response in register 1 that corresponds to the fourth word. The signals R_{i-1}, R_{i-2}, R_{i+1}, and R_{i+2} provide the necessary communication between adjacent CAM words for multiword response capability. Bit-mode operations use R_{i-1} and R_{i+1}, and quad-mode operations use all four signals.

The MRR function allows the topmost response in the CAM array to be selected for reading or writing. The output of the MRR register from a given row will be 1 if there is a response in that row. Through the MRR function, these outputs combine to generate a response signal for the segment, and then for the entire device. When the device is accessed for reading or writing through the MRR, the enable signal first passes to only one segment, depending on where the topmost response is in the linear array. This enable signal then passes back through the MRR tree of that segment in a mutually exclusive fashion, until it ultimately arrives at the topmost row of the segment. This topmost row will be the only row in the array to generate an MRR output signal, and thus it will be the only row accessed during the read or write operation.

Further, when a "select-next" operation executes, the row that has MRR output set will reset the contents of its MRR register to 0. Now when the MRR function is evaluated, the enable signal will be sent to the next row that has a response. If this row is in the next segment (or device, for an array of chips), the cascading logic transparently steers the enable signal. All this happens in a time proportional to the logarithm of the array size, which for reasonable array sizes is one clock period.

The Boolean logic unit can perform one of 256 logic functions to registers 1, 2, and 3, and their inverted outputs. The Boolean logic unit output can be stored in the MRR register or registers 1, 2, or 3; or it can be used as the select path signal. This block provides the interconnection path for transferring data between registers, as well as the means to modify these registers' contents. In addition, the use of the Boolean logic unit output as the select path signal allows access to a CAM word based on any combination of the response registers.

The select path is the signal that accesses the CAM word. This signal is enabled when a CAM word is read or written. A word can be selected via one of four paths: MRR output, Boolean logic unit output, the shift register, or an unconditional selection of all words.

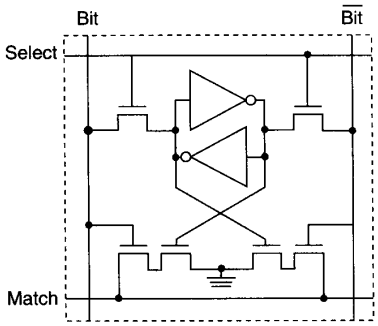


Figure 4. Static CAM cell.

Table 2. Encoding scheme for quad values in CAM.

1	0	Don't care	Never match
11	00	10	01

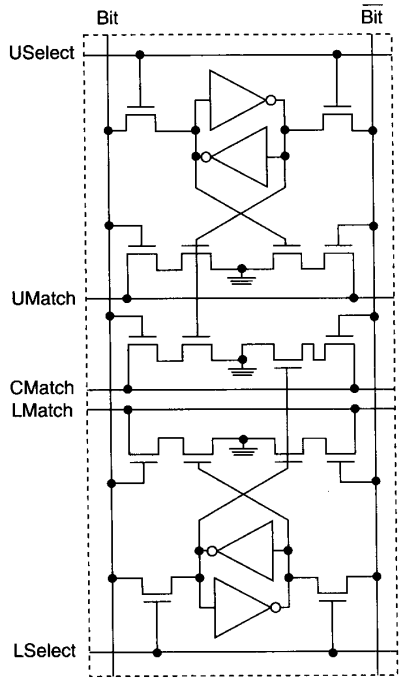


Figure 5. Static CAM cell for quad word storage.

Circuit specifics

In the CAM array design, our first concerns were simplicity and performance. We extensively simulated and verified the CAM cell, the basic logic row, and the MRR block in small test chips. The basis of the CAM cell design is the 10-transistor static cell shown in Figure 4. We verified this building block in a test structure of 32×32 cells with driving, sensing, and scan-path test circuitry.

To store don't-care and never-match states, we combined two CAM cells vertically with a third match line and associated comparator. (Matching against a don't-care state will always yield a response, whereas matching against a never-match state will never yield a response.) The don't-care state presents a double 0 to the inputs of the middle match-line comparator; the never-match state presents a double 1 to these inputs (Table 2). Figure 5 shows this extended CAM cell architecture. The four transistors between the two 10-transistor cells implement the comparator for the middle match line. A row of these new 24-transistor cells provides storage for two binary words or for a single quad word, depending on which match line is used.

The 24-transistor twin cell is 44×98 microns, using 2-micron MOSIS rules. It is 20 percent taller than a pair of 10-transistor cells stacked vertically. By providing the capability to use the upper and lower halves separately when operating in a binary mode, we maintained a good balance between functionality and chip area. Also, the decision to view the storage area as consisting of bits or quads alternatively made the overall chip design simpler: We passed the details of quad reading and writing to the user. The tag bits provide bitwise selective writing. A vertical select line in each column controls two extra select transistors.

The CAM array's parallel nature presents special difficulties in achieving robust write and match operations.[4] With multiple write, the pull-ups in each cell present a resistive load to the column drivers. The strength of the pull-ups, the vertical size of the array, and the strength of the drivers have to be carefully balanced, given power limitations and acceptable noise levels. For match operations, all match lines must be precharged, and then single-rail sensing must occur in the presence of noise from the search argument broadcasting drivers. These energy-intensive operations have no counterparts in static RAM design, and we had to respect the dynamic power limits of a CMOS device.

We used Mead and Wawrzynek's complementary set-reset logic (CSRL) design principle[5] extensively in the on-chip registers. The data-path registers—including bit-line sense amplifier, match sense amplifiers, response registers, and shift register circuits—are variations of the basic CSRL design. We used CSRL because it combines static retention, fast-sensing action, and low power in one compact circuit.

A variation, which we believe is original, is the implementation of a single-phase clock master-slave structure using

dual CSRL latches cascaded in series (a modified N stage followed by its dual P stage). Figure 6 shows this version of CSRL, where the first stage, the master, is a memory cell with differential inputs In and $\overline{\text{In}}$ that sense when the clock input is high and hold when it is low. The second stage, the slave, has the output and complementary output of the master as its differential inputs. This stage senses when the clock input is low and holds when it is high. The outputs of the slave stage, Q and $\overline{\text{Q}}$, are the actual outputs of the register. Mead and Wawrzynek provide further details on CSRL.[5]

The MRR is one of the more complex circuits in the chip (see Figure 7). When a search of the CAM array results in more than one match, the MRR allows these matching entries to be accessed in a prioritized fashion. The MRR circuit has a tree structure to ensure fast operation. Its speed is critical, because this circuit must combine results from all rows in the array (feedforward path) and then steer the enable signal to the topmost row (feedback path). To achieve maximum speed while using minimum space, we used pseudo domino logic in this block. The feedforward logic uses modified domino logic,[6] while the feedback logic uses full domino logic.[7] It is very important to avoid charge redistribution problems when adopting this combination of logic families.

Chip performance

The complete chip in 2-micron CMOS combines 256 rows, each with 36 bits of CAM, and the row logic pipeline. Control is largely external to allow overlapping of control sequences and broadcast of these sequences to multiple chips. In the worst case, control signals and data must be valid for at least 20 ns, which translates to a maximum microcode clock rate of 50 MHz. Complete operations take from two to five control words and may be partially overlapped. Table 3 (next page) gives typical operation times. The CSRL sensing latches used as sense amplifiers typically exhibit a 15-ns sensing time for the single-ended match lines and 8 ns for the complementary bit lines.

The 9.2×9.2-mm die size includes pads. The chip contains approximately 200,000 transistors and is packaged in a 108-pin ceramic pin-grid array. As with all full CMOS devices, the power dissipation depends only on the rate of switching. Despite the high energy requirements of the parallel operations, the average power consumption remains below 400 mW, even at the maximum clocking rate.

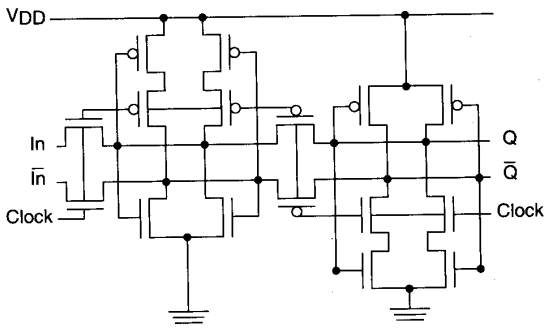


Figure 6. Master-slave CSRL latch with one clock.

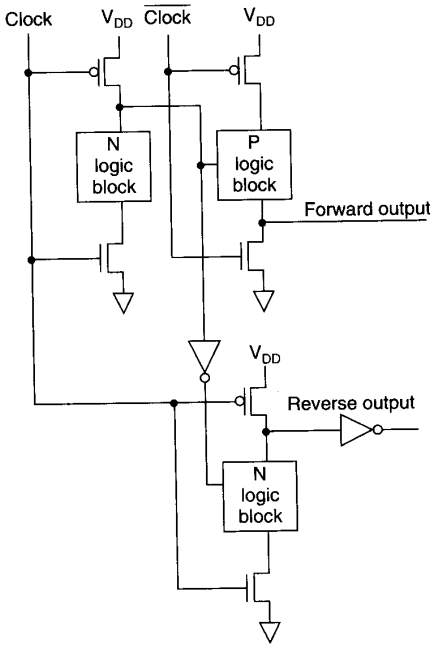


Figure 7. Organization of a single MRR block.

System hardware design and environment

Since the CRC32256 is a microcode-controlled device, it can work with a variety of system architectures. Design possibilities range from tightly coupled memory subsystems to loosely coupled processing subsystems. The first such integration of the CRC32256 is a microchannel memory device, known as the Coherent Processor.

We designed the Coherent Processor to resemble system memory; thus the interface is a 32-bit slave type. Accesses to the Coherent Processor are made through function calls, which translate to memory-mapped reads and writes. These memory accesses are decoded to control a microcode sequencer, which passes prestored control sequences to the CRC32256 pins. The Coherent Processor has an array of 16 CRC32256 components, totalling 4,096 associative processing elements. Re-

Table 3. Operation times.

Operation	Clocks	Maximum time (ns)
CAM write	1	20
CAM match	2	40
CAM read	3	60
Shift	2	40
Select next	4	80

turned values from read functions are either contents from a specific CAM word or flags signifying the Coherent Processor array's status. Write functions provide a 32-bit input value, which is stored in a specific CAM word, matched against the CAM, or used as control data. Runtime access to the Coherent Processor causes the microcode sequencer to present control words to the array until it encounters a stop bit in the microcode. At the end of a sequence, control returns to the host CPU, along with any requested return value.

Four Xilinx field-programmable gate arrays provide decoding, registers, and alternative data and control paths. Under the runtime environment, the microcode sequencer controls presentation of the microcode to the array. Using the single-step function, it may present microcode one word at a time from the system bus. This lets us scrutinize the performance of the CRC32256 in a multichip configuration. The microcode presented to the array by the sequencing engine is retained in a writable control store. High-speed (25-ns) static RAM modules implement the 16,000×64-word store. Its size permits many sequences, and thus it can accommodate a broad range of application-specific macros for the Coherent Processor. The user program loads appropriate microcode into the writable control store during initialization of the Coherent Processor. Providing the correct offset address into the store executes the individual sequences.

System software design and environment

To make application development easy, we designed the Coherent Processor software environment to be simple, understandable, and powerful. The Coherent Processor development system, shown in Figure 8, consists of an assembler and linker for writing programs, as well as a software simulator and source-level debugger. The assembler reads files containing statements in the Coherent Processor macro assembly language. The language specifies memory and logic operations using an assignment syntax. For example, the instruction

```
R2 = Match(0,1000,Mask,Bit)
```

specifies a match operation matching the 32-bit value 0 with the contents of the data part of the CAM array, and the binary value 1000 with the four tag bits. It also specifies that masking should be enabled, that the match is performed in bit mode, and that the results are to be stored in register 2.

The assembler lets the programmer write custom Coherent Processor operations. For example, the following instruction specifies that the contents of the first row, where both registers 1 and 2 contain a 1, are to be read:

```
GetR1AndR2:
        MRReg = R1 ^ R2,
        *0 = CAM[MRROut].
```

This GetR1AndR2 operation stores the logical AND of registers 1 and 2 in the MRR register. Next, multiple responses are resolved, and the word of memory indicated by the topmost responder (CAM[MRROut]) is read out. The result is placed in *0, which indicates that it should be placed in position 0 in a user data structure. This operation is called from a C program by a procedure named callCP, as follows:

```
callCP(GetR1AndR2,mydata).
```

These callCP procedure calls are placed in a C program compiled and linked to a special Coherent Processor library. The result of the read operation is placed into mydata[0], where we assume that mydata is an array name. The programmer specifies the index (0) in the macro assembly code. The assembler's output is a file included in the user's C program and contains a data array defining the contents of the writable control store. A set of definitions lets the code reference operation names (for example, GetR1AndR2).

Applications

In many applications, the Coherent Processor can provide a cost-effective performance increase. Traditional applications of CAM have been in virtual-memory-translation look-aside buffers and local area network routers. The Coherent Processor's increased functionality opens up a host of additional applications.

CAM application. When searching text, we frequently need to find a short string within a longer string. This operation is called a substring search. The Coherent Processor can store text by putting four 8-bit ASCII characters into each 32-bit CAM word. To find any occurrence of a substring, we present that substring as a search pattern four times, because the substring could begin in any of the four characters stored in a word. For example:

```
match ("A B C D")
match ("* A B C")
matchnext ("D * * *")
match ("* * A B")
```

```
matchnext ("C D * *")
match ("* * * A")
matchnext ("B C D *")
```

Using this algorithm, the substring search takes the same number of operations as the number of characters in the short string. The search is independent of the long string's size, provided that the long string fits in CAM. We can extend this method to situations where the search string is "fuzzy." We use don't cares for parts of the search string that are not completely specified and perform multiple searches.

FM applications. Several functional memory applications experience increased performance.

Quadtree Manhattan rectangle generation. In quadtree software, it is usual to apply a recursive divide-and-conquer algorithm to generate the quadrants for a given Manhattan rectangle. But if we use a quadtree variant of the scheme illustrated for Figure 1—that is, with four children per parent, and storing the quadtree leaves in FM—the order is immaterial. Thus we can generate the quadrants directly with a "covering sequence" method,[8] which covers the x- and y-coordinate ranges by minimum sequences and then forms their Cartesian products. For example, the integer range 3 through 9 is covered by the sequence 0011, 01**, and 100*. This method is much faster, particularly for small rectangles, and also permits nonstrict quadrants whose sides are $2^{**}m \times 2^{**}n$.

Region growing. Image processing often requires grouping together contiguous regions of pixels with the same or a similar color. The algorithm for grouping is called region growing, and it can benefit greatly from the use of functional memory.[9,10]

The first step is to encode the image with a quadtree representation.[8] Next, we search the image for the quadrants with the desired color and store the set of responses in register 2. Picking one of those quadrants, we label it region 1 and remove this quadrant from the list in register 2. Picking the next quadrant from register 2, we see if it is a nearest neighbor (in image space) of the region 1 quadrant (four CAM searches). If it is, we mark it region 1; otherwise, we mark it region 2 and remove it from the list in register 2, and so on.

Generally, for each quadrant that has the correct color, we search for all neighboring quadrants of the same color. If there is none, we name the current quadrant a new region. If there is at least one match, but none already labeled, we name them all a new region. If there is at least one match and at least one is already labeled, we rename the current quadrant to the region named by the first responder, and rename all connected regions to that same region name.

The remaining set of regions represents the set of contiguous coherent areas in the image. The computation time is proportional to the number of quadrants that have the correct color.

Pattern and symbol recognition. This application compares a library of patterns against an incoming pattern. Each library element is constructed over a training period by overlaying two or more example patterns and making them don't-care bits where they differ. Thus, each library element is a template with which to match incoming patterns. By carefully selecting the categories and the examples to be combined, we can use each library pattern to recognize a class of input patterns. A symbol is a special case of a pattern, which is built of pixels that represent a symbol to a human observer. With the library of templates stored in FM, we can quickly categorize an incoming pattern. The incoming pattern is the comparand, and any template that matches it indicates the pattern category to which it belongs. The trouble with this method is that it is very sensitive to noise and distortion (rotation, scale, translation, and so on).

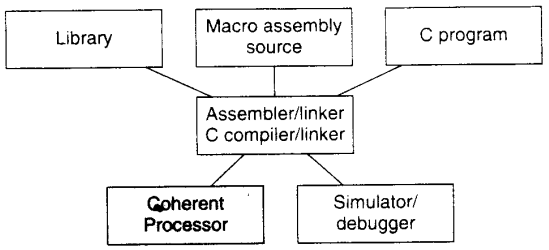


Figure 8. Coherent Processor development system.

We can easily overcome scale and translation distortion by preprocessing the pattern. Correct rotation can often be deduced by contextual clues. If not, either applying a rotation-invariant transform (for example, the Hough transform) or presenting the pattern in a variety of rotations should be effective. We can eliminate random noise by applying the region-growing algorithm just reviewed and eliminating any regions less than a certain size. The quadtree representation will also reduce the amount of FM needed for patterns with a good deal of coherency. Provided that the library fits in the FM, the pattern-recognition step requires only a single associative search (not including whatever preprocessing is necessary).

Prolog accelerator. An experimental Prolog compiler for the CP[11] uses the CAM to provide parallel backtracking and to improve the efficiency of the well-known Warren Abstract Machine (WAM).[12] The compiler stores instantiations (values bound to a variable) for variables in CAM and retrieves them by presenting the variable name as the search key. This approach eliminates the overhead of creating space for unbound variables, dereferencing argument registers for these variables, and trailing the various bindings of one variable to another. In this model, the CAM stores the Prolog terms and the global stack. An abstract instruction set, based on this CAM storage model, is similar to but simpler than the WAM, simplifying

both the compiler and the runtime system. In addition, operations on lists, garbage collection, and the occur check are faster and more efficient because we store variables and terms in CAM.

An earlier development of an experimental Prolog interpreter explored how other aspects of Prolog execution can be accelerated using CAM.[13] In addition to the stack management used in the compiler, the interpreter also explores clause filtering, and unification of lists and other data structures.

Clause filtering is an indexing technique that uses CAM to store a superimposed code word for the head of each clause in the Prolog program. When the current goal is to be executed, the interpreter computes and presents its code word to the CAM as a search key. Only clauses that match the functor, arity, and any instantiated variables are candidates for unification. This technique reduced the number of unnecessary unifications by a factor of 2 to 20, depending on the program run.

By representing list structures in CAM with an efficient tree representation, we can also reduce the time to unify one structure with another. The Prolog interpreter's unification algorithm reduces the complexity of comparing lists to, at worst, the number of variables and ground terms in the larger list. Conventional structure unification requires the algorithm to traverse each list sequentially. The more complex the nested list structure, the better the CAM algorithm performs by comparison.

Parameter window addressable memory. We can implement a broad set of applications efficiently using a functional memory combined with the covering-sequence algorithm (see the section on the quadtree Manhattan rectangle algorithm) for computing the set of 1,0,* patterns required to represent a range of integers. A parameter window is a region in an n-dimensional space represented by n parameters and the set of values these parameters may assume within the window. If we restrict the set of values to a single contiguous range of values, the parameter window will be rectangular. We can represent more complex parameter windows by combining a set of rectangular windows. We represent a parameter window in functional memory by finding the set of rectangles that covers the window, finding a covering sequence for each parameter for each rectangle, forming the Cartesian product of the covering sequences, appending the name of the window, and storing the resulting words in FM with each parameter in a corresponding positional field. Given that, finding whether a point is in one or more parameter windows requires a single search.

Determining which parameter windows cross another selected window amounts to computing the covering sequence and presenting it as a sequence of comparands. The search time is independent of the number of windows—a very useful property because this problem tends to be exponential in the number of dimensions when performed sequentially.

Rule-base accelerator. Kogge et al.[14] have experimented with CAM support for production systems and found it to provide up to two orders of magnitude performance improvement over conventional approaches. OPS5, a forward-chaining inference system, scans facts to determine which rules' conditions are satisfied. From this set, OPS5 selects a single rule and modifies the facts according to that rule. The cycle then repeats.

By compiling OPS5 programs into a structure known as a Rete net, conventional implementations make this comparison as efficient as possible. Using a CAM to store the components of the Rete net further increases the efficiency because a single CAM search can identify all rules whose conditions may be satisfied by a given fact, or conversely all facts that fit the conditions of a given rule. In addition, since facts in OPS5 frequently need to be created or destroyed, the functional memory can identify all instances of a fact and free them immediately rather than traverse the network looking for instances of a fact that needs to be deleted.

APP applications. Two applications are particularly interesting.

Neural network simulation. When processing the inputs to each layer of a neural network, the system multiplies each input by a weight specific to that connection, sums the weighted inputs to each neuron, and compares them to a threshold. If we represent each connection and its associated weight in the Coherent Processor, the system can fan out each input as needed by multiplying all the connection weights in parallel.

The summing and thresholding steps are combined so that we check whether any weighted input alone is enough to exceed the threshold. If not, we subtract the largest weighted input from the threshold and try again. We continue until the neuron fires or we exhaust the nonzero weighted inputs. This is usually much faster than computing the summation, particularly because the Coherent Processor performs the searches in parallel.

Sparse matrix computations. Just as in the neural network application, we do not need to spend any time on matrix terms whose values are zero or some other default value. The Coherent Processor's CAM stores the nonzero entries in the matrix, along with their row and column numbers. The zero entries are simply not stored. Now we can perform any operation on a row or column in parallel. Likewise, we can arrange particular patterns of parallel processing (such as multiply every other entry in every other row by 2).

We move data in the matrix by rewriting the row and column identifiers to appropriate new values. Naturally, searching the data becomes a unit operation. This technique is useful in Gaussian elimination, Fourier transforms, discrete set operations, and virtually any other matrix operation in which the data is sparse—because no time is spent on zero entries.

WE ARE WORKING ON SEVERAL IMPROVEMENTS in the system design. First, bit-selective update and the storage of don't cares are so essential that we will optimize future designs for these capabilities. A promising approach would be to combine a complementary dynamic CAM cell like that of Sodini and Wade[15] (in which the storage of don't cares comes naturally because of the lack of feedback) with a two-dimensional selection mechanism.

We can easily achieve higher density through smaller geometry fabrication techniques. Scaling for smaller geometries is straightforward, given the hierarchical design. At 0.8 micron, a device of 1,000 words (eight segments, 36 Kbits total) would require a die size of 7×7 mm. By generating more of the control internally (and trading some performance), we can reduce the pin count to 68.

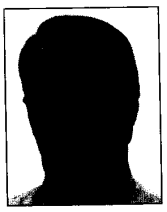
We are also designing a second-generation Coherent Processor board-level architecture to make the Coherent Processor more autonomous. Adding direct memory access capability is one way we will achieve this goal. Also, we want to decouple the execution sequences from the host processor using a scheme in which the host CPU passes a command to the Coherent Processor, and the Coherent Processor signals the host when it has completed a function. Together these efforts will result in an efficient single-board system with 64,000 processing elements and 256,000 bytes of CAM.

Acknowledgments

We thank the Syracuse University CASE Center, the IBM Federal Systems Division, and the USC/ISI MOSIS Service. Particular thanks are due to P.D. Kogge, J.A. Coleman, J.C.R. Ribeiro, K.A. Greene, G.C. Stiles, D. Elmendorf, J.A. Robinson, R.D. Williams, N.E. Wiseman, J.D. Kim, K.E. Twardowski, K. Mcveary, T. Park, S. Ramirez, and H. Krempel.

References

1. V. Bush, "As We May Think," *Atlantic Monthly*, Vol. 176, July 1945, pp. 101-108.
2. T. Kohonen, *Content-Addressable Memories*, 2nd ed., Springer-Verlag, Berlin, 1987.
3. T. Kohonen, *Associative Memory*, Springer-Verlag, Berlin, 1977.
4. N. Troullinos and C. Storman, "Design Issues in Static Content-Addressable Memory," Case Center Tech. Report 9208, Syracuse, Univ., Syracuse, N.Y., Aug. 1991.
5. C. Mead and J. Wawrzynek, "A New Discipline for CMOS Design: An Architecture for Sound Synthesis," *Proc. Chapel Hill Conf. Very Large Scale Integration*, Computer Science Press, Rockville, Md., 1985, pp. 87-94.
6. V. Friedman and S. Liu, "Dynamic Logic CMOS Circuits," *IEEE J. Solid-State Circuits*, Vol. SC-19, No. 2, Apr. 1984, pp. 263-266.
7. R.H. Krambeck, C.M. Lee, and H.F. Law, "High Speed Compact Circuits with CMOS," *IEEE J. Solid-State Circuits*, June 1982.
8. J.V. Oldfield et al., "Content-Addressable Memories for Storing and Processing Recursively Subdivided Images and Trees," *Electronics Letters*, Vol. 23, No. 6, 1987, pp. 262-263.
9. R.D. Williams, *Quadtree Operations for Content-Addressable Memories*, doctoral dissertation, Univ. of Cambridge, Cambridge, UK, 1988.
10. W.E. Snyder et al., "Content-Addressable Read/Write Memories for Image Analysis," *IEEE Trans. Computers*, Vol. C-31, No. 10, Oct. 1982.
11. H. Bacha, "Beyond the WAM, a PAM for the CAM," Case Center Tech. Report 8816, Syracuse Univ., Syracuse, N.Y., Dec. 1988.
12. D.H.D. Warren, "An Abstract Prolog Instruction Set," Tech. Note 309, SRI Int'l, Menlo Park, Calif., 1983.
13. C.D. Stormon et al., "An Architecture Based on Content Addressable Memory for the Rapid Execution of Prolog," *Proc. Fifth Int'l Conf. and Symp. Logic Programming*, 1988, pp. 1448-1473.
14. P.M. Kogge et al., "VLSI and Rule-Based Systems," in *VLSI for Artificial Intelligence*, J.B. Delgado-Frias and W.R. Moore, eds., Kluwer, Boston, 1989, pp. 95-108.
15. C.G. Sodini and J.P. Wade, "Dynamic Cross-Coupled Bitline Content Addressable Memory Cell for High Density Arrays," *IEEE J. Solid-State Circuits*, Vol. SC-22, Feb. 1987, pp. 119-121.



Charles D. Stormon is CEO and chief scientist of Coherent Research. While pursuing a PhD in computer engineering, he founded the company in 1987 with the assistance of the New York State Center for Advanced Technology in Computer Applications and Software Engineering (the CASE Center) at Syracuse University. His technical interests lie in computer systems engineering, neural networks, parallel processing, and knowledge-based systems.

Stormon received his BS and MS degrees in computer engineering from Syracuse University. He has won multiple Small Business Innovation Research (SBIR) awards. His affiliations include an adjunct faculty position at Syracuse University's College of Engineering, membership in the IEEE and the Computer Society, board membership in the Northeast Parallel Architectures Center and the Technology Development Organization of Central New York, and a manuscript reviewer's position for *IEEE Transactions on Computers*.

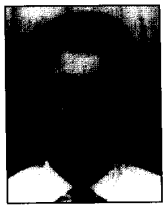
Nikos B. Troullinos is a senior design engineer at Coherent Research, working on new-generation CAM devices. He was involved with the development of the associative processing architecture presented in this article, focusing on circuit and physical design. He is also a PhD candidate in

the Department of Computer and Information Science at Syracuse University, where he works on reduction schemes for the lambda calculus as they apply to architectures targeted to declarative languages.

Troullinos received his electrical engineer's diploma from the Aristotle University of Thessaloniki, Greece, and his MS degree in computer engineering from Syracuse University. He was awarded a Fulbright Fellowship. He is a member of the IEEE Computer Society and the ACM.

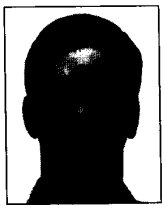
Edward M. Saleh is a computer engineer at Coherent Research involved in the design of associative processor VLSI and system architectures. He has worked as a computer engineer for General Electric, where he participated in the design and development of the Warp Systolic Array Processor. His professional interests include parallel processing computer architectures, specifically SIMD machines. His master's work was on fault tolerance in parallel processing architectures.

Saleh received his BS in electrical engineering from the State University of New York at Buffalo and his MS in computer engineering from Syracuse University. He is also a graduate of the General Electric Edison Engineering Program and the General Electric Advanced Course in Computers.



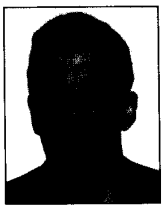
Abhijeet V. Chavan is an advanced project engineer with the Delco Electronics division of General Motors. When this article was written, he was a design engineer with Coherent Research. He has also worked for Siemens AG as a design engineer for industrial automation systems and as a research assistant at the CASE Center at Syracuse University. His research interests include systems engineering, VLSI design, VHDL, and object-oriented programming.

Chavan received his BS degree in electrical engineering in 1983 and his MS in computer engineering in 1988.



Mark R. Brule is vice president of engineering at Coherent Research. He currently works with image processing and pattern recognition applications using CAM technologies. Previously, he worked for Genigraphics as a software engineer. His master's work concerned approaches to logic programming execution using content-addressable memory.

Brule received his BS degree in computer science and his MS degree in systems and information science from Syracuse University. He is currently finishing his MS in mathematics at Syracuse University.



John V. Oldfield, a professor in the Department of Electrical and Computer Engineering at Syracuse University, presently is on leave as a visiting professor at Tektronix. In 1983 he started the Syracuse University Machines for Associative Computation (SUMAC) project, under the aegis of the CASE Center at Syracuse University. He has held academic and research positions at Bruce Peebles, Ltd., Edinburgh; the University of Edinburgh; University College Swansea, and the University of Wales. His research interests include content-addressable memories, with applications in computer graphics and data and image compression; field-programmable gate array architectures; and self-timed systems.

Oldfield received his BSc and PhD degrees from the University of London. He is a senior member of the IEEE and a member of the IEEE Computer Society.

Direct questions concerning this article to Edward M. Saleh, Coherent Research, Inc., 1 Adler Drive, East Syracuse, NY 13057, or via e-mail at saleh@cri.com.