MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Laboratory for Computer Science

Computation Structures Group Memo 165

A Structure Processing Facility for Data Flow Computers

by

William B. Ackerman

(To be published in the Proceedings of the 1978 International Conference on Parallel Processing.)

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract N00014-75-C-0661.

July 1978

A STRUCTURE PROCESSING FACILITY FOR DATA FLOW COMPUTERS[a]

William B. Ackerman
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract -- A data structure processing facility for data flow computers is proposed. It supports dynamic creation, allocation, and garbage collection of arrays and general acyclic trees. The structures are processed in a completely "pure" manner, that is, free of side effects, which requires careful design of the controller. The structures are stored in a memory that uses "packet communication" at its interface and can handle many concurrent transactions. The overall system is designed to use distributed processing throughout, and hence an arbitrarily high throughput rate can be achieved with a sufficient number of components of a given speed.

Introduction

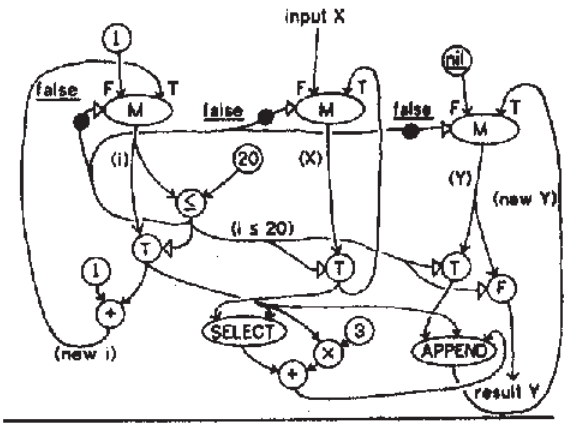
Data flow computers of the Dennis-Misunas type [4] have been proposed as a means of achieving extremely high processing rates through concurrent execution of many instructions. The execution model of this type of computer is a directed graph in which the nodes represent instructions and the arcs represent data paths which may contain "tokens". An instruction at the program level "fires" when its necessary data values are available and its destinations (the instructions to which it will send its results) are ready to receive data. Equivalently, it fires when its incoming arcs have tokens and its outgoing arcs are empty. It consumes the input tokens, computes its particular operation on the input data, and emits result tokens on its output arcs.

There are many useful computational problems for which such a computer requires a data structure processing facility. A contrived example program, showing data structure operations, is illustrated in Figure 1. It takes as input an array (vector) X with indices 1 to 20 inclusive, and returns a similar array Y computed according to the rule $Y[i] = X[i] + (3 \times i)$ for $1 \leq i \leq 20$. The SELECT operator reads the element from the array given on its left input at the index indicated by its right input. APPEND places the data from its third input on the array given as its first input at the index given as its second input.

A few instructions behave slightly differently from the rules given above. The constant operators (those with no inputs) fire when their destinations are ready, always producing the indicated constant. When the "T gate" fires, it produces a copy of the data from its top (data) input if the value on its side (boolean) input is true, and consumes both input tokens. If the boolean value is false, it consumes both input tokens but produces no output. The "F gate" acts in a complementary fashion. The "M gate" fires when it has an

(a) This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract no. N00014-75-C-0661.

Figure 1. A data flow graph

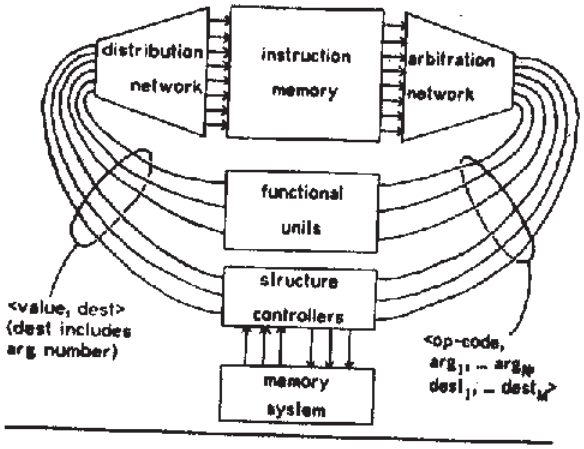


input token present on its boolean input and on whichever data input is labeled by that boolean value. It consumes those two input tokens and produces as its output a copy of the value present on the selected data input. A token is not required on the unselected data input, and if one is present it is not consumed.

The program is stored in the instruction memory of the data flow computer. (See Figure 2.) Each instruction cell contains one elementary instruction along with the addresses of the cells in which its destinations are stored. An instruction cell has a register to hold each input argument for that cell's instruction. When those registers are filled, the cell fires, emitting an "operation packet" containing the input data value(s), the instruction operation code, and the destination cell addresses. The packets so generated by all of the instruction cells are passed through a very high bandwidth arbitration network to a collection of functional units. When a functional unit receives such a packet, it performs the indicated operation and emits a "result packet" for each of the indicated destinations. These packets contain the data value and destination cell address (including the destination argument number). They are passed through the distribution network back to the instruction cells, where they are stored in the appropriate argument registers.

The implementations of the SELECT and APPEND instructions are the subject of this paper. APPEND returns an output array equivalent to the input array except that the new data exists at the given index, replacing whatever may have been there previously. It must be emphasized that the update takes place without side effects. From the program's point of view, no existing array is ever modified; rather, new and different arrays are constructed. (In the example program, the array being constructed was passed from the

Figure 2. Data flow computer organization



APPEND operator back to itself as each item was added. It was not simply "written on".)

The requirement that structure operations be free of side effects places a burden on the structure controller, but it is a nearly unavoidable consequence of the nature of the data flow computer's method of computation. The data flow computer's ability to concurrently process operations, without making the programming task intolerably complex, relies on complete noninterference of different parts of the program. If side effects were permitted, either the computer would exhibit serious and uncontrollable nondeterminacy, or a mechanism would have to be provided whereby the programmer could explicitly specify instruction ordering constraints other than the constraints arising naturally from data dependency. Such a mechanism would make the computer nearly impossible to program efficiently. The data flow computer's freedom from side effects has significant influence on the nature of the programming languages that are suitable for data flow. Data flow languages, whether high level or low level, should be "value oriented" rather than "object oriented". While value oriented languages may at first seem nonintuitive to programmers accustomed to handling arrays in object oriented languages such as Fortran or PL/I, they appear to be well suited to clean program design, and they make sequencing and concurrency problems trivial.

In the following sections, we will describe the binary tree model used for storage of data structures, and how the "purity" criterion is met. This will be followed by an overall description of the structure controller and memory, showing the interfaces among the various modules. The behavior of the structure controller will be outlined, followed by a discussion of the mechanisms for making a high performance memory system.

Tree Structures

The structure facility's fundamental data type is acyclic trees of fixed order, that is, fixed number of branches emanating from each node. With this base structure, it is possible to handle a very general class of data structures, of which arrays are a particular case. Arrays are stored as trees with the array elements at the leaves. Array indices are partitioned into the selectors necessary to specify the appropriate branch at each level. For example, with trees of order 16, an array of size 4096 could be stored in a tree of depth 3. The SELECT and APPEND instructions would divide each 12 bit index into 3 fields of 4 bits each. Each field, beginning with the most significant, would be used to select the correct outgoing branch at each node of the tree.

For concreteness and simplicity, the order of the trees will be fixed at two, that is, they are equivalent to the binary trees of LISP [7]. This assumption implies that large arrays must be stored as trees of considerable depth, but it simplifies the discussion. The trade-off is between the necessity for many memory references to access an array element if the order is small, and unnecessary data transmission and wasted space due to excessively large memory blocks if the order is large. Future simulations of data flow computers should help to decide the optimum order.

The "selector" arguments to the SELECT and APPEND instructions are bit strings. The structure controller traverses the given structure, under control of the selector bits, to find the node to be read or written. The selector bits are read from left to right, that is, the leftmost bit corresponds to the root of the tree. For efficiency, it may be assumed that these bit strings are elementary data types, and that the computer's instruction set allows efficient transformation of array indices (integers) into the appropriate bit strings.

The totality of the data values handled by the computer are divided into two classes: the elementary values of all types (equivalent to LISP atoms), and structures. A structure is the concatenation of two data values, whether they be elementary values or structures. There is a special elementary value nil which is used to represent "nonexistent" substructures. It is the only elementary value specifically recognized by the structure controller – all others are treated as uninterpreted data.

Each node of a structure is stored in one word of the memory system, which has a unique address (pointer). All structures and substructures are represented by these pointers: nodes in the memory consist of pairs of items which are either elementary values or pointers, and data tokens in the computer are also elementary values or pointers.

The two machine level operations performed by the structure controller may now be defined:

SELECT[structure, selector] → value
returns the value at the point in the structure indicated by the selector. It is an error if the structure controller attempts to "run past" an elementary value other than nil, that is, if there is such an elementary value in the given structure whose selector is a proper prefix of the given selector. If the structure controller attempts to run past the value nil, it simply returns nil as the result.

APPEND[structure, selector, value] → structure
returns a structure which is identical to the given one, except that it contains the given value at the position indicated by the given selector. Whatever value was previously at that position is absent in the result. Any elementary value in the original structure at a position whose selector is a prefix of the given selector will also be absent in the result.

Nodes and branches are added during an APPEND as necessary to create a place for the item to be appended. They are removed as necessary to avoid any substructures whose only leaf nodes are nil, as shown in Figure 3. This behavior of the APPEND operation gives nil two very useful properties, which make explicit "CREATE" and "DELETE" operations unnecessary:

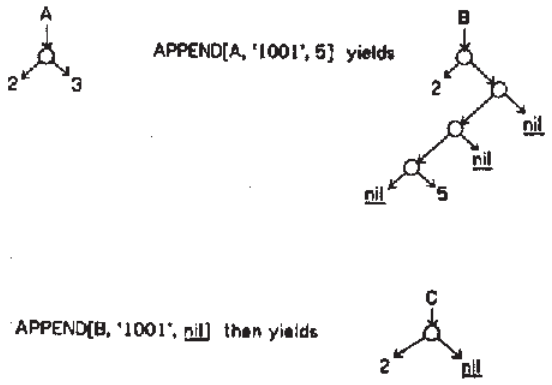
1) To delete part of a structure, simply APPEND nil in its place.

2) To create a structure, simply use nil as the initial structure, and APPEND things to it.

Reference Counts, Sharing, and Copying

As explained above, the controller must handle structures in a manner that is free of side effects. The meaning of a data structure must be independent of any conceptual "global state" of the memory, even though the structures are stored in a global memory. This requires very careful design of the structure controller, and proving the controller's design correct is an intriguing problem.

Figure 3. Behavior of append operation



One solution to the problem would be to forbid any sharing or overlapping of structures. Since every structure would have its own private area of memory, there would be no side effects. However, this would probably be prohibitively expensive. It would require each structure to be completely copied whenever its value is duplicated. Instead, the solution used here is to copy nodes when they are to be written on and there are other pointers to the same node. The determination of whether there are other pointers to a node is made by examining the reference count associated with the node.

Each node of a structure has a reference count, which is the total number of pointers to that node from all sources: other nodes and tokens in the computer. All operations that create or destroy pointers must alter the reference count. For example, when a true or false operator destroys a token, the count must be decreased. When a SELECT is performed, the count of the result node must be increased, and the count of the original structure must be decreased, to account for the input token that was destroyed. Similar care must be taken in accounting for pointers that are copied from one structure to another or destroyed within a structure by being over-written.

The structure controller meets the criterion of avoiding side effects by never writing on a node whose reference count is greater than one. If the reference count is greater than one, a pointer to the node exists elsewhere, and modification of the node would alter the structure represented by that other pointer. In this case, the structure controller copies the node onto a freshly created node, whose reference count is set to one. If the original node's reference count is one, the structure controller knows that it has exclusive access to the node, and can write on it directly.

The cell allocation and management algorithm (see [6] section 2.3.5) is as follows: Free cells are kept on a free storage list. When a new cell is needed for creation of a structure node, one is removed from this list, and its reference count is set to one. Whenever a reference count is decreased and the result is zero, that cell is reclaimed, that is, returned to the free storage list. When this happens, the pointers contained in the cell are destroyed, so the reference counts of the nodes pointed to must be decreased. If either or both of those counts go to zero, those cells are reclaimed, and the process repeats.

The reference count method of cell management is known to work if no directed cycles ever exist in any structure. (If directed cycles can exist, such a cycle could be "abandoned" by destroying all pointers to it from the rest of the computation, but it would never be reclaimed because each node would have a reference count of one.)

Absence of directed cycles can be shown to be a consequence of the method for avoiding side effects. This is because whenever a cell would be written on to make a cycle, its reference count would have to be at least two: one for the process doing the writing and one for the structure that would form the cycle.

The effect of the structure controller's copying behavior, and its avoidance of circular lists, is illustrated in Figures 4 and 5. The numbers appearing inside nodes are reference counts. If A and B denote pointers into the structure of Figure 4 (ignoring the dotted line), a naive execution of APPEND[A, '000', B] might replace the pointer to "1" with a pointer back to B, as shown by the dotted line. This is incorrect. The correct result of APPEND[A, '000', B] is shown in Figure 5.

Because reference counts must be updated whenever structure-valued tokens are duplicated or destroyed, and only the structure controller can manipulate reference counts, the structure controller must perform all instructions that duplicate or destroy structure tokens. In particular, the T and F conditional gates cause tokens to be conditionally destroyed. These instructions can be handled by very simple functional units if the type of the data token is elementary, but if it is a structure, the instruction must be processed by the structure controller to decrease the reference count.

The Controller's Interface

The structure controller is one of several identical units connected to the rest of the computer through the arbitration and distribution networks, to the memory through the interconnection network, and to each other through the UID network, as shown in Figure 6. (UID stands for unique identifier.)

Figure 4. Structure prior to APPEND

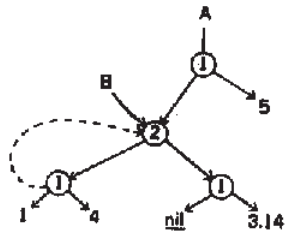


Figure 5. Structure after APPEND

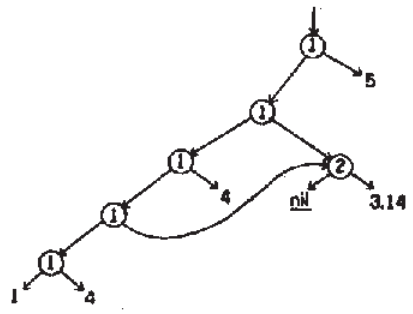
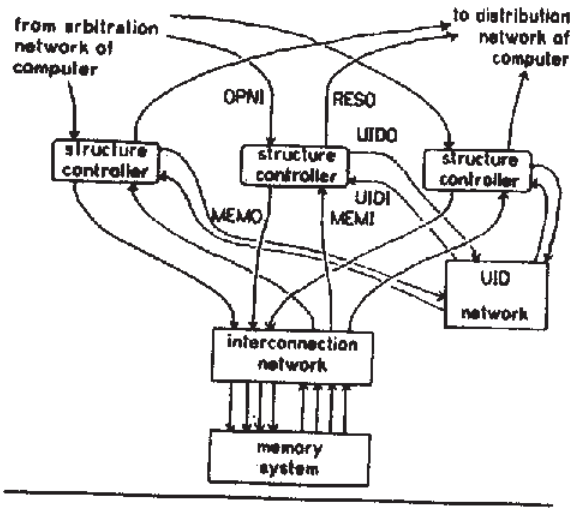


Figure 6. Interconnection of controllers and memory



The reason for using several controllers is that the processing rate for any one controller is limited by the technology used in its communication ports and other circuitry. A major design goal of the data flow computer is that its overall processing rate not be limited by the speed of its individual components. This requires careful design to eliminate "bottlenecks". There are multiple functional units for other instructions (e.g. arithmetic) for the same reason. The arbitration network sends each operation packet to any functional unit or structure controller that is of the right type and is not busy.

All communication is through asynchronous unidirectional transmission of packets [3]. Every packet transmission link has a link in the opposite direction carrying acknowledgment signals. The packet communication concept imposes no timing restrictions on packet transmissions except that a system will eventually transmit its required output. This freedom from global timing considerations is very important in large systems that perform many processing operations concurrently.

The ports of the structure controller have the following functions:

OPNI - "operation in" - receives operation packets from the computer. These packets are either SELECT(struct, sel, $dest_1$, ... $dest_N$) or APPEND(struct, sel, new-val, $dest_1$, ... $dest_N$).

RESO - "result out" - transmits results of structure operations back to the computer, one packet for each destination that was given in the operation packet. These packets are RESULT(value, dest).

MEMO - "memory out" - transmits commands through the interconnection network to the memory. These packets are named FET ("fetch") or UPD ("update").

MEMI – "memory in" – receives replies from FET commands. These packets are named RTR ("retrieve"). The interconnection network routes them from the memory back to whichever structure controller sent the command.

UIDI – "unique ID in" – receives UID packets, giving fresh cells to be used for the creation of new nodes. These packets are UID(addr).

UIDO – "unique ID out" – transmits UID packets. The UID network simply takes UID packets according to the various structure controllers' abilities and redistributes them according to the controllers' needs.

The Memory System Behavior

The principal design criterion for the memory is that it be able to handle a great number of overlapped random access transactions to achieve a high data rate. A packet memory satisfies this need through its ability to treat a request for data and the return of the requested data as completely distinct events on the interface, possibly separated by many other unrelated requests and data transmissions. Unlike conventional "crossbar switch" networks between processors and memory units, it is not necessary to allocate a path through a packet switching network for the duration of a memory cycle. The command and response packets pass through the interconnection network independently.

From the memory's point of view, each word contains a data field and a reference count field; the subdivision of the data field into pointers and indicator bits is of no concern to the memory.

Since result packets from the memory are separate from the command packets, some means of associating a result packet with its originating command is necessary. This is done with a "tag" field in the command and result packets. The memory returns the contents of the tag field of each command packet in the tag field of the corresponding result packet. The interconnection network uses the tag fields to transmit result packets from the memory back to the correct structure controller, and each controller uses the tag fields to match result packets with the correct structure operations, if more than one operation is in progress at one time.

The basic memory operations are reading with optional increase or decrease of reference count, and writing. The commands are:

FET(addr, tag) – reads the addressed word and returns its data and reference count in a packet RTR(addr, data, refct, tag). The returned tag is the same as the tag in the command.

FET$^+$(addr, tag) – same as FET, but increases the reference count and returns its new value. The returned packet is RTR$^+$(addr, data, refct, tag).

FET$^-$(addr, tag) – same as FET$^+$, but decreases the reference count instead. The returned packet is RTR$^-$(addr, data, refct, tag).

UPD(addr, data, refct) – writes the data and reference count on the addressed word. This generates no reply packet, and hence uses no tag field.

The UID Network and Free Storage

The controller expects UID packets to be provided in an "unending" supply at port UIDI. When it needs a fresh cell, the controller simply takes a packet and acknowledges it. Upon receiving the acknowledge, the UID network provides another packet. The structure controller must constantly provide fresh cells to the network through port UIDO. Whenever the UID network acknowledges the last packet from UIDO and the controller's free storage list is not empty, the controller takes a cell from the list and transmits it at UIDO.

Whenever the structure controller reduces the reference count of a cell to zero, it reclaims that cell by placing it on the free storage list. It must also decrease the reference counts of the cells pointed to by the pointers in the reclaimed cell. This may cause one or both of those cells to be reclaimed, so the procedure is recursive. Since each reclamation can cause two others, the recursion is difficult to handle. (The structure controller has no stack memory.) The procedure used is only to reduce the reference count of the node pointed to by the right half of the word at the time it is reclaimed. While the word is on the free storage list, its left half is preserved, and its right half is a pointer to the next cell on the list. This makes reclamation of the cell give rise to only one other reclamation instead of two, so the recursion can be handled iteratively without a stack.

When the structure controller receives a cell at UIDI, the left half still contains a pointer to a node that must have its reference count reduced. The structure controller then reduces that count before using the cell.

If no packet is available at UIDI when one is needed, the computer has presumably run out of memory. This is an unpleasant situation to deal with, since its occurrence is nondeterminate (one run of a program might succeed while another run fails). The simplest thing to do is to terminate the computation. There is a chance that, by simply waiting, a free cell might be created by another part of the program, allowing the computation to proceed. However, this strategy gives no positive indication that a computation has failed other than the fact that it stops executing instructions, which may be undesirable.

Initialization of the Free Storage Lists

Before starting program execution in a data flow computer, all of memory must be put into the free storage lists. These lists are initialized by dividing up the total memory space into as many parts as there are controllers, and linking all words in each part into a chain. The head of each

chain is pointed to by a register of its structure controller. The right half of the data in each word contains the address of the next word in the chain, and the right half of the last word contains an elementary value to mark the end. Since the left half of each word contains a value whose reference count is to be reduced when that word is taken from the list, the left halves must be initialized to some harmless elementary value such as nil. As the final initialization step, each controller must then behave as though it had received an acknowledge on port UIDO, that is, it must take a word from the list and send it out through UIDO.

General Design of the Controller

The controller's algorithm will be given here only in broad outline. A detailed algorithm may be found in [2].

Each controller can handle some fixed number of concurrent operations. Each of these operations requires a number of private variables that must be stored in the controller's local memory. When an operation begins, it is given a "tag", or index, which is used to identify the private variables and memory transactions.

The total number of controllers, and the number of concurrent operations that each controller is designed to handle, is a complex decision based on the speed of the memory system, the speed of the controllers' logic and internal memory, the speed of the switching networks and communication ports, and the required performance of the system. In general, a controller should handle enough concurrent operations to keep its internal logic busy processing the required memory transactions.

When a structure operation is received at OPNI, an unused tag is selected for it. (If there are no unused tags, the controller does not accept packets at OPNI.) The operation then proceeds concurrently with all the others, with all of its memory transactions labeled with its tag, and all of its state information stored in the controller's internal memory. When the structure operation is complete, the result packets are sent out and its tag becomes free.

The "SELECT" and "APPEND" Algorithms

The SELECT operation is straightforward. The controller scans the selector string from left to right. For each bit, it reads the current node (starting from the given structure) by issuing a FET command and waiting for the RTR packet. It then picks out the appropriate half of the data, depending on the selector bit, and uses this as the pointer to the next node. When finished, the current node is the answer. It increases its reference count with a FET^+/RTR^+ transaction, and then decreases the reference count of the original structure with a FET^-/RTR^- transaction.

The APPEND operation is much more complex. It is designed to require only a single pass down through the structure, creating new cells only when the reference count forbids writing on existing cells. As soon as a node is passed whose reference count is greater than one, all subsequent nodes must be copied.

A special case arises if the value to be appended is nil. The structure controller must take care to remove any structure that would contain nothing but nil as its terminal nodes. It does this by remembering the last place in the structure that contained data other than nil. When it reaches the end, the superfluous chain of nil's can be destroyed.

Structure of the Memory System

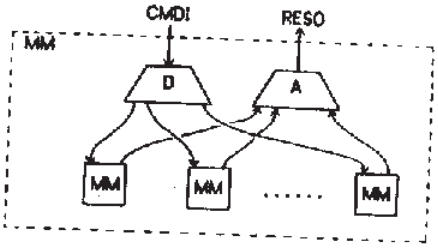
The memory system is designed in terms of an abstract memory module called "MM" that obeys a formally defined specification [1]. MM has an input port called CMDI (command in) and an output port called RESO (result out).

MM can exist in many forms. It can be constructed directly out of conventional memory circuits. A large MM can be constructed out of smaller MM's and some packet routing networks, and a fast MM can be constructed from a slow MM and a cache module. These constructions will be given below. By treating MM as an abstract object which always obeys a uniform functional specification, one can construct hierarchical interconnections of devices and subsystems with complete flexibility, and use interconnection theorems for packet systems to prove that the resulting system will behave properly [5] [8].

The Horizontal Interconnection

MM modules may be connected in an arrangement similar to the "interleaving" of customary memories in order to increase transaction rate. The address space to be realized by the interconnection is divided into parts, with one module handling each part. For example, the least significant bits of the address might be used to select the module. Incoming commands are sent to a distributor which sends them out through one of several ports depending on the least significant address bits. Those bits are removed from the packet, so the individual memory modules only "see" their share of the address space. Result packets are sent to an arbitrator, which merges them into a single stream. The incoming port number is encoded into the output packets, reinstating the removed address bits. Such an interconnection is shown in Figure 7. If each of the small boxes labeled MM obeys the functional specification, the entire interconnection will also.

Figure 7. Horizontal interconnection



This connection is one of the methods by which the transaction rate can be increased. Random access memory devices usually have the property that every read or write transaction causes the device to become busy for some period of time, during which it cannot handle any other transactions. For example, a MOS RAM might be busy for 500 nanoseconds during every transaction, and therefore be able to handle 2 million transactions per second. The only way to increase the data rate is to use many memory units. If a distributor can handle 64 million packets per second on its input port, and an arbitrator can handle 64 million packets per second on its output port, it might be reasonable to use 32 MOS RAM's, each in a separate MM unit. These are connected to a 32 port distributor and a 32 port arbitrator. The average rate at which packets come out of each port of the distributor is 2 million per second, which is the rate at which individual units can handle them. Assuming the commands are uniformly distributed over the address space, this interconnection will handle 64 million transactions per second. The retrieval delay for each item will still be 500 nanoseconds, but that is an unavoidable consequence of the memory technology being used.

Vertical Interconnection (The Cache Module)

A memory module may also be connected to a cache module "CM", which then realizes an MM system with the same address space. CM sends commands to the "main memory" through port MEMO, and receives results through port MEMI, as shown in Figure 8.

In each of these interconnections, the small boxes labeled MM may be further interconnections, so arrangements such as those in Figure 9 are permissible.

The Interconnection Network

The interconnection network is somewhat similar to the horizontal interconnection described above, except that it handles packets from multiple structure controllers. Command packets are distributed to the correct memory module according to address, and the tag fields of FET/FET$^+$/FET$^-$ packets are are augmented with bits

Figure 8. Vertical interconnection

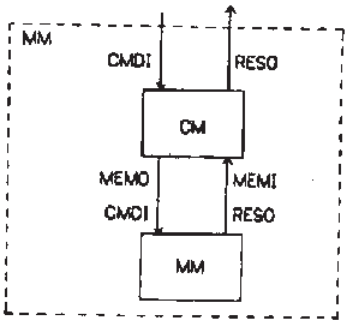
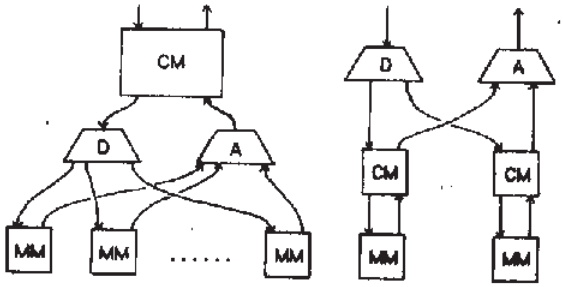


Figure 9. Memory structures



indicating which controller sent the packet. When a RTR/RTR$^+$/RTR$^-$ packet is received from the memory system, the extra bits in the tag field are removed and used to determine the controller to which the packet is to be sent.

References

[1] W. B. Ackerman, A Structure Memory for Data Flow Computers, Laboratory for Computer Science, M.I.T., TR-186, (August, 1977), 125 pp.

[2] W. B. Ackerman, A Structure Controller for Data Flow Computers, Laboratory for Computer Science, M.I.T., Computation Structures Group Memo 156, (January, 1978), 16 pp.

[3] J. B. Dennis, "Packet Communication Architecture," Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, (August, 1975), pp. 224-229.

[4] J. B. Dennis, D. P. Misunas, and C. K. C. Leung, A Highly Parallel Processor Using a Data Flow Machine Language, Laboratory for Computer Science, M.I.T., Computation Structures Group Memo 134, (January, 1977), 79 pp.

[5] D. J Ellis, Formal Specifications for Packet Communication Systems, Laboratory for Computer Science, M.I.T., TR-189, (November, 1977), 138 pp.

[6] D. E. Knuth, The Art of Computer Programming, Vol. I: Fundamental Algorithms, Addison-Wesley, (1973), 634 pp.

[7] J. McCarthy, et. al., LISP 1.5 Programmer's Manual, MIT Press, (1966), 106 pp.

[8] S. S. Patil, "Closure Properties of Interconnections of Determinate Systems," Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, (July, 1970), pp. 107-116.