1982

A multi-microcomputer intercommunication structure and multi-tasking algorithm

Barry A. Andrews
Iowa State University

Andrews, Barry A.

A MULTI-MICROCOMPUTER INTERCOMMUNICATION STRUCTURE AND MULTI-TASKING ALGORITHM

Iowa State University PH.D. 1982

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark __✓__.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages ✓
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print ✓
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Other _____

A multi-microcomputer intercommunication structure and multi-tasking algorithm

by

Barry A. Andrews

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Electrical Engineering

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1982

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

This document describes research performed for a Ph.D. in electrical engineering at Iowa State University, in the computer systems area. The main accomplishments are:

1. The invention of a new structure, which can be used as an interconnection topology for multi-microcomputer systems, and possibly for other applications.

2. The design of a microprocessor-based element which can be used in such multi-microcomputer structures, with LSI and VLSI technology.

3. The completion of a series of experiments with a multi-tasking software system, which provides some insight into the design and operation of multi-microcomputer systems.

In the past decade, the interconnection of large numbers of microprocessors to form a multi-microcomputer has become an increasingly attractive prospect. Several authors have described topologies which would permit a set of nodes, (computers or computer busses) to be linked together by means of communication channels, into a multiple instruction stream, multiple data stream, (MIMD) architecture (1,8-10,12,13,15,20).

The key is to find an efficient way to utilize identical, mass-produced microcomputers in a large system. Such a computer could solve many separate tasks simultaneously, realizing an almost linear increase in processing power as the number of nodes increases. If microprocessors can be used in this fashion, then arbitrarily large and powerful multi-microcomputers could be constructed.

Such machines could be used in conventional multiprogramming environments, to replace expensive mainframe computers in general-purpose applications. However, an even greater potential may exist in special applications which have generally required multi-million dollar supercomputers. Current supercomputers are not only expensive, they are also very heavily utilized. Many applications are not being pursued because the necessary computer time is not obtainable at a reasonable cost -- the available resources being reserved for high priority areas such as fusion research (3,7,9,17,19).

Most such applications involve complex systems of equations which can be handled by parallel/pipelined machines such as the Cray-I or CDC 205. However, in many cases it may be possible to recast such problems so that they can be broken down into a collection of tasks, which can be executed in parallel by a collection of cooperating uniprocessors.

Provided that suitable control and communication mechanisms can be devised, it seems likely that an MIMD computer, based on microcomputers, could perform such

computations. As each processor completed its assigned task, its results would be communicated to neighboring processors, so that all the results could be combined into a global solution. If such a computer could be implemented, then many costly, computationally intensive problems would be within the reach of systems costing one-tenth to one-hundredth as much as current supercomputers. The availability of large amounts of inexpensive computational power would be extremely beneficial in many research areas and applications, such as the following:

1. Seismology.
2. Cosmology.
3. Aerodynamics.
4. Meteorology.
5. Nuclear physics.
6. Signal processing.
7. Tomography.
8. Pattern analysis.
9. Artificial intelligence.

The global weather problem is a classic example. One may imagine that the earth's surface could be sub-divided into hundreds or thousands of sectors, with a separate microcomputer assigned to process the data acquired from each. A global solution of such problems would require the sharing of information across sector boundaries, so that interactions between sectors could be resolved -- for example, the effect

of a low-pressure area in Montana on wind velocity in Minnesota. Analysis of such boundary conditions requires communication facilities, which would be provided by the interconnection structure of the multi-microcomputer topology.

If such an architecture is to be used in this type of problem solving, then better performance could probably be obtained if the interconnection structure maps into the nature of the problem. For instance, Illiac IV was a two dimensional processor array, well-suited for the solution of matrix problems. Perhaps such a two dimensional structure would be appropriate for the weather problem, or other surface effects, while a one dimensional array, (chain) would be more efficient in spectrum analysis, and a three dimensional structure would be applicable to large spatial problems, such as those found in nuclear physics and cosmology.

In all cases, it is assumed that a well-designed microcomputer could be replicated and interconnected in different structures to fit various problems. In this way, a mass-produced microcomputer could drastically reduce the cost of computations, while increasing throughput. If limiting the architecture to one type of processor resulted in some processor-dependent inefficiencies, then more processors could be added to compensate, without a major redesign of the system. If regular structures proved inappropriate for some algorithms, then more processors could be added at specific critical areas, to form a structure which maps into the

problem -- again, using the same basic processor.

However, other computationally intensive problems, or applications requiring many special functions, could use structures of special purpose processors. Such microcomputers would probably be more expensive than a common, general purpose design, but many applications would justify the extra cost as a tradeoff for increased performance.

The remainder of this paper will include a short overview of multi-microcomputer systems, followed by three main sections. The first section will describe the Geode interconnection topology, which permits variations in dimensionality and processor concentration. The second section will present a structural element which can be used to construct multi-microcomputer systems like Geode, X-tree, hypercubes, trees, stars and so forth, using LSI or VLSI technology. Finally, the third main section will present a multi-tasking software package, which seems generally applicable to multi-microcomputer systems. This section will also discuss some experimental results obtained with the multi-tasking programs.

TECHNOLOGICAL OVERVIEW

This section is intended to provide an overview of multiple microcomputer systems, rather than an exhaustive review. Therefore, several worthy proposals or implementations will be discussed only in passing, or not at all. Many good reviews have been published, and the interested reader is urged to consult the bibliography at the end of this paper. The proposals and systems described here are Illiac IV (3), X-tree (8), Cm* (12), and the "Pruned Spanning-bus Hypercube," or PSBH, (20). Collectively, these four cover most of the general ideas behind multi-microcomputer systems.

Illiac IV

Illiac IV was chosen because it was one of the first computers designed to use multiple identical processing elements, organized into a physical structure which maps into the logical structure of certain problems, i.e. discrete elements. Also, Illiac IV was actually built and operated -- yielding valuable insights into the nature of large multiprocessor systems.

The Illiac IV structure was a square array of 64 elements, as shown in Figure 1. As originally proposed, it would have been implemented with four 64-node arrays organized into a 16 by 16 structure; however, only one 8 by 8 array was actually built. The edge-links shown in Figure 1 were actually interconnected in a wrap-around fashion.

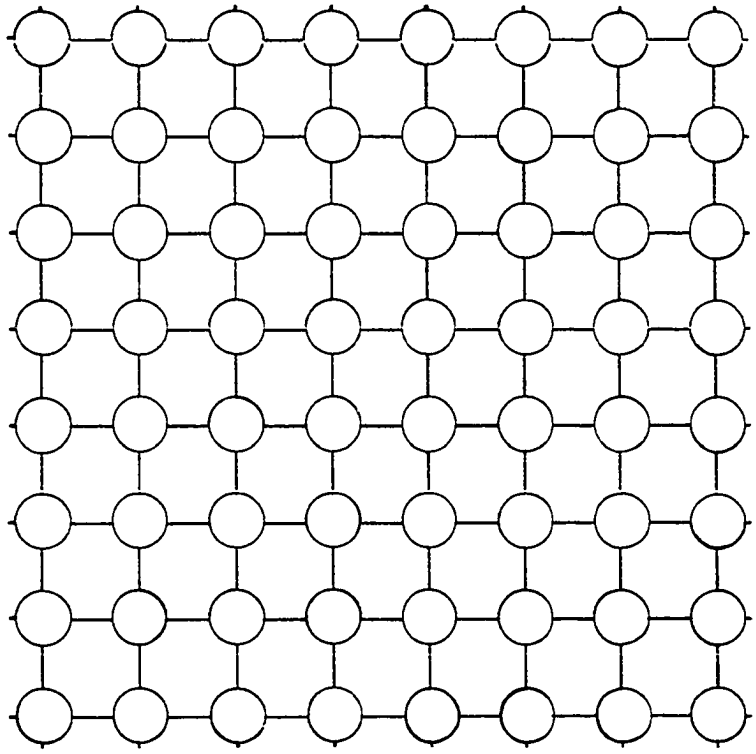

Figure 1. Illiac IV Structure

Illiac IV differs from most proposed multi-microcomputer structures in that its processing elements, (PEs) were synchronized and controlled by a single sequence of instructions. In a given instruction cycle, a PE either executes the instruction on its local data, or it remains idle. This is frequently described as a single instruction stream, multiple data stream, (SIMD) architecture.

In contrast, multi-microcomputers would likely be implemented as MIMD architectures. This could permit greater concurrency than an SIMD arrangement, since each element would operate independently, and would not need to enter idle states while other processors executed special functions.

However, an MIMD array processor would probably be more complex than a corresponding SIMD machine, for several reasons. Obviously, a separate controller would be needed for each processor. The need for a sophisticated control and communication mechanism for the entire structure would also introduce many additional problems.

On the other hand, an MIMD machine might have advantages in addition to greater concurrency. Greater flexibility would be one likely characteristic. MIMD systems would allow a set of related tasks to execute in parallel on a subset of the available processors, while independent tasks or other collections of related tasks were handled by other elements. Thus, an MIMD computer could be very useful in general purpose timesharing, batch and multi-tasking environments, giving it a greater potential for utilization, in a high-level sense.

An MIMD machine should also be somewhat cheaper to implement than an equivalent SIMD computer, using modern VLSI technology. This factor is related to flexibility and the potential for general purpose utilization. The fixed structure of an SIMD computer might limit its useful PE configurations. An MIMD element could be used alone, or in

small collections, or in many different configurations, because each would be an independent computer. This kind of flexibility could lead to much greater production volumes than for SIMD elements, resulting in reduced costs to users.

Finally, the increased complexity of the MIMD approach is not the obstacle it was in the 1960s. VLSI circuits are now being fabricated with over 100,000 transistors on a single die. Circuits with over one million elements may be practical by 1985, allowing the implementation of a 64-bit processor, a significant amount of local memory or cache, and a communication structure on a single integrated circuit.

X-tree

Most recent proposals for new computers, both uniprocessors and multiprocessors, seek to take advantage of the improvements in circuit speed and complexity offered by VLSI technology. X-tree is one such proposal which has drawn considerable interest, due to the combination of VLSI nodes into a tree-like structure. Figure 2 shows a 15-element X-tree.

The X-tree proposal calls for nodes comprised of a main processor with a collection of specialized communication processors on a shared bus, to be interconnected by means of 8-bit bidirectional communication links. The main processor would handle computation while the communication processors handled network functions like queueing and routing.

One X-tree application would have users or devices

associated with the leaf nodes, while the higher level nodes handled interaction between leaf node processes, as with shared data base operations. The unused links at each leaf node could then be used for I/O interfaces to peripherals such as terminals or disks. In this way, X-tree structures could be configured for operation in general purpose timesharing environments, or as special purpose backend processors, possibly handling relational database operations.

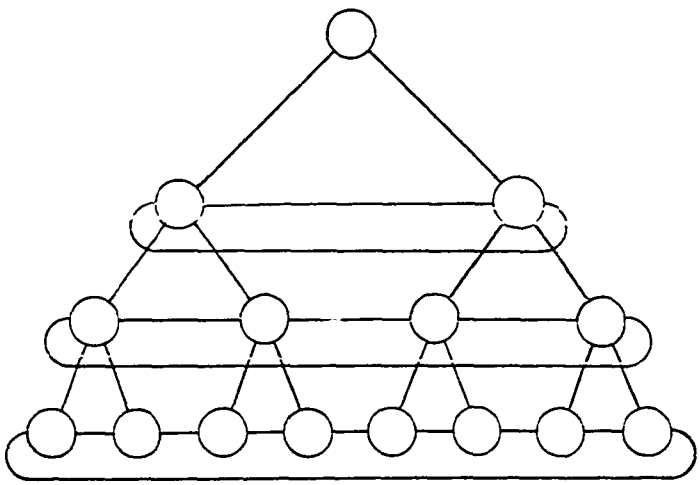

Figure 2. X-tree with 15 Elements

Pruned Spanning-bus Hypercube

The Pruned Spanning-bus Hypercube proposal, as depicted in Figure 3, has much in common with X-tree, in that VLSI microcomputers would share busses, and be interconnected to form regular structures. However, the PSBH elements would all be identical, with computation and communication tasks distributed uniformly among the nodes. The shared busses would be the only communication media present in the system, so specialized communication processors or interfaces would not be required. Thus, a typical PSBH structure could be constructed using identical VLSI elements, each with two or more bidirectional communication ports.

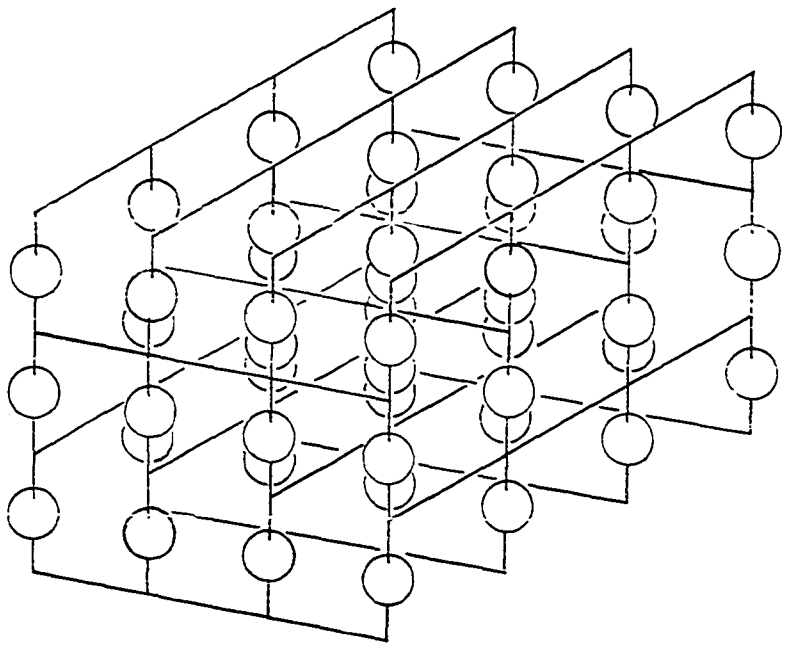

Figure 3. Pruned Spanning-bus Hypercube

Cm*

Cm* is another design which has actually been in operation for some time. Cm* uses a hierarchy of busses and memories to interconnect conventional LSI-11 and PDP-11 computers, using memory mapping techniques. Experiments with this system have shown that algorithms can be executed as parallel tasks on an MIMD machine, with an almost linear speedup as nodes are added.

Cm* has also demonstrated the value of the principle of locality to the efficient execution of cooperating tasks. Simply stated, locality in this context means that tasks which are related should be located in clustered groups of nodes, so that communication delays will have a minimal impact on the speed of computation.

However, Cm* uses several different types of bus interface and memory mapping units, so it probably would not be cost-effective when compared to X-tree, PSBH and other proposed structures which are more regular, and which call for mass-produced VLSI elements.

Remarks

Not too long ago, approaches like X-tree, Cm* and PSBH would have probably been thought foolish, because the doctrine of economy of scale would have dictated larger computers instead of more computers. That is, until fairly recently, it was more cost-effective to make or buy one large computer instead of several smaller ones. However, such a philosophy

may now have been replaced by the principle of "economy of volume". That is, computers implemented with many identical VLSI microcomputers are more cost-effective than computers built with high-speed gate-level integrated circuits, or with many diverse VLSI units.

Eventually, a limit in circuit size and density may be reached, so that VLSI circuits larger than a maximum size would not be cost-effective. The limits could be related to problems with decreasing yield, lower reliability, higher operating temperature, or simply a limit in complexity which makes circuits of too large a size inconvenient to design. Limits of this type are now being encountered with large uniprocessors. These factors should be enough to suggest that multi-microcomputer systems, like those described above, have considerable potential in applications which require large amounts of computing power.

The structures of computing elements proposed so far take two forms -- bus-oriented and link-oriented. Broadcast systems are similar to bus-oriented systems, but proposals for broadcast-oriented multi-microcomputer systems have been limited to local data networks. For present purposes, bus and broadcast systems will be considered essentially identical.

Of the three MIMD structures described above, all are bus-oriented, in that all processors are connected to a shared bus, at least within each node. In the case of X-tree, the bus is not used for extranodal communication -- with 8-bit

bidirectional links suggested for that function. Thus, X-tree seems more like a network than a multiprocessor, with the distinction less clear in the cases of PSBH and Cm*.

When information is transferred from one place to another in a computer system, it is actually being transferred from one area of memory to another, regardless of the intervening mechanisms or media. The information may be moved a word at a time, by means of random-access memory operations; however, it may also be organized into buffers, packets or messages, and transferred by means of communication channels in bit-oriented formats.

In memory systems, we think of transfers involving a certain number of address, data and control signals, which are used in each operating cycle. In the case of communication links, we usually think of a conductor by which information is transmitted and received, with address, data and control information imbedded in messages or packets, instead of being expressed as separate signals. In either case, the message is placed in a communication channel, along with enough information to describe the transfer -- perhaps its source, destination, and the number of words to be moved.

For example, let us consider a 32-bit computer connected to a bus with 32 data lines, 32 address lines, and ten control lines for bus and memory access. The bus requires 74 conductors, which must all participate in the transfer of 32 bits of information in a single cycle. Thus, a transfer of

two memory words would require the activation of 148 signals, while a 256-word transfer would need 18,944 signals.

Let us assume that the same 32-bit computer is also connected to a high-speed serial communication link, which has the same bandwidth, in bits per second, as the bus. In other words, the link could transfer 32 message bits in the time required for a single bus operation. The link would require a protocol and a message format to control the transfers; so, let us assume that 100 bits would suffice for routing and flow control fields. Then, the transfer of two memory words would require 164 signals, compared to only 148 for the bus, and the message would require over 2.5 times as long to transmit. However, a 256-word transfer would be far more efficient, requiring only 8,292 communication signals, and about 1% more time.

This example illustrates that memory bus implementations are more efficient for short transfers, while communication links are preferable for long messages. The tradeoffs are actually more complicated, both in cost and in speed. For example, it may be inappropriate to assume that a communication link could have 32 times the clock rate of a bus implemented with the same basic technology. However, the main point is that one approach is not inherently superior. The choice depends on the application, which determines the required communication bandwidth. Again, this suggests that multi-microcomputer structures should be designed to suit

particular applications, or ranges of applications, if the structures already proposed are unsuitable.

As mentioned above, communication in both bus- and link-oriented systems is basically a matter of moving information from one memory area to another. The information content of a message can be separated from the physical operations involved in a transfer. Therefore, the data structures used in the memories may be the same, regardless of whether communication links or direct transfers are involved. This is a desirable characteristic, since it permits systems to be implemented in a modular fashion.

One logical communication technique, useful for both bus- and link-oriented systems, is to set up a queuing structure in memory for each process or processor. A process or task consists of a code segment, which performs a series of operations on an input data segment, producing an output data segment. Therefore, it would seem natural to provide an input queue and an output queue for each processor. If more than one task executed on a given processor, then multiple queueing structures could be organized.

Then, communication in a multi-microcomputer system can be reduced to a set of processes, and a set of queuing operations, which represent the input and output functions of the processes. The processes could be said to be in communication when the output set of one intersects the input set of another. For example, a pipelined system would consist

of a chain of processors, linked by queues. The input queue of each element would be the output queue of its predecessor in the chain. Organizations with greater parallelism would involve processors linked by more complicated, parallel queueing structures.

So, the difference between bus- and link-oriented systems can be regarded as purely physical. The operations and data structures may be the same, but link-oriented systems have mechanisms which carry out data transfers, through some intervening medium, from the address space of one bus to another. The bus-oriented system merely has an intersection between two or more address spaces, with direct transfers.

In either case, some type of partition separates local memory areas from one another. The link-oriented system uses communication controllers, coaxial cables, and so forth. The bus-oriented system uses three-state transceivers and arbitration logic to determine which processors are connected to a given bus at a given time. The effect is to isolate processors from one another to avoid memory contention, for the purpose of efficient computation. Contention is a problem only when the processes are in communication -- when a processor or some intervening device attempts to access a memory at the same time as another processor, delaying one of them while the other completes its operation.

This is where the principle of locality plays an important role. If a multi-microcomputer is designed to

perform some large computation, then that computation will execute faster if its processors spend most of their time performing operations in their local memory areas. Accesses outside the local area imply that communication is taking place. If processors spend most of their time communicating, then they will have few cycles available for computation. In other words, if a multi-microcomputer system is to replace a large uniprocessor, then the multi-microcomputer system will be much more effective if its tasks are not communication bound.

Any factor which tends to increase the communication load on the processors, such as large messages with relatively few operations to be performed on them, or large numbers of small messages, or inefficient data transfer techniques, will reduce the computational effectiveness of the system. On the other hand, some large problems are naturally I/O bound, such as telephone switching or data acquisition. The multi-microcomputer should be as effective as a uniprocessor in such cases.

In summary, multi-microcomputer systems could function effectively in the solution of large computational problems, if suitable microcomputers could be implemented with VLSI technology. The computational problem would be divided into tasks, to keep all the processors busy. More than one problem could be handled by assigning a subset of the available processors to each collection of tasks. The system memory

should be partitioned so that processors do not slow each other through contention. A queuing structure is needed, so that the processors may communicate by chaining outputs to inputs. Finally, the amount of time a processor uses for communication limits the time it has available for computation; so, the amount of communication should be limited, and efficient methods should be used, if many processors are to be utilized effectively.

The next major section describes a new multi-microcomputer structure, which is related to X-tree and PSBH. The similarity is only general, since this structure is neither a tree or a hypercube. It is a structure based on recursion and geometric symmetry.

GEODE STRUCTURES

The structured architecture described in this section has been labelled "Geode," as a name for both an interconnection topology and a proposed computer system. The name is a combination of the words "geometry" and "node." Geodes may be regarded as data structures, formed by recursively organizing a directed graph in a symmetrical fashion. The resulting collection of nodes and links is geometrically organized into polygons, or even polyhedrons, varying with the number of communication ports available at a node.

Figure 4 shows three Geodes, each based on nodes with four ports. The single node on the left is the basic unit of implementation for all 4-port Geodes. Therefore, the position of a node in a structure is its only distinguishing feature. The three Geodes obviously differ in complexity, and it should be apparent that Geodes of higher complexity are constructed from lesser Geodes. In fact, Geodes can be recursively extended to any size. Asymmetrical structures can also be constructed, but this paper will deal primarily with symmetrical Geodes and their basic characteristics.

Properties of Geodes

The recursive, geometrical and symmetrical properties of Geodes allow arbitrarily complex multiple computer structures to be modelled, and traversed from node to node, using a very simple routing algorithm. This algorithm could be used to control communication within such structures.

Since a routing program will be presented in subsequent paragraphs, let us briefly describe the parameters required for a Geode traversal. These are:

1. Source address.
2. Destination address.
3. Number of ports per node.
4. Level of recursion (complexity).

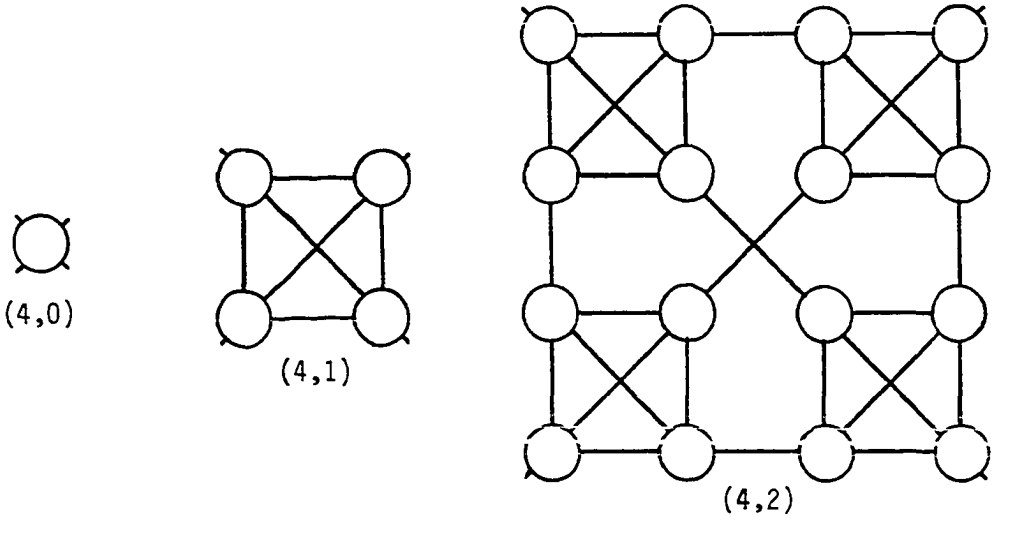

Figure 4. Three Four-port Geodes

The first two parameters -- the addresses of the source and destination nodes, use a recursively-ordered addressing scheme over the full structure. This implies that the position of a node can be inferred from its address, and that the node can use its own address to determine the addresses of other nodes connected to its communication ports.

Thus, the addressing scheme is fixed and global, providing each node and its communication ports with unique identifications, which are known to the node and its neighbors.

The third and fourth parameters are "P" and "R"; the number of communication ports per node, and the level of recursion, respectively. To simplify discussion, a tuple will be used to identify symmetrical Geodes, based on P and R. The tuple will have the form (P,R). So, a Geode with a P-value of four and an R-value of three would be called a (4,3). Figure 5 depicts Geodes with 2, 3, 4 and 5 ports per node, at various levels of recursion.

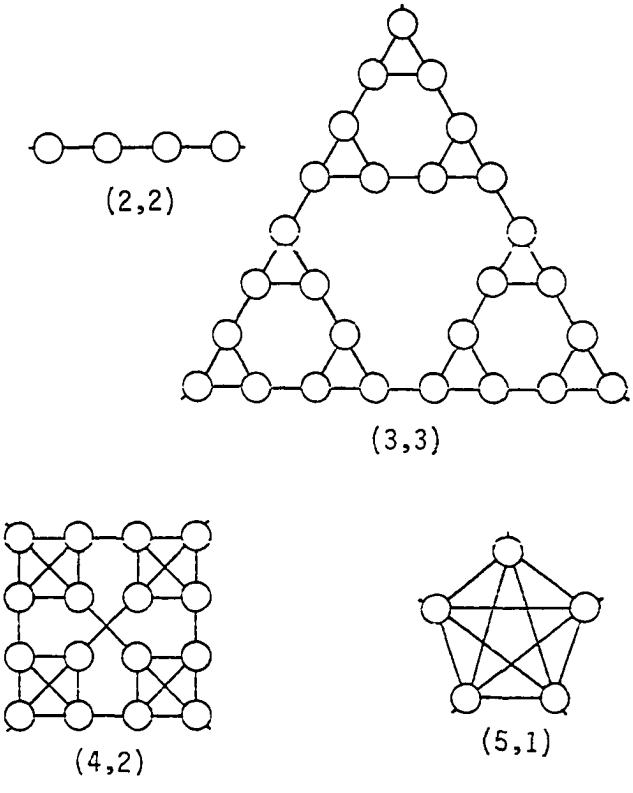

Figure 5. Various Geodes

Using a recursive definition, a Geode with an R-value of X, for X greater than zero, is comprised of "clusters," where the clusters are defined as Geodes with R-values between one and X-1. For example, a (4,1) is formed by connecting four 4-port nodes to each other. Each node is connected to P-1 neighbors, and has one communication port left over for external connections. Thus, the (4,1) has four unconnected ports, like a single node, and it can be directly substituted for any node or cluster in a four-port structure. Such substitutions can give rise to asymmetrical topologies, where some nodes are more "concentrated" than others.

Given the parameters P and R, one can construct an appropriate symmetrical Geode. As shown below, these parameters can also be used to describe several general characteristics of symmetrical Geodes:

The number of nodes (N):

$$N = P^{**}R.$$

The total number of communication links (T):

$$T = (P^{**}(R+1)+P)/2.$$

The number of internal communication links (I):

$$I = (P^{**}(R+1)-P)/2.$$

The maximum distance (M) between any two nodes:

$$M = (2^{**}R)-1.$$

Geode Addressing

The Geode addressing scheme uses base P numbers to identify the nodes, and to enumerate the communication ports. Figure 6 depicts a (4,2) with the nodes and ports fully identified. The addressing technique follows the recursive definition. Basically, nodes or clusters are aligned in geometrical patterns, and a base P number is used to identify each position in the structure. A node address has R digits, one for each level of recursion. An address digit describes the position of a node in a cluster, or the position of the cluster in the next higher-order cluster, and so forth.

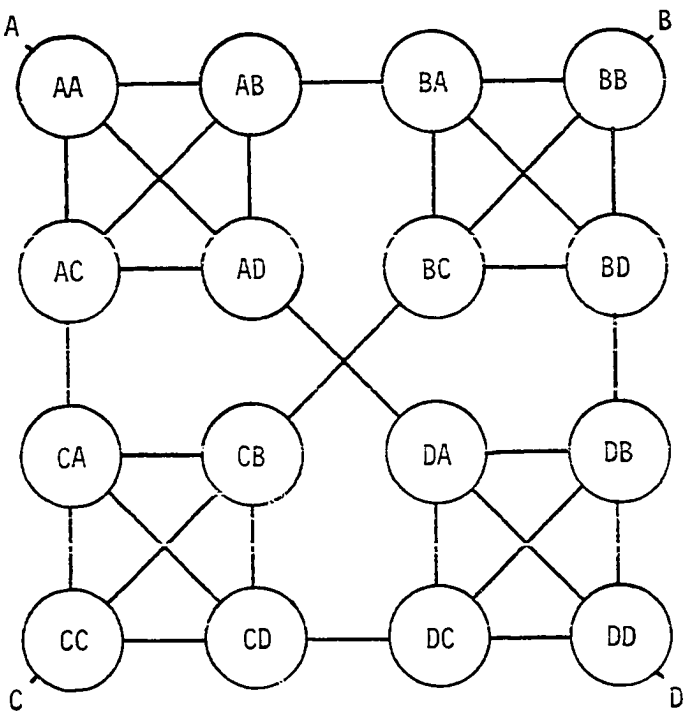

Figure 6. Geode Addressing Method

Construction of a Geode is begun by labelling the ports of each node in a consistent fashion, so that all nodes are identical. The port addressing scheme must be known to the nodes, so that a selection can be made by a routing program, which is also identical for all nodes. In the examples presented here, alphabetical characters will represent base P numbers, so that three-port Geodes would use letters A, B and C as port addresses.

The second step is to connect the ports of the nodes together, so that the 'A' port in the 'B' position connects to the 'B' port in the 'A' position, and so on. As a consequence of this strategy, the P "left-over" links of each cluster correspond to the position of the nodes where they are found. For example, the A-node has an unconnected A-port, and the C-node has an unconnected C-port. The resulting Geode is then logically equivalent to a single node. Thus, structures can be connected to each other, or they can be internally expanded to any size, with no basic changes in the address assignments of the components. As each new level of recursion is implemented, an extra digit is added to the beginning of each node address, to identify its parent cluster.

Geode Traversals

The PL/1 program TRAVEL, shown in Appendix A, is a simple program which traverses symmetrical Geodes. The source address is specified in the string SRC, and the destination address is held in DST. The variable R is the level of

recursion, and it determines the length of the address strings -- one character per level. TRAVEL will handle Geodes with any number of ports, up to the limits of the PL/I character set, as long as the characters in the SRC and DST strings represent base P digits. No tests are performed to verify the correctness of addresses.

This version of TRAVEL performs only one traversal, using the initialized values of SRC, DST and R. However, the program can be modified to perform many traversals, and to count the number of hops in each. In this way, the average distance between nodes can be computed, for any symmetrical Geode. The program could also be adapted to simulate Geode-based computer systems.

The main routine invokes the function NEW iteratively, until it returns false ('0'B). The current node (SRC) and the last output port (TCHAR) are printed at each iteration. NEW invokes PORT to get the output port ID. This character is then used in the generation of the address of the node at the other end of the output link. NEW hops from node to node in this fashion until PORT returns an exclamation point in the variable TCHAR.

PORT compares the SRC and DST strings until it finds the highest-order DST character which does not match the corresponding character in SRC. The DST character is returned as the output port ID, unless the strings match. If SRC=DST then the traversal is finished.

NEW must find the lowest-order character in SRC which does not match the port ID. It then replaces all lower-order characters in SRC, using this as a replication constant. The replication character is then replaced by the port ID. This transposition is accomplished with a concatenation operation and a built-in REPEAT function. If the lowest-order character in SRC does not match the port ID, then only this character is changed, corresponding to a hop within the same first order cluster. Otherwise, the new SRC address will represent a hop to a neighboring cluster.

PORT is the basis for "real" routing algorithms, which might be used to switch messages through Geodes. Each node would use PORT to determine the appropriate output link to any other node. It would then transmit a message, packet or other data representation through the port. The process would be repeated at each node, until the message reached its destination.

This type of fixed routing algorithm can easily be implemented in software, firmware or combinational logic. However, fixed routing schemes are not suited for fault-tolerant systems, because the entire system could be disrupted by a single node or link failure. Therefore, a more pragmatic approach might be to use the algorithm merely to initialize routing tables in each node. Failures or traffic congestion could then be handled by dynamically modifying the tables, to switch communications onto alternate paths.

Geode Performance

The average distance between nodes in a multiple computer architecture is of considerable importance if random traversals are frequently attempted. This could be the case in a general purpose system which used the communication links for interprocessor synchronization, and for access to a distributed data base.

The average path length in several symmetrical Geodes was computed, using a variation of the TRAVEL program. The results of the computations are shown in Table 1. The following information is presented:

1. Number of ports per node (P).
2. Level of recursion (R).
3. Total number of nodes per structure (N).
4. Total number of links per structure (T).
5. Maximum path length (M).
6. Ave. path length, with SRC-SRC traversals (AVE1).
7. Ave. path length, without SRC-SRC traversals (AVE2).

Two values are given for the average path length. AVE1 includes a zero-length traversal from each node in a structure to itself. AVE2 does not include such traversals, because this condition appears meaningless. However, similar path length computations have been performed for X-Tree, using zero-length traversals in the calculations of average path length. Therefore, AVE1 is used to compare Geode and X-Tree, as shown in Figure 7.

Table 1. Average Path Length

P	R	N	T	M	AVE1	AVE2
3	2	9	15	3	1.78	2.00
3	3	27	42	7	3.93	4.08
3	4	81	123	15	8.20	8.30
3	5	243	366	31	11.16	11.20
4	2	16	34	3	2.06	2.20
4	3	64	130	7	4.64	4.71
4	4	256	514	15	9.78	9.82
5	2	25	65	3	2.24	2.33
5	3	125	315	7	5.09	5.13
5	4	625	1565	15	10.78	10.80
8	2	64	260	3	2.52	2.56
8	3	512	2052	7	5.78	5.79
8	4	4096	16388	15	12.32	12.32

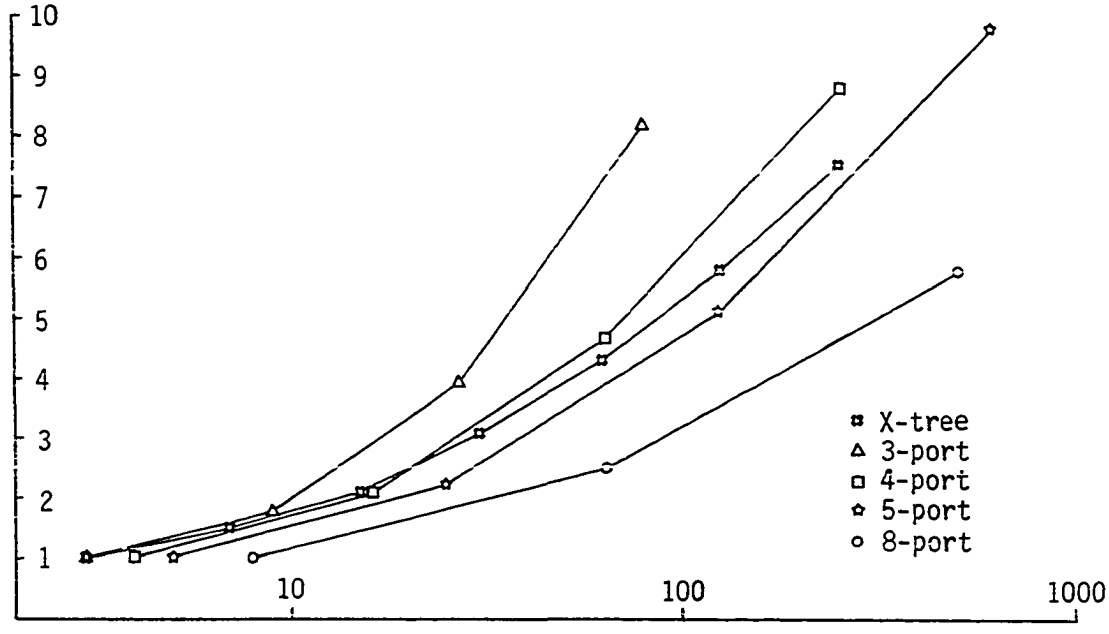

Figure 7. Average Path Length: Geode vs. X-tree

The two architectures seem very close when a fully-ringed X-Tree, with five ports per node, is compared to symmetrical four- and five-port Geodes. However, the slopes of the curves differ. It is interesting that a four-port Geode is both faster and less complex than a fully-ringed X-Tree with small numbers of nodes, ($N < 20$) while X-Tree appears to outperform five-port Geodes when large structures are considered, ($N > 1000$).

Practical Considerations

Average path length is one important consideration in the design of structured multiple computer systems. However, the difference between Geode and X-Tree is not very great, for 5-port nodes, so other factors may be more important. Structured architectures could readily be used in simulators, in general-purpose computers, and in special-purpose machines such as pattern-recognizers, relational database processors, and intelligent automata. Therefore, implementation and applications problems probably deserve some consideration.

One may assume that nodes for any structured architecture could eventually be fabricated on a single VLSI chip. Such a microcomputer could include CPU(s), memory, user-I/O and DMA-based controllers for interprocessor communications. The links could be implemented in a variety of ways, with either serial or parallel data transfers. Such choices will depend on the bandwidth requirements, and may require careful analysis of various applications.

One immediate observation is that less-complex chips will be cheaper and easier to produce. Nodes with four ports can be more readily implemented than 5-port nodes. Therefore, it would appear that four-port Geodes have a clear advantage over fully-ringed X-Trees, which require 5-port nodes.

One of four ports can be addressed with only two bits, compared to three for one of five ports. Thus, any address in a (4,4) can be represented with a single byte, allowing 256 processors to be addressed very conveniently. This factor can reduce the complexity of internal node architectures, and it can speed-up communication because the address fields in messages would be expressed more efficiently.

Also, port selection and message routing are very simple procedures for Geodes. This factor could result in reduced complexity and better performance, regardless of the number of ports or the level of recursion.

Clustering is another factor which can improve performance in the execution of concurrent tasks. Processors which are closely-connected can communicate faster than those which are far apart. Therefore, if the tasks of a job are executed on processors in the same cluster, the average communication bandwidth should improve, compared to randomly-located tasks. Geodes can easily take advantage of this principle of locality, because they are clustered by definition.

Implications

Structured architectures are one way to increase the power of computer systems at a very low cost. If all nodes in a structure are identical, at least in hardware, then a VLSI processor could be mass-produced, implementing most of the architectural features on a single chip. Structures like X-Tree or Geode could then be expanded to extraordinarily large sizes.

Many problems must be solved before structured systems become a reality. It has not been demonstrated that a large system can function without centralized control. If not, then perhaps "supernodes" should be added to the structures.

Figure 8 shows a (4,2) Geode with a central supernode, which is also implemented as a Geode. Structures of this type could be used for applications where a main task, requiring a high processor concentration, coexists with several peripheral tasks, which are less computationally intensive. Such a model corresponds roughly to the functions of the fovea and periphery of the human retina. Therefore, this type of organization might be useful in artificial vision.

This illustrates that the efficiency of various topologies could be extremely dependent on the applications. The required link bandwidth for interprocessor communications has not been established. Some of the nodes or links in a given structure may constitute bottlenecks, depending on the structure and the application. Finally, problems like task

synchronization, resource management, and interprocessor communication need much more attention. It is hoped that the unique characteristics of Geode architectures will make such problems more manageable.

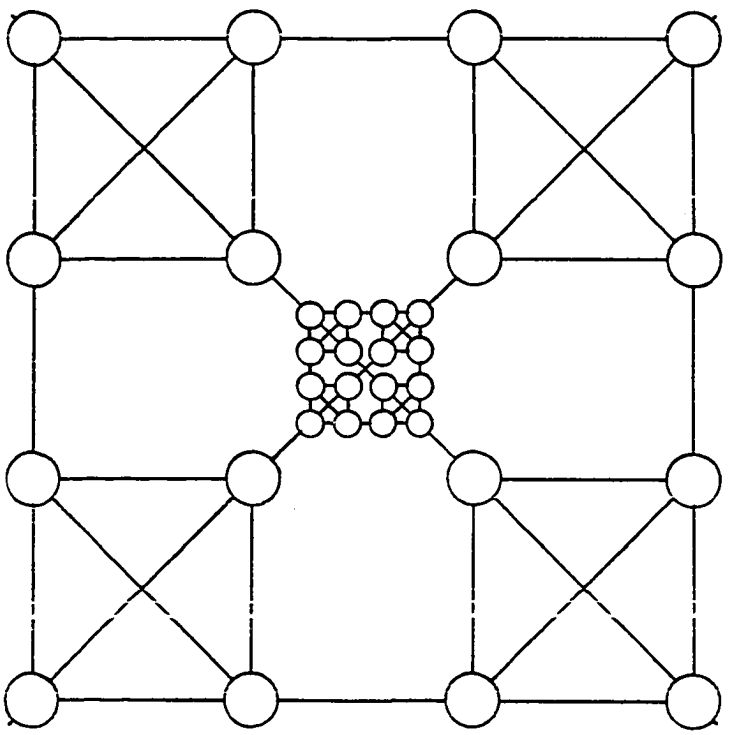

Figure 8. (4,2) Geode with Central Supernode

The next section describes a unit of implementation for structured architectures like X-tree, PSBH and Geode. This processing element, utilizing only two communication ports per processor, appears to be the "lowest common denominator" for such structures.

MP: A STRUCTURAL ELEMENT

The Geode structures described above represent only one of many classes of multi-microcomputer structures. X-tree and PSBH are two others. Such proposals are reasonably general, in that the methods for implementing and interconnecting nodes are not specified in detail.

The following paragraphs describe a design for a general purpose processing element, which can serve as the basic unit of implementaiton for the nodes and links of essentially any structure. This element, called an MP, (for multi-processor or memory-processor) uses two identical communication ports, to ease the connectivity and pin-out problems encountered in VLSI designs.

Nodes are formed by attaching one port of each MP to a local shared-bus, for intranodal communication. The remaining port of each MP is used for internodal communication. Either serial or parallel busses could be used; however, only a parallel memory-bus implementation will be described here.

Communication Bus

A bus is a collection of conductors which can be shared by a number of active devices, like computers or communication controllers. Passive devices such as memories may be connected to a bus for the use of the active devices. However, only one active device can use a bus in a given cycle. Consequently, at least two capabilities are required of each active device or processing element.

First, a method is needed for connecting and disconnecting units from a bus, in response to a request for a bus cycle. Logic gates which can be enabled and disabled, such as open collector or three-state devices, are required for such purposes.

Secondly, a method is required for recognizing the bus requests of several devices, and for granting the bus to them in a sequential order. That is, only one device can be enabled at a time. Priority arbiter circuits, like the 74148, are suitable for this role. If system memory is divided into several distinctly addressable areas, one for each bus, then address decoding logic can be used to generate the bus request signals. Figure 9 shows a generalized shared-bus organization.

Internal Architecture

The internal architecture of a processing element can be divided into several sections, as shown in Figure 10. Among these are the CPU, local memory, local I/O, and a communication structure. If the communication structure is memory mapped, then it may be viewed as a secondary or tertiary memory level. Therefore, the local memory could be separated into a cache and a working space. This would permit virtual memory techniques to be used, in conjunction with the multi-tasking principles previously described.

However, many different architectures could be implemented, depending on the nature of the problems a

particular structure is designed to solve. An internal arrangement optimized for image processing would probably not be suitable for digital filters, and vice versa. The applications, and hence the internal architectures are not of primary interest in this paper. But, we can assume that any architecture will involve, at a minimum, a processing section and a communication section. The logical operation of the latter is of primary importance here.

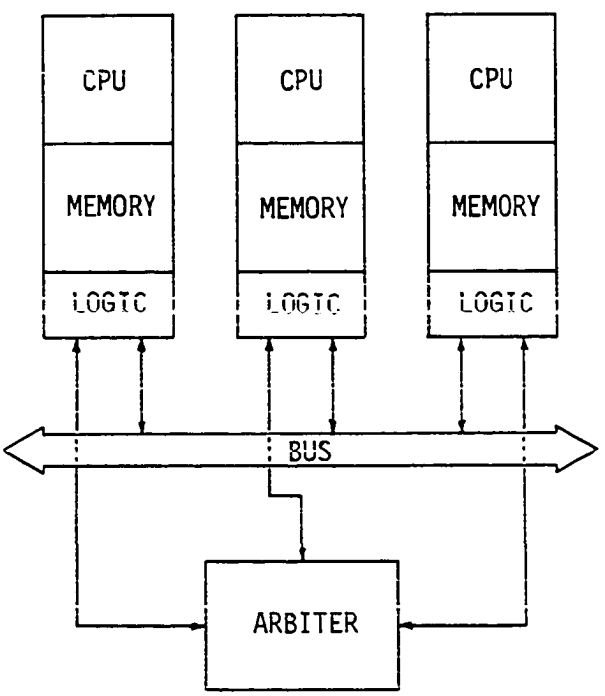

Figure 9. Generalized Shared-bus Structure

Deadlock

Aside from the problems of arbitration and connectivity, memory-oriented multi-microcomputers suffer a potential for deadlock. This is demonstrated in Figure 11, where a cycle of three processors is depicted. If each processor simultaneously requests an access to an adjoining memory, as shown by orientation of the arrows, and then waits for its request to be granted, the system will be deadlocked.

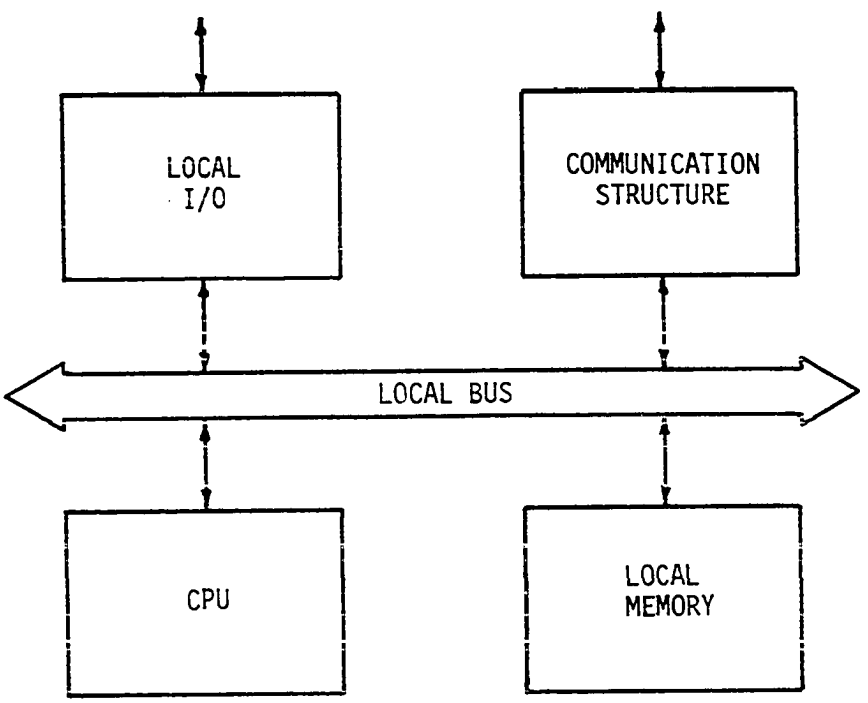

Figure 10. Internal Structure of a Processor

In issuing a memory request, a processor must activate its address, data and control lines. These lines are tied to the shared memory, which is "locked-up" while the signals are active. A processor must be disconnected from its shared memory for an adjoining processor to gain access; yet, a processor cannot yield the memory while it is "trapped" in a wait state. This paradox results in a potential for deadlock when conventional microprocessors are used as processing elements.

The problem can be solved by the addition of a partition between each processor and its shared memory areas. Such partitions are implemented like memory-bus interfaces -- using open collector or three-state devices. This allows a processor to be disconnected from its shared memory until its own external requests are granted. Thus, processors with high priorities can access the shared memories of their lower priority neighbors, if an arbiter is used to prevent ongoing cycles from being disrupted.

Conventional microprocessors would be suitable candidates for multi-microcomputer applications, if the addition of extra circuitry is acceptable. A better approach would involve new VLSI designs which have the desired characteristics incorporated into a single package.

Two such designs are diagrammed in Figures 12 and 13. Both have partitions between the processor sections and the shared memory sections. The main difference between them is

that the design in Figure 12 has two unidirectional bus-access ports, while the other has two bidirectional ports.

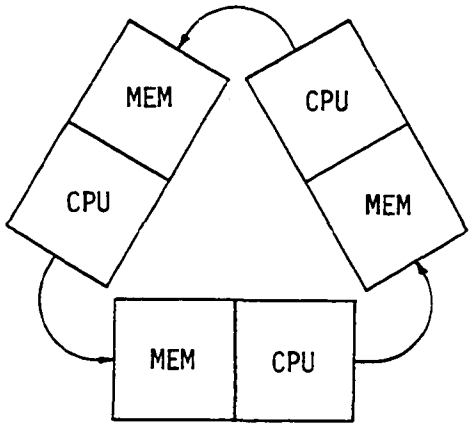

Figure 11. Cycle of Three Processors

The processors can access the shared memories in both cases, and external processors also have access. Therefore, a capability for communication exists in both configurations. The tradeoff is primarily one of complexity vs. flexibility. The design shown in Figure 12 would be less complex and costly, because unidirectional address buffers are simpler than bidirectional transceivers, and because less arbitration logic would be required. However, a bidirectional design would allow more flexibility in accessing shared data. This

feature might be of value in certain applications, especially if messages were routed through the shared memory without involving the local processor. The communication load of intermediate processors could then be reduced, for traversals of two or more hops.

Z80 Microprocessor Implementation

The schematics shown in Figures 14 through 16 represent a very simple implementaion of the circuitry for an MP. Ideally, a more advanced processor/interface would be used, but this arrangement allowed most of the multi-microcomputer principles described above to be tested. The circuits depicted here use a Z80 microprocessor as a processing element. The Z80 and its memory and I/O rescurces are not shown, but the essential control lines are included in the diagrams. This implementation is similar to the unidirectional MP shown in Figure 12, except that the processor partition is omitted.

As mentioned earlier, the function of the processor partition is the prevention of deadlock -- a condition which can also be avoided by eliminating cycles from the MP interconnection structures. In a general sense, the extra partition is required, but not for the simple configuration used in the software experiments to be described in the next major section.

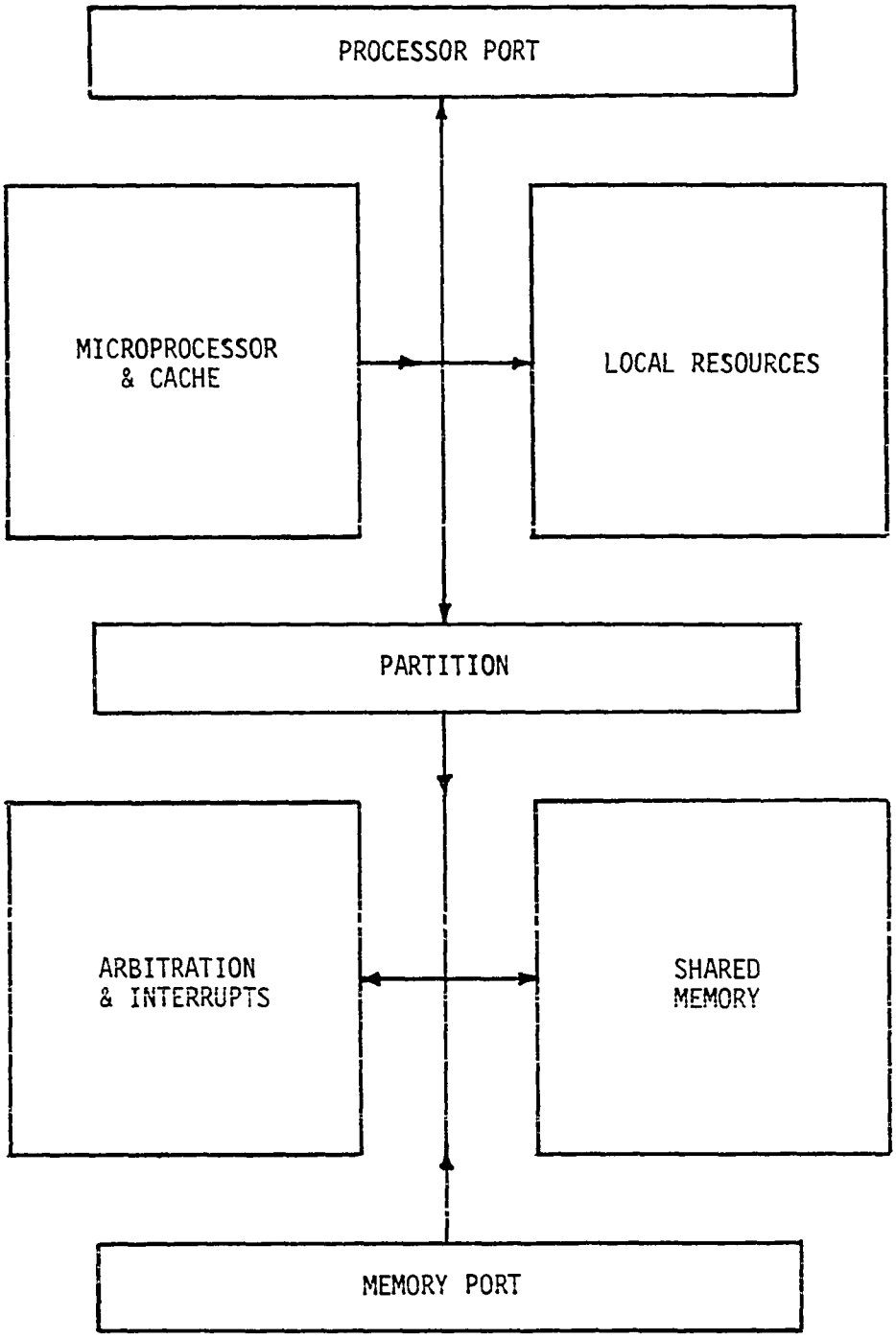

Figure 12. MP with Unidirectional Ports

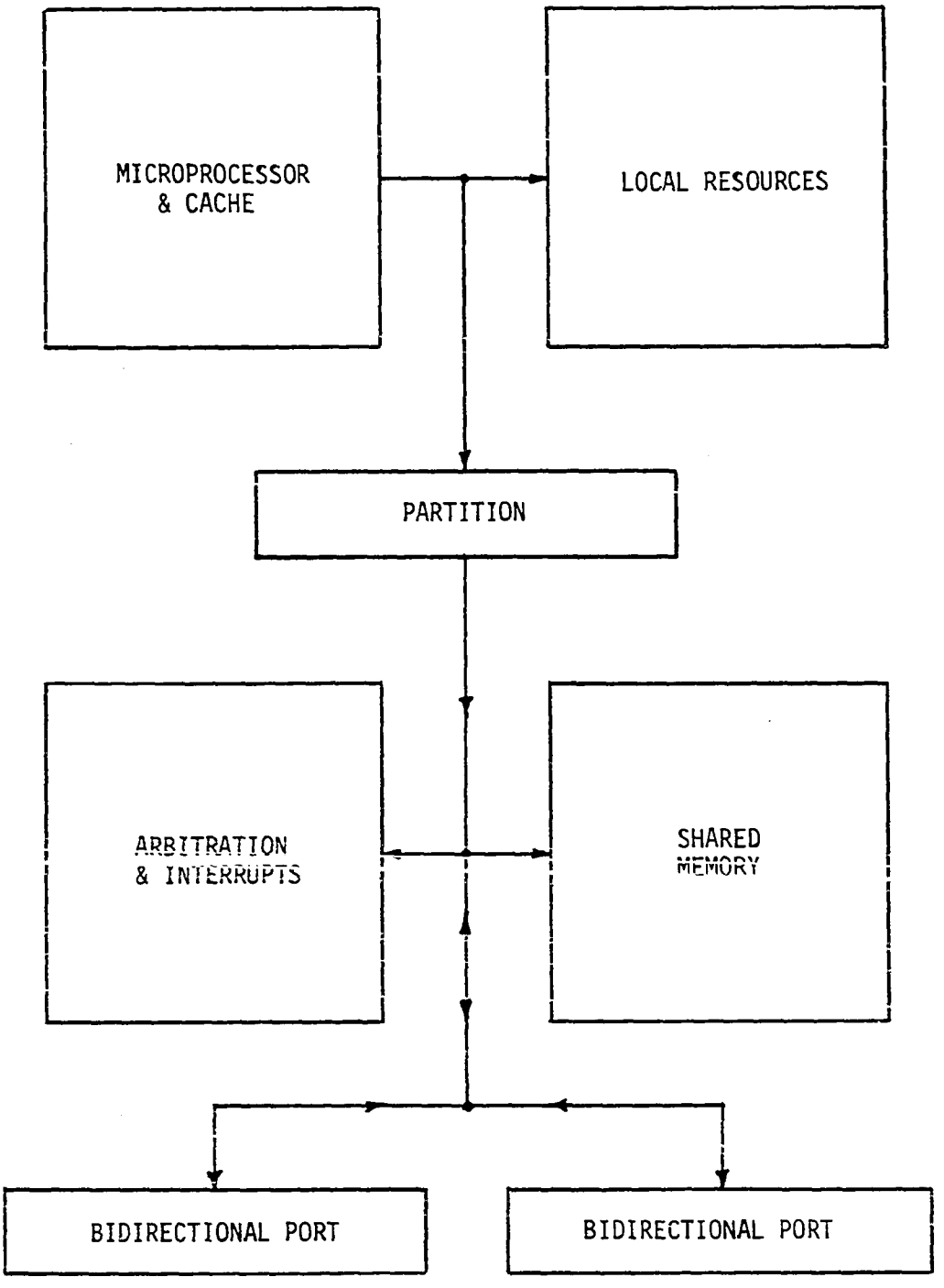

Figure 13. MP with Bidirectional Ports

Figure 14 shows the processor port of Figure 12 in greater detail. The two 74LS244 octal buffers are used to enable the local Z80 address lines onto an external bus. The 74LS245 octal transceiver allows data to pass between the Z80 and the bus, in either direction. The 74LS125 switches the four Z80 memory and I/O control lines onto the bus. These ICs are TTL three-state devices, and all are enabled by the bus grant signal from an external arbiter. Additionally, the 74LS245 requires the local read signal, to control the direction of the data transfers.

Figure 15 shows essentially the same functions, but for the memory port of Figure 12. The circuits differ in that the memory port includes some additional address mapping logic to transform the external address to an internal address, in the range of the local memory and I/O decoding circuitry. The three-state ICs in the memory port are enabled by the DMA, (direct memory access) signal of the local processor. A separate arbiter and a Z80 bus partition could be used to divide the Z80 address space into local and shared areas, but this was deemed unnecessary for experimental purposes. Additionally, the external write signal is used with the memory port 74LS245, to control the direction of the data transfers.

Figure 16 shows the remainder of the circuitry required for a simple MP interface. This consists of a section for decoding an address in the range of the external bus, a

generator for the bus request and Z80 wait signals, and a semaphore circuit. The address decoder uses the upper address lines of the Z80 to determine when the address is within the bus segment. The decoder drives the bus request and Z80 wait lines low. When the bus grant line from the external arbiter drops low, the Z80 wait line goes high. This terminates the Z80 wait state, and allows it to proceed with its external bus cycle.

The semaphore logic was included to allow external processors to synchronize their queuing operations with those of the local processor. If one processor attempts a queuing operation while another processor has one in progress, then the queue structure may be disrupted. Thus, a doctrine of mutual exclusion is followed, so that only one processor is allowed to perform communication queuing at a time.

A semaphore is the name for a circuit or operation which permits mutual exclusion. A Z80 semaphore must be implemented with additional logic as shown here, because the Z80, like most microprocessors, is incapable of performing semaphore operations on memory locations. Larger computers use special test-and-set instructions to implement memory semaphores.

The semaphore circuit used here is set by a write operation at its I/O port address. It is reset by a read operation at that address. However, the value of the semaphore can be determined by a read operation, through data line #7, before the reset signal is generated.

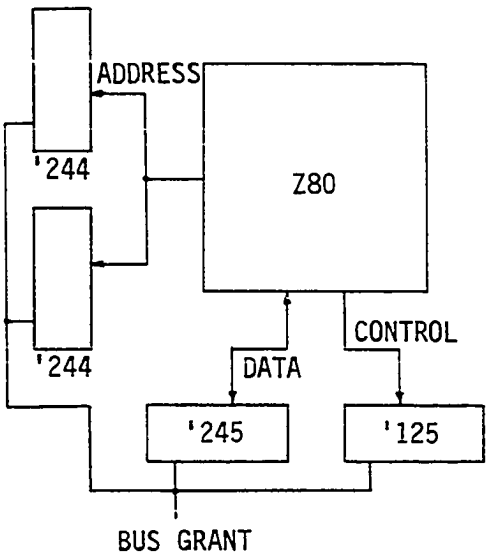

Figure 14. Z80 Processor Port

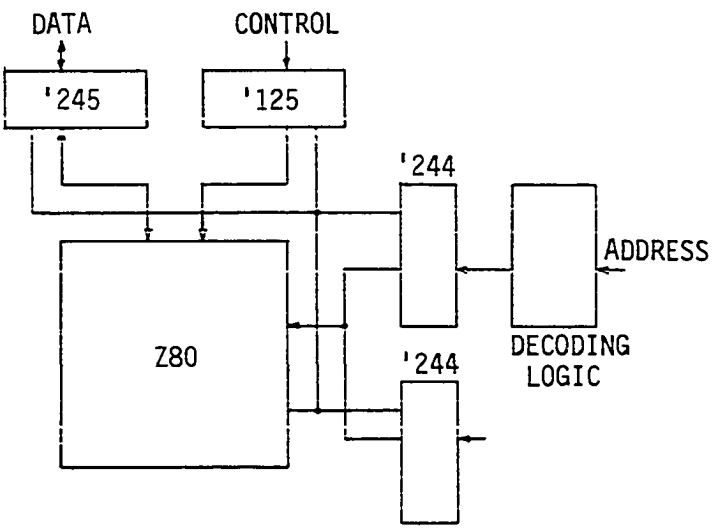

Figure 15. Z80 Memory Port

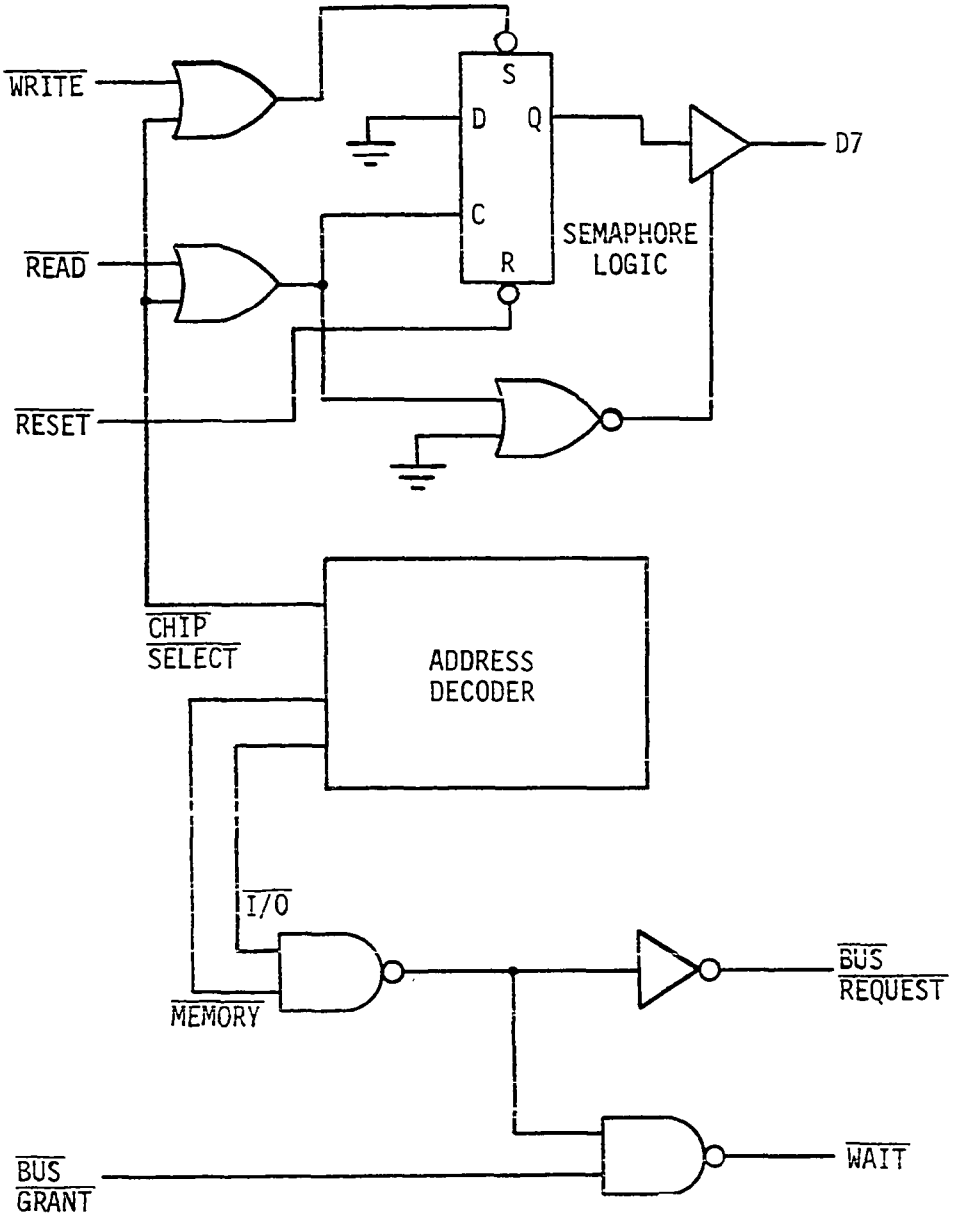

Figure 16. Remainder of MP Circuitry

So, if the semaphore is set, this indicates to the processor which reads it that the communication queuing structure is available for manipulation. If the semaphore is found to be reset at the time of a read, then the communication structure is temporarily in use, and the processor must wait. Since the semaphore is always reset at the termination of a read cycle, a second read operation, without an intervening write, will find the queuing structure unavailable. When a processor completes its queuing operation, it sets the semaphore by writing to it, allowing a single blocked processor to proceed.

The purpose of the MP interface described above was to allow the interconnection of two or three small Z80 microcomputers, so that a multi-tasking software package could be tested. As it turned out, the operation of the MP and the software was verified with a dual-processor configuration, as described in the following section.

MULTI-TASKING SOFTWARE

The programs described in the following pages were used to explore several questions about the effectiveness of the techniques presented in preceding sections. First, can a program written and optimized for a uniprocessor be effectively rewritten for a multi-microcomputer system? Secondly, to what extent does the communication implicit in a multi-microcomputer implementation influence computational efficiency? Is memory contention a significant factor? Does a mutually exclusive queuing system provide a workable and reliable communication channel? And finally, can a near-linear speedup be achieved as more processors are added? The experiments presented here do not address these questions rigorously; however, the results seem to speak positively for multi-microcomputer implementations, at least for certain types of problems.

The programs presented in Appendix B were written in a high-level language developed at Iowa State University, called Portal (6). Several small programs were written in Z80 assembly language, for utility purposes, and are contained in Appendix C. The software was compiled or assembled, linked and downloaded, using a PDP-11/34 system under Unix. The Z80 microcomputers used firmware monitors, to allow program downloading and debugging. The Portal programs were all developed and checked on the host system, before being recompiled for the Z80s.

The first program, called "unipro," was written for a uniprocessor, to calculate the average path length through Geode structures. It uses the same algorithm as the PL/I program TRAVEL, presented earlier. The main difference is that many traversals are performed by unipro. The traversals are produced by the function "produce," and are accomplished by "consume."

Three other functions are invoked in the main routine of unipro. The function "initialize" first sets the initial value of the program variables. Then, "sclk" starts the real-time clock interfaced to the Z80 microcomputer. When the main loop finishes, "rclk" stops and reads the clock. The two clock functions were written in Z80 assembly language, and were linked with the main module.

These programs use two parameters, and produce two results. Respectively, the parameters P and R are the number of ports and the level of recursion of a given Geode. These constants determine the complexity of the resulting series of traversals. This, in turn, determines the run-time of the main loop -- an interval measured by the clock routines. The other result is a record of the total number of hops performed in the main loop, which is stored in the 16-bit words u16 and l16. This quantity, when divided by the number of traversals, gives the average path length through the selected Geode.

The second program "produce" is a version of unipro, modified for multi-tasking with more than one Z80 processor.

It is more complex than unipro, reflecting the inclusion of a queuing system for communications. This required two additional assembly language programs, "p_prd" and "v_prd," which perform the semaphore operations described previously. Four additional Portal functions were required as well, to perform the queuing operations.

The program starts in the same way as unipro, by invoking the initialization and clock start-up routines. However, the semaphore is set before entering the main loop, by invoking v_prd, to indicate to other processors that the queuing system is available.

The structure of a queue element is declared in the first part of the variables section. The queue elements contain fields for source and destination addresses, and for the traversal length. They are chained together, through their link fields, into two separate queues. The two queues consist of elements containing tasks, with "thead" and "ttail" as pointers, and of elements containing replies, using "rhead" and "rtail." A task consists of a source and destination address pair, while a reply gives the distance between the two nodes, using the "length" field.

The function "get-reply" removes an element from the reply queue, and adds the length to l16 and u16. Then, produce rewrites the queue element with a new source and destination address pair. Next, "rel_task" releases the element onto the task queue, where it may be picked-up and

executed by any free processor. The processor executing produce will become free when it empties the reply queue -- a condition detected by get_reply, and indicated when the Boolean variable "flag2" is set.

This condition causes produce to perform one of the traversals it has previously generated, by invoking get_task, consume and rel_reply. These routines remove a task from the task queue, execute it, and return the reply to the reply queue.

This results in a producer/consumer relationship between the two sections. The output queue of the producer is the input queue of the consumer. The opposite is true in the case of the reply queue. This relationship is demonstrated by the third program, "consume," which is essentially identical to the consumer section of produce.

The consumer is far more computationally complex than the producer. Therefore, one producer can serve many consumers. Since a global queuing structure is used, with semaphore synchronization, any processor with access to the memory containing the queue structure can function independently as a producer or consumer. Such an arrangement permits expansion of the system to any size, by simply adding processors loaded with the appropriate producer or consumer software.

However, additional care is required to achieve a proper balance. Since one producer can serve many consumers, it only makes sense to add extra consumers at first. Once the limit

of a single producer is reached, it will tend to become a system bottleneck. This calls for the addition of another producer and a group of consumers, if increased performance is required.

At least three main factors tend to reduce the efficiency of such multi-tasking systems. First, communication implies that queuing routines must be invoked, requiring some of the available processor cycles. Secondly, a processor may spend some time waiting at a semaphore, while another processor performs queuing operations. Finally, memory contention can cause a processor to wait on a cycle-by-cycle basis, while another processor completes a memory access. The experiments described in the following paragraphs provide some insight into the significance of these factors to the operation of multi-microcomputer systems.

Experiments

The five simple experiments described in the next few paragraphs were performed with a dual Z80 configuration. One processor was connected to the Unix system and a CRT terminal, while the other communicated only through the memory of the first. This secondary processor was attached to the main unit through the memory port diagrammed in Figure 15.

Programs for the secondary processor were first downloaded into the memory of its host. The semaphore circuit was set twice in succession -- once to tell the secondary processor to load the program into its own local memory, and

again to tell it to begin execution. The programs for the primary Z80 were then loaded and executed.

The secondary processor had access to the memory and I/O space of the primary through its memory port. Since it used the queue structure for communication, its first action was to read the semaphore. The semaphore was initially reset, causing the secondary to wait for the primary to load, start and initialize its program, and then to set the semaphore. After this point, both processors were in full operation, communicating through the semaphore-protected queuing structure.

The first experimental step was to compare the performance of a single processor executing the first working version of unipro, with a dual-processor running the earliest versions of the programs produce and consume. Since none of the programs had been optimized, the results are of limited value, but the observed speedup was 1.22, for a (3,4) Geode.

At this point, the main goal had been to get the system working, so the software had not been fully developed. The produce program had no consumer section. Its only function was to produce tasks and sum the replies. The consume program accessed the task element as contained in the memory of the producer, instead of obtaining a local copy. This caused some memory contention, since the task element was accessed frequently by the consumer.

So, the second experimental step was to modify the

consumer into its present form, to test the effect on the computation rate. The function get_task now copies task elements into variables in the local memory of the consumer, before they are used in traversals. This reduces the number of accesses to the producer's memory, and so improves performance -- as long as the copy operation requires less time than would be involved in contention. The speedup ratio increased to 1.47 as a result of this modification.

Later analysis determined that this was not a simple case of memory contention. Contention, by definition, is a factor; however, it may not have been a very important one in this case. The performance improvement can be accounted for by the improved code generated by the compiler when data accesses use ordinary variables in local memory, instead of pointer-type variables. Before the modifications to the consumer, pointers were used to give indirect access to variables outside the local data segment -- the queue elements in the producer's memory area. Elimination of this level of indirection was probably more responsible for the resulting speedup than was the virtual elimination of contention.

The results of this analysis led to an examination of the techniques used in developing the software, as the third experimental step. All programs were modified, like the consumer, to take advantage of the characteristics of the compiler and the Z80 processor. One major change was the use of bytes as variables, instead of 16-bit words, whenever

possible, allowing more efficient code generation for the 8-bit Z80. These changes were more effective for unipro than for the dual-processor arrangement, because the speedup factor dropped to 1.07, with a (3,4) Geode.

Obviously, the dual-processor arrangement was seriously out of balance. The consumer program proved to be much slower than the producer, causing a bottleneck. This was demonstrated by a fourth series of experiments, which also resulted in the observation of an interesting paradox.

The approach involved the addition of successively greater amounts of delay to the producer, to see how much effect this had on the execution speed of the dual-processor system. Each time the producer, (operating without a consumer section) produced a task, a delay function was invoked. Since a queuing system was used, this had no effect on the execution speed, as long as the producer ran fast enough to keep the task queue completely full. Since the producer processor was not allowed to consume tasks, it idled most of the time, accounting for the low 1.07 speedup ratio.

Adding delay to the producer did not delay the consumer, as long as the task queue remained full, and the reply queue empty. The delay paradox was observed when the performance of the system suddenly increased after an increase in the delay. At this point, the producer and consumer became balanced, in terms of their relative execution speeds. This improved performance because the queuing programs ran faster when their

respective queues were neither full nor empty, but somewhere in between. This is the case with a balanced system, with the queue length varying over a range. Thus, queue size is a factor in queuing efficiency, even with a well-balanced system.

The results of this fourth series of tests led to the inclusion of a consumer section in the main loop of the producer, to bring the producer to its present form. Since delays, up to a certain point, had no detrimental effects, the extra computations could only help improve efficiency. The addition of the consumer section, yielding one producer and two consumers, caused a dramatic jump in the speedup ratio from 1.07 to 1.67 for a (3,4) Geode.

The fifth series of experiments involved testing the final configuration with Geodes of several types, as shown in Table 2. Three comparisons were made. First, unipro was compared to the producer program, in its present form, (with a consumer section) to determine effect the additional communication programs had on performance.

These ratios are shown in column C1 of the table. As the average distance through the Geode increased, C1 increased, indicating a lesser effect of communication. A greater average distance means that more hops are performed in a typical traversal. In turn, this causes the consumer to become more compute bound, lessening the communication load.

Table 2. Experimental Results

GEODE	AVE1	C1	C2	C3
(2,7)	42.66	0.91	1.82	2.00
(3,5)	11.15	0.85	1.71	2.00
(4,4)	9.78	0.81	1.62	1.99
(5,3)	5.09	0.71	1.50	2.10
(8,3)	5.78	0.75	1.47	1.96

The C2 column shows a comparison between unipro and the full dual-processor configuration. Because a semaphore-protected queuing system was used, no changes were required in the producer -- it was only necessary to plug in the additional processor and start its consumer software. More consumers could have been added, if the required hardware had been implemented.

The results follow those shown in column C1. The consumer becomes more compute bound as the average path length increases, reducing the need for communication, which results in greater speedup. This illustrates that the performance of a multi-microcomputer system can be estimated from observations of a uniprocessor.

If the use of communication functions slows the augmented uniprocessor algorithm considerably, then less speedup will be attained by adding extra processors. However, if tests with the augmented uniprocessor are encouraging, then a basis exists for proceeding with the multi-microcomputer implementation.

Column C3 shows a comparison of the producer, running with a consumer section on a uniprocessor, with the dual-processor configuration. In most cases, the execution speed is essentially doubled with the dual processor. Again, this illustrates that the inefficiencies in multi-tasking systems are mainly associated with the process of communication. Except for communication overhead, a linear speedup could be realized as extra consumers are added, until the producer becomes overloaded.

CONCLUSIONS

The work described in this document demonstrates that a recursive interconnection structure can be used to construct large multi-microcomputer systems. Such systems are very promising for future implementation with VLSI technology. This paper describes how a Geode system can be constructed, using a processor element called an MP. The design alternatives were considered, and two MP prototypes were built and tested. The two prototypes were used to develop a multi-tasking algorithm based on producers and consumers. A semaphore was implemented in hardware, to allow synchronization of the multi-tasking software. The results of a series of experiments indicate that multi-microcomputers can be cost-effective, as long as the appropriate design techniques are utilized, as described below.

Advanced VLSI microcomputers are needed, with features equivalent to most mainframe computers incorporated into a single package. Some of these features are 32- and 64-bit word lengths, hierarchical memories with mapping hardware, and a full set of arithmetical and logical instructions. Advanced uniprocessor techniques such as pipelining would be useful if they could be fitted into the package. The microprocessors developed in the last decade, including the latest 16-bit versions, are generally too primitive for most computationally intensive applications.

The applications can be compute bound, or in some cases, I/O bound. Compute bound algorithms spend little of their time communicating, so computations proceed with maximum rapidity. Algorithms which are not compute bound can be efficiently implemented if they naturally involve buffer-oriented data manipulations which fit into a queue structure. In this case, a uniprocessor would also be limited by the queuing operations, so parallel execution of tasks would introduce no additional overhead.

The memory system should be partitioned into local and global hierarchies to minimize the effects of memory contention. If processors access their data segments frequently, then the data should be moved into the local area. For this reason, code should not be shared directly, but a common copy could be maintained in the shared memory. Partitioning also fits well with virtual-memory and cache-oriented designs.

A producer/consumer multi-tasking algorithm works well, and can be expanded to any size. The division, in terms of execution time, need not be equal, as long as the bottleneck process can be replicated to achieve a proper balance. One producer could serve many identical consumers, or one consumer could process the output of several producers, depending on their relative speeds. The queuing system tends to mask any temporary variations in speed, as long as the queues are long enough.

The MP approach can be used to implement many different structures, with the shared busses representing nodes, and the attached MPs providing links. Irregular structures can also be implemented, using routing tables instead of fixed routing algorithms. In this way, structures can be created to handle specialized problems.

Memory-bus oriented designs are not inherently better than broadcast channels and serial or parallel communication links, but they are simple to implement, and work well for short data transfers. Since low communication loads are necessary for good computational efficiency, slower and cheaper communication methods can be used as long as they provide sufficient bandwidth to allow a queuing system to maintain a relatively constant throughput. Serial communication seems advantageous from the standpoints of complexity and connectivity, especially if communication bandwidth is low and messages are long.

The Geode interconnection structure could be useful in orthogonal types of problems. Applications like spatial correlation and artificial vision seem to fit particularly well. The four-port structure of Figure 8, with the recursive central supernode, could be especially effective in the latter, while eight-port Geodes might be preferable for three dimensional spatial problems.

Cost-effectiveness is the central idea behind multi-microcomputer proposals. Large mainframe computers can

provide the same power, but VLSI processors can be mass-produced very inexpensively -- at least, this will soon be the case. Structures like Geode, which allow easy interconnection of processors like MP, promise to make large-scale computing relatively cheap during the next decade. Some areas, such as multi-tasking operating systems, and concurrent high-level languages need more development, but the necessary principles are well-established.

BIBLIOGRAPHY

1. Adams, G. and Rolander, T. "Design Motivations for Multiple Processor Microcomputer Systems." Computer Design 15 (July 19, 1978): 81-89.

2. Anderson, G. and Jensen, E. "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples." ACM Computing Surveys 7 (December 1975): 197-213.

3. Barnes, G.; Brown, R.; Kato, M.; Kuck, D.; Slotnik, D. and Stokes, R. "The ILLIAC IV Computer." IEEE Transactions on Computers C17 (August 1968): 746-757.

4. Brinch Hansen, P. "Structured Multiprogramming." Communications of the ACM 15 (August 1972): 574-578.

5. Davies, D.; Barber, D.; Price, W. and Solomonides, C. Computer Networks and Their Protocols. New York: Wiley, 1979.

6. Davis, J. "A High-level Programming Language for Microcomputers." Master's thesis, Iowa State University, 1981.

7. Dennis, J. "Data Flow Supercomputers." Computer 13 (November 1980): 48-56.

8. Despain, A. and Patterson, D. "X-Tree: A Tree Structured Multi-Processor Computer Architecture." Symposium on Computer Architecture, Conference Proceedings 5 (April 1978): 144-151.

9. Enslow, P. "Multiprocessor Organization- a Survey." ACM Computing Surveys 9 (March 1977): 103-129.

10. Finkel, R. and Solomon, M. "Processor Interconnection Strategies." IEEE Transactions on Computers C29 (May 1980): 360-371.

11. Friedman, A. and Simoncini, L. "The Effect of LSI Technology on the Theory of Modular Computer Design." Computer 11 (July 1978): 60-67.

12. Fuller, S.; Ousterhout, J.; Raskin, L.; Rubinfeld, P.; Sindhu, P. and Swan, R. "Multi-microprocessors: an Overview and Working Example." Proceedings of the IEEE 11 (February 1978): 216-228.

13. Jansen, P. and Kessels, J. "The DIMOND: A Component for the Modular Construction of Switching Networks." IEEE Transactions on Computers C29 (October 1980): 884-889.

14. Kimbleton, S. and Schneider, G. "Computer Communications Networks: Approaches, Objectives, and Performance Considerations." ACM Computing Surveys 7 (September 1975): 129-173.

15. Lipovski, G. "On a Varistructured Array of Microprocessors." IEEE Transactions on Computers C26 (February 1977): 125-138.

16. Reeves, A. "A Systematically Designed Binary Array Processor." IEEE Transactions on Computers C29 (April 1980): 278-287.

17. Rodrigue, G.; Giroux, E. and Pratt, M. "Perspectives on Large-scale Scientific Computation." Computer 13 (October 1980): 65-80.

18. Schwartz, M. Computer Communication Network Design and Analysis. New York: Prentice-Hall, 1977.

19. Sugarman, R. "'Superpower' Computers." Spectrum 17 (April 1980): 28-34.

20. Wittie, L. "Communication Structures for Large Networks of Microcomputers." IEEE Transactions on Computers C30 (April 1980): 264-273.

ACKNOWLEDGEMENTS

I would like to thank Professor A.V. Pohm of the Electrical Engineering Department at Iowa State University, for giving me the opportunity to work freely in my area of interest, for providing valuable theoretical insights relating to processor/memory configurations and communications, and for arranging to obtain equipment, parts and computer time.

I would also like to thank Professor R.J. Zingg for giving his attention to some of the questions raised by the Geode structure, and for the hours spent in discussing the implications of Geode, X-tree and other structures.

Additionally, Gary Bridges, Steve Christiansen and James Davis provided assistance with the Unix operating system, the ISU local data network, and the Portal programming language.

This work was supported by the Engineering Research Institute and the Department of Electrical Engineering at Iowa State University.

APPENDIX A: TRAVEL PROGRAM

```
TRAVEL: PROC OPTIONS(MAIN);

  DCL SRC        CHAR(8) VAR;     /* SOURCE ADDRESS */
  DCL DST        CHAR(8) VAR;     /* DESTINATION ADDRESS */
  DCL R          FIXED BIN(15);   /* LEVEL OF RECURSION */
  DCL CTR        FIXED BIN(15);   /* COUNTER VARIABLE */
  DCL PTR        FIXED BIN(15);   /* STRING POINTER */
  DCL TCHAR      CHAR(1);         /* CHARACTER VARIABLE */

SRC = 'AB'; DST = 'DC'; R = LENGTH(SRC);

PUT EDIT(SRC,DST,R) (A(R),X(2),A(R),X(2),F(1));

DO WHILE(NEW);  /* HOP FROM NODE TO NODE */
  PUT SKIP EDIT(SRC,TCHAR) (A(R),X(2),A(1));
END;
STOP;


/*********************************************************/

NEW: PROC RETURNS(BIT(1));

  TCHAR = PORT;  /* GET THE PORT ID */
  IF TCHAR = '!' THEN RETURN('0'B);
  DO PTR = R TO 1 BY -1 WHILE(SUSTR(SRC,PTR,1)=TCHAR); END;
  SUBSTR(SRC,PTR,R-PTR+1) =
    TCHAR || REPEAT(SUBSTR(SRC,PTR,1),R-PTR-1);
  RETURN('1'B);  /* NOT YET FINISHED */

END NEW;

/*********************************************************/

PORT: PROC RETURNS(CHAR);

  DO CTR = 1 TO R;
    TCHAR = SUBSTR(DST,CTR,1);
    IF TCHAR ¬= SUBSTR(SRC,CTR,1) THEN RETURN(TCHAR);
  END;

  RETURN('!'); /* SRC=DST */

END PORT;
END TRAVEL;
```

APPENDIX B: PORTAL PROGRAMS

Unipro: Uniprocessor Version

```
procedure sclk  external;     (* start real-time clock *)
procedure rclk  external;     (* read real-time clock *)

const
  P             = 3;     (* number of ports per node *)
  R             = 4;     (* level of recursion *)
var
  src[0:R-1]:   byte;    (* address of source node *)
  dst[0:R-1]:   byte;    (* address of destination node *)
  sstr[0:R-1]:  byte;    (* producer's copy of src *)
  dstr[0:R-1]:  byte;    (* producer's copy of dst *)
  first[0:R-1]: byte;    (* producer's starting point *)
  (* base P digits-- used in addresses *)
  ports[0:7]:   byte {'A','B','C','D','E','F','G','H'};
  u16:          word public {0};     (* hop counter-- hish *)
  l16:          word public {0};     (* hop counter-- low *)
  step:         word;    (* production step counter *)
  last:         word;    (* production step limit *)

procedure main public;
begin
  initialize;   (* initialize variables *)
  sclk;         (* start real-time clock *)
  while step < last do begin
    produce;
    consume;
  end;
  rclk;         (* read real-time clock *)
end;

(* compare two strings *)
procedure cmpstr(ptr1,ptr2,cnt): byte;
parm
  ptr1:         @byte;
  ptr2:         @byte;
  cnt:          byte;
begin
  while cnt > 0 do begin
    if @ptr1++ <> @ptr2++ then return false
    else cnt--;
  end;
  return true;
end;
(*
```

```
*)
procedure initialize;    (* initialize variables *)
var x,y:                 byte;
begin
  step := 0;
  last := 1;
  for x := 1 to R-1 do last := last*P;
  for y := 0 to R-1 do begin
    sstr[y] := 'A';
    dstr[y] := 'A';
    first[y] := 'A';
  end;
end;

procedure next(sptr);    (* increment addresses-- base P *)
parm sptr:       @byte;
var
  ctr,indx:      byte;
  alpha:         byte;
  switch:        byte;
begin
  indx := R-1;
  switch := true;

  while switch do begin
    alpha := sptr[indx];
    ctr := 0;
    while alpha <> ports[ctr] and ctr < P do ctr++;
    if ctr = P-1 then alpha := 'A'
    else alpha := ports[ctr+1];
    sptr[indx] := alpha;
    if alpha <> 'A' or indx = 0 then switch := false
    else indx--;
  end;
end;

procedure produce;
var x:           byte;
begin
  for x := 0 to R-1 do begin    (* copy {src,dst} *)
    src[x] := sstr[x];
    dst[x] := dstr[x];
  end;
  next(.dstr[0]);    (* increment destination address *)

  if cmpstr(.dstr[0],.first[0],R) then begin
    next(.sstr[0]);    (* increment source address *)
    step++;
  end;
end;
(*
```

```
*)
procedure consume;     (* hop from node to node *)

var
  ctr,tmp:                byte;
  enabled,switch:         byte;
  schar,dchar:            byte;

begin
  enabled := true;

  while enabled do begin
    if cmpstr(.src[0],.dst[0],R) then enabled := false
    else begin
      switch := true;
      ctr := 0;

      while switch and (ctr < R) do begin
        schar := src[ctr];
        dchar := dst[ctr];
        if schar <> dchar then switch := false
        else ctr++;
      end;

      tmp := R-1;


      (* hop within same cluster *)
      if dchar <> src[tmp] then src[tmp] := dchar

      else begin    (* hop outside local cluster *)
        switch := true;

        while switch do begin
          if dchar <> src[tmp] then switch := false
          else tmp--;
        end;

        schar := src[tmp];
        src[tmp] := dchar;

        while tmp < R-1 do begin
          tmp++;
          src[tmp] := schar;
        end;
      end;

      if not ++116 then u16++;     (* count hops *)
    end;
  end;
end;
```

```
                    Produce: Multi-tasking Producer
procedure p_prd external;   (* P semaphore operation *)
procedure v_prd external;   (* V semaphore operation *)
procedure sclk external;    (* start real-time clock *)
procedure rclk external;    (* read real-time clock *)

const
  P               = 3;              (* ports per node *)
  R               = 4;              (* level of recursion *)
  offset          = $4000;          (* address offset *)

var
  (* queue structure *)
  struct q {
    link:         @q;       (* link to next element *)
    length:       byte;     (* length of a traversal *)
    src[0:R-1]:   byte;     (* address of source node *)
    dst[0:R-1]:   byte;     (* address of destination node *)
  };

  (* queue allocation and pointers *)
  task[0:7]:      q         public;     (* queue allocation *)
  thead:          @q        public;     (* head of task queue *)
  ttail:          @q        public;     (* tail of task queue *)
  rhead:          @q        public;     (* head of reply queue *)
  rtail:          @q        public;     (* tail of reply queue *)
  temp:           @q        public;     (* temporary pointer *)
  pptr:           @q        public;     (* producer pointer *)
  cptr:           @q        public;     (* consumer pointer *)

  (* address strings *)
  sstr[0:R-1]:    byte;     (* consumer's copy of source *)
  dstr[0:R-1]:    byte;     (* consumer's copy of destination *)
  first[0:R-1]:   byte;     (* producer's starting address *)
  src[0:R-1]:     byte;     (* producer's copy of source *)
  dest[0:R-1]:    byte;     (* producer's copy of destination *)
  (* base P digits-- used in addresses *)
  ports[0:7]:     byte      {'A','B','C','D','E','F','G','H'};

  (* variables, flags and counters *)
  flag:           byte      {true};       (* temporary flag *)
  flag2:          byte      {false};      (* temporary flag *)
  distance:       byte      {0};      (* length of traversal *)
  step:           word      {0};      (* source counter *)
  last:           word      {1};      (* source limit *)
  pstep:          word      {8};      (* production counter *)
  cstep:          word      {0};      (* completion counter *)
  l16:            word      public    {0};      (* hop counter-- low *)
  u16:            word      public    {0};      (* hop counter-- high *)
 (*
```

```
*)

procedure main public;
begin
  initialize;   (* set up queue structure *)
  sclk;         (* start real-time clock *)
  v_prd;        (* set semaphore-- start synchronization *)
  while cstep < pstep do begin    (* consume all produced *)
    set_reply;    (* try for a queue element *)
    if flag and step < last then begin    (* produce *)
      produce;    (* generate {src,dst} *)
      rel_task;   (* put {src,dst} on task queue *)
    end;

    if flag2 then begin    (* task queue is full *)
      set_task;    (* consume *)
      if flag then begin
        flag2 := false;    (* consume only one task *)
        consume;           (* hop from node to node *)
        rel_reply;         (* put answer on reply queue *)
      end;
    end;
  end;
  rclk;    (* finished-- read real-time clock *)
end;
(* compare two strings *)
procedure cmpstr(ptr1,ptr2,cnt): byte;
parm
  ptr1:         @byte;
  ptr2:         @byte;
  cnt:          byte;

begin
  while cnt > 0 do begin
    if @ptr1++ <> @ptr2++ then return false
    else cnt--;
  end;
  return true;
end;
procedure count;    (* count hops *)
var x,y:        byte;

begin
  y := pptr.length;
  for x := 1 to y do begin
    if not ++l16 then u16++;
  end;
  cstep++;
end;
(*
```

```
*)
procedure get_reply;     (* try for a queue element *)
begin
  p_prd;          (* P semaphore operation *)
  if rhead <> nil then begin     (* check head of reply queue *)
    if rtail = rhead then begin     (* last element *)
      rtail := nil;
      flag2 := true;     (* enable consumption *)
    end;
    pptr := rhead - offset;
    rhead := pptr.link;
    flag := true;
  end
  else flag := false;
  v_prd;
  if flag then count;     (* count the hops *)
end;
procedure get_task;     (* try for a queue element *)
var     x:      byte;
begin
  p_prd;
  if thead <> nil then begin
    if ttail = thead then ttail := nil;
    cptr := thead - offset;
    thead := cptr.link;
    flag := true;
    for x := 0 to R-1 do begin     (* copy {src,dst} *)
      sstr[x] := cptr.src[x];
      dstr[x] := cptr.dst[x];
    end;
    distance := 0;
  end
  else flag := false;
  v_prd;
end;

procedure rel_reply;     (* put answer on reply queue *)
begin
  p_prd;
  cptr.link := nil;
  if rtail <> nil then begin
    temp := rtail - offset;
    temp.link := cptr + offset;
  end;
  rtail := cptr + offset;
  if rhead = nil then rhead := rtail;
  cptr.length := distance;
  v_prd;
end;
(*
```

```
*)
procedure rel_task;     (* put {src,dst} on task queue *)
begin
  p_prd;
  pptr.link := nil;
  if ttail <> nil then begin
    temp := ttail - offset;
    temp.link := pptr + offset;
  end;
  ttail := pptr + offset;
  if thead = nil then thead := ttail;
  v_prd;
end;

procedure next(sptr);    (* increment address string *)
parm sptr:       @byte;

var
  ctr,indx:      byte;
  alpha:         byte;
  switch:        byte;

begin
  indx := R-1;
  switch := true;

  while switch do begin
    alpha := sptr[indx];
    ctr := 0;

    while alpha <> ports[ctr] and ctr < P do ctr++;

    if ctr = P-1 then alpha := 'A'
    else alpha := ports[ctr+1];

    sptr[indx] := alpha;

    if alpha <> 'A' or indx = 0 then switch := false
    else indx--;
  end;
end;
(*
```

```
*)
procedure initialize;    (* set-up queue structure *)
var x,y:         byte;

begin
  for x := 1 to R-1 do last := last*P;
  for x := 0 to R-1 do begin
    srce[x] := 'A';
    dest[x] := 'A';
    first[x] := 'A';
  end;

  for y := 0 to 6 do begin
    task[y].link := .task[y+1] + offset;
    task[y].length := 0;
  end;

  task[7].link := nil;
  task[7].length := 0;
  rhead := .task[0] + offset;
  rtail := .task[7] + offset;
  thead := nil;
  ttail := nil;
end;

procedure produce;    (* generate {src,dst} *)
var x:           byte;

begin
  for x := 0 to R-1 do begin    (* copy current {src,dst} *)
    pptr.src[x] := srce[x];
    pptr.dst[x] := dest[x];
  end;

  pptr.length := 0;
  next(.dest[0]);    (* increment destination *)

  if cmpstr(.dest[0],.first[0],R) then begin
    next(.srce[0]);    (* increment source *)
    step++;
  end;
  pstep++;
end;
(*
```

```
*)
procedure consume;     (* shortest path through Geode *)
var
  ctr,tmp:                          byte;
  enabled,switch:                   byte;
  schar,dchar:                      byte;

begin
  enabled := true;

  while enabled do begin     (* loop til strings are equal *)
    if cmpstr(.sstr[0],.dstr[0],R) then enabled := false
    else begin
      switch := true;
      ctr := 0;

      (* find first location where sstr <> dstr *)
      while switch and (ctr < R) do begin
        schar := sstr[ctr];
        dchar := dstr[ctr];
        if schar <> dchar then switch := false
        else ctr++;
      end;

      tmp := R-1;

      (* hop within same cluster *)
      if dchar <> sstr[tmp] then sstr[tmp] := dchar

      else begin     (* hop outside local cluster *)
        switch := true;

        while switch do begin
          if dchar <> sstr[tmp] then switch := false
          else tmp--;
        end;

        schar := sstr[tmp];
        sstr[tmp] := dchar;

        while tmp < R-1 do begin
          tmp++;
          sstr[tmp] := schar;
        end;
      end;
      distance++;     (* count the hops *)
    end;
  end;
end;
```

Consume: Multi-tasking Consumer

```
procedure p_cns          external;
procedure v_cns          external;

const
  P              = 3;    (* number of ports per node *)
  R              = 4;    (* level of recursion *)

var
(* structure of queue elements *)
struct q {
  link:        @q;       (* link to next element *)
  length:      byte;     (* length of a traversal *)
  src[0:R-1]:  byte;     (* address of source node *)
  dst[0:R-1]:  byte;     (* address of destination node *)
};

(* pointers to queue pointers, in producer's address space *)
thead:         @word   {$56E3};        (* head of task queue *)
ttail:         @word   {$56E5};        (* tail of task queue *)
rhead:         @word   {$56E7};        (* head of reply queue *)
rtail:         @word   {$56E9};        (* tail of reply queue *)
ptr:           @q      public;         (* queue pointer *)
temp:          @q      public;         (* temporary pointer *)
 (* address strings *)
 sstr[0:R-1]:  byte;   (* local copy of src address *)
 dstr[0:R-1]:  byte;   (* local copy of dst address *)

 (* variables, flags and counters *)
 distance:     byte;   (* current length of traversal *)
 flag:         byte;   (* task flag-- set if available *)
 x:            byte;   (* temporary counter *)

procedure main public;
begin
  loop
    get_task;   (* try for a queue element *)
    if flag then begin  (* true if task is available *)
      for x:= 0 to R-1 do begin   (* copy the addresses *)
        sstr[x] := ptr.src[x];
        dstr[x] := ptr.dst[x];
      end;
      distance := 0;
      consume;              (* hop from node to node *)
      rel_reply;            (* release the answer *)
    end;
  end;
end;
(*
```

```
*)
procedure set_task;      (* set a task element *)
begin
  p_cns;          (* semaphore operation *)
  if @thead <> nil then begin    (* check head of task queue *)
    if @ttail = @thead then @ttail := nil;
    ptr := @thead;
    @thead := ptr.link;
    flag := true;
  end
  else flag := false;
  v_cns;          (* reset semaphore *)

end;

procedure rel_reply;     (* release the queue element *)
begin
  p_cns;
  ptr.link := nil;
  if @rtail <> nil then begin
    temp := @rtail;
    temp.link := ptr;
  end;
  @rtail := ptr;
  if @rhead = nil then @rhead := @rtail;
  ptr.length := distance;        (* return the answer *)
  v_cns;
end;

procedure cmpstr(ptr1,ptr2,cnt): byte;  (* compare strings *)
parm
  ptr1:          @byte;
  ptr2:          @byte;
  cnt:           byte;

begin
  while cnt > 0 do begin
    if @ptr1++ <> @ptr2++ then return false
    else cnt--;
  end;
  return true;
end;
(*
```

```
*)
procedure consume;      (* shortest path through Geode *)
var
  ctr,tmp:                      byte;
  enabled,switch:               byte;
  schar,dchar:                  byte;

begin
  enabled := true;

  while enabled do begin        (* loop til sstr=dstr *)
    if cmpstr(.sstr[0],.dstr[0],R) then enabled := false
    else begin
      switch := true;
      ctr := 0;

      (* find the first location where sstr <> dstr *)
      while switch and (ctr < R) do begin
        schar := sstr[ctr];
        dchar := dstr[ctr];
        if schar <> dchar then switch := false
        else ctr++;
      end;

      tmp := R-1;

      (* hop within same cluster *)
      if dchar <> sstr[tmp] then sstr[tmp] := dchar

      else begin        (* hop outside local cluster *)
        switch := true;

        while switch do begin
          if dchar <> sstr[tmp] then switch := false
          else tmp--;
        end;

        schar := sstr[tmp];
        sstr[tmp] := dchar;

        while tmp < R-1 do begin
          tmp++;
          sstr[tmp] := schar;
        end;
      end;

      distance++;       (* count the hops *)
    end;
  end;
end;
```

APPENDIX C: Z80 UTILITY PROGRAMS

```
        ; monitor for secondary processor
        pub     _boot
_boot:  in      a,(0b4h)        ;read the semaphore flip/flop
        bit     7,a             ;test bit 7
        jr      z,_boot         ;keep testing until it's set
        ld      de,1000h        ;dst pointer
        ld      hl,5000h        ;src pointer
        ld      bc,400h         ;counter
        ldir                    ;block move
pause:  in      a,(0b4h)        ;read the semaphore flip/flop
        bit     7,a             ;test bit 7
        jr      z,pause         ;keep testing until it's set
        call    1000h           ;enter new routine
        jp      0h              ;set another one


        ; start real-time clock
        pub     _sclk
_sclk:  ld      a,36h           ;counter 0, mode 3
        out     (0efh),a        ;set mode
        ld      a,0h            ;clear A
        out     (0ech),a        ;lsb
        out     (0ech),a        ;msb
        ld      a,76h           ;counter 1, mode 3
        out     (0efh),a        ;set mode
        ld      a,0h            ;clear A
        out     (0edh),a        ;lsb
        out     (0edh),a        ;msb
        ret


        ; read real-time clock
        pub     _rclk
_rclk:  ld      a,0h            ;latch counter 0
        out     (0efh),a        ;set mode
        in      a,(0ech)        ;lsb
        ld      (1700h),a       ;move to memory
        in      a,(0ech)        ;msb
        ld      (1701h),a       ;move to memory
        jp      8h              ;restart the monitor
```

```
        pub     _p_prd
_p_prd: in      a,(0f4h)        ;read the semaphore flip/flop
        bit     7,a             ;test bit 7
        jr      z,_p_prd        ;keep testing until it's set
        ret

        pub     _v_prd
_v_prd: ld      a,0h            ;clear the accumulator
        out     (0f4h),a        ;reset the semaphore flip/flop
        ret

        pub     _p_cns
_p_cns: in      a,(0b4h)        ;read the semaphore flip/flop
        bit     7,a             ;test bit 7
        jr      z,_p_cns        ;keep testing until it's set
        ret

        pub     _v_cns
_v_cns: ld      a,0h            ;clear the accumulator
        out     (0b4h),a        ;reset the semaphore flip/flop
        ret
```