FERMTOR: A Tunable Multiprocessor Architecture

Jonathan Rose, Wayne Loucks, and Zvonko Vranesic
University of Toronto

Multiprocessing can be cost-effective when a general-purpose system is adaptable to specific uses.

In the search for faster and more powerful computers, researchers have followed two paths. The first concentrates on increasing the speed of a uniprocessor. This can be achieved by making the components faster,[1] by using pipelining,[2] and by exploiting architectural features such as a cache memory and reduced instruction sets.[3]

The second approach is directed to gaining high performance through the use of more than one processor. Indeed, multiprocessors have often been considered a panacea for computing problems. Recent developments in Very Large Scale Integration (VLSI) technology have further motivated this work, because integration promises to make multiprocessing cheaper. This is manifested in three ways:

1) Systolic architectures place many small asynchronous processors in a regular array that can be implemented on one chip.[4] More conventional SIMD (Single Instruction stream, Multiple Data stream) architectures can also be highly integrated.
2) Microprocessors become more powerful as higher levels of integration allow the inclusion of more architectural features on a chip. Today's microprocessors are architecturally similar to yesterday's mainframes. Thus a multiprocessor architecture incorporating general-purpose microprocessors naturally becomes more powerful as technology improves.
3) The MIMD (Multiple Instruction stream, Multiple Data Stream) hardware for communication between processors can be integrated. A circuit that formerly required many TTL chips can be realized on one large-scale chip, limited principally by pin count. Thus, although data paths may need to be external to a VLSI chip, the communication protocol implementation and controlling logic can be integrated easily.

The ideal objective of multiprocessor structures in general, and MIMD architectures in particular, is to obtain linearly increasing throughput, dependent upon the number of processors. Rarely, however, will n processors be n times faster than one processor unless the application lends itself to being subdivided into many parallel subtasks. It is true that while two processors can be made to work almost twice as fast as one processor, this property does not hold for more general cases.

0272-1732/85/0800-0005$01.00 © 1985 IEEE

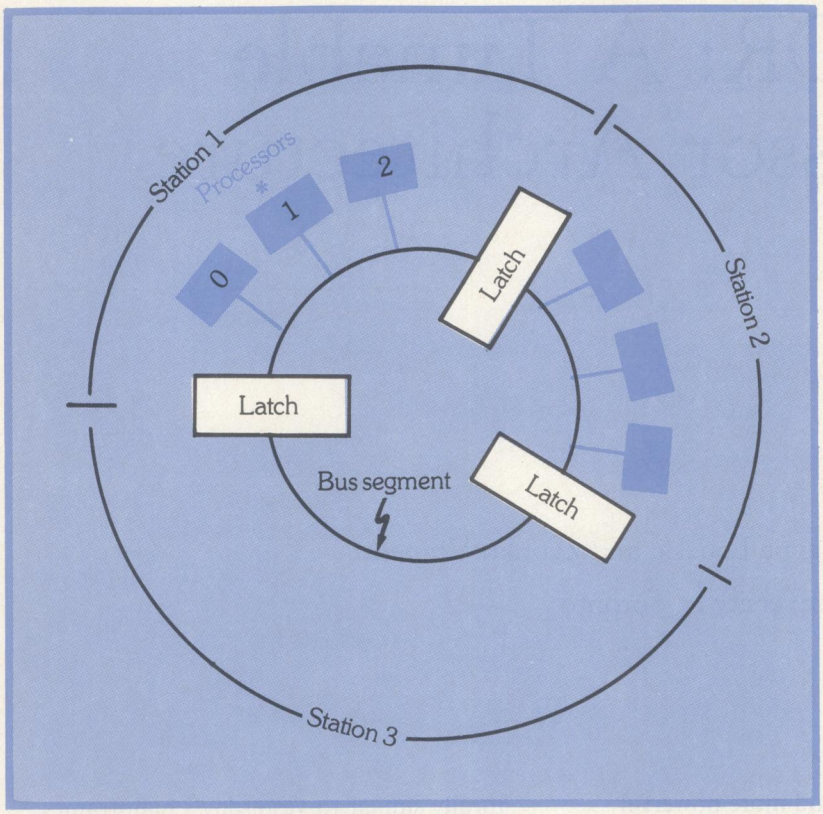


Figure 1. The basic FERMTOR architecture.

The greatest degree of success in multiprocessor systems occurs when they are applied to a specific purpose and the entire machine (i.e., hardware and software) is designed toward that purpose. SIMD processors such as ILLIAC IV[5] have been used for weather prediction, and architectures such as the Cytocomputer[6] have been used for image processing. Associative processors have been used in database machines.[7] MIMD computers have been successful when each processor is assigned a permanent task; for example, in recent personal workstations one processor handles the graphics screen, another acts as a disk server, and a third performs the computation.

However, it is very tedious to have to design and optimize a new architecture and communication system every time one requires a new MIMD system. It would be far more efficient if there were general architectures available, with a standard interprocessor communication scheme and programming environment that could be *tuned* to each new application. Such a system would allow fast design and construction of a powerful special-purpose multiprocessor. It could take advantage of an existing family of VLSI microprocessor chips intended for a general-purpose architecture.

Nevertheless, there is certain to be a trade-off between the efficiency of each implementation and the generality of the basic architecture. The tuned architecture *must* be cost-effective. If the architecture allows cost-performance trade-offs, it will be that much more valuable. An example of an interesting cost-performance trade-off is the SIMD Cytocomputer.[6] Rather than implement a full array of processors, this architecture pipelines pieces of the array through a sub-array processor.

Finally, one of the greatest difficulties in using the parallelism available in an MIMD system is the task of scheduling the work of each processor. The programming environment of a multiprocessor must address this problem directly.

In this article we present FERMTOR, an MIMD architecture developed at the University of Toronto with the goal of addressing the issues raised above.[8, 9, 10] The name stands for "Flexible Extendible Range Multiprocessor at TORonto." FERMTOR is a general MIMD architecture that can be tuned to many applications. It is also a practical multiprocessor whose communications hardware is much less complex than that of an MIMD crossbar or a Banyan network. It has packet-switched, combined-ring, and shared common-access bus interconnection schemes. Processors communicate directly with each other by means of packets. Each processor can be used for either general purposes or special purposes, such as for high-speed numerical computation. A simple version of FERMTOR has been constructed and tested.

FERMTOR has several features in common with a number of previous architectures, although we believe that its overall design provides certain unique characteristics. The manner in which packets flow around the ring, and the fact that a packet arrival actively interrupts a processor, likens FERMTOR to data-flow machines.[11, 12, 13] Farrel[14] uses a ring structure to implement a Generalized Control Flow (GCF) machine. The GCF architecture is a practical attempt at data-flow implementation. FERMTOR

also bears some similarity to CM*[15] in that *processes*, rather than lower level program units such as instructions, communicate among each other. Indeed, CM*, if configured as a packet-switching multiprocessor, could be used in a manner similar to FERMTOR.

Further, FERMTOR's ring and common-access bus structure resemble the topology of the EMMA architecture.[16] EMMA is used as a pattern-recognition machine for postal sorting and is highly successful, incorporating fault tolerance and graceful degradation under faults. The software structure of the prototype FERMTOR is also similar to HM2P[17] in that it uses Hoare's monitors and signals.[18] A great deal of other multiprocessing work exists, but little of it addresses the question of tuning to a specific purpose.

This article discusses the general architecture of FERMTOR, surveys the software structure of the programming environment, gives some details of the hardware implementation, and provides some results obtained with the prototype, including performance measurements. We also discuss two potential applications of FERMTOR and suggest avenues for future work.

FERMTOR architecture

FERMTOR is an MIMD architecture with a part ring, part shared common-access bus communication scheme. The basic structure is shown in Figure 1. The processors are connected to a parallel-pipelined bus called the *P-Bus*. The P-Bus is a ring-like structure of a number of bus segments. Parallel data flows synchronously between the latches, which delineate the bus segments. Each latch, bus segment, and group of processors following it is known collectively as a *station*. There can be a number of processors at each station, the number being limited by the desired bandwidth of the shared common-access bus. Four different types of processors are used:

1) General-purpose processors, such as conventional microprocessors.
2) Special-purpose hardware for high-speed computation, such as array processors.
3) Memory processors that contain and manage the global memory of the system.
4) Input/Output processors that control data flow between FERMTOR and I/O devices such as disks, terminals, etc.

Data is exchanged within a station using a shared common-access bus. Every processor on the P-Bus occupies a unique address by which it can be unambiguously referenced. The address consists of two parts: a station number and a processor number within that station. For example, the processor marked with an asterisk in Figure 1 is at station 1, unit number 1, and so has a P-Bus address of 11.

Basic packet structure. The "slot" of data within a station contains a *packet* of information. The packet is the basic unit of communication among all processors. A packet contains the following fields:

Source. The P-Bus address of the transmitting processor.
Destination. The P-Bus address of the receiving processor.
Operation. Specification of the nature of the packet.
Operand. Data to be operated on.

In addition, when slots travel between stations, three status bits are appended to specify the presence or absence of a packet in the slot and whether or not the packet has been successfully received by the destination station.

P-Bus operation. At each station, a *station manager* controls the flow of packets among local processors and between the neighboring stations. Figure 2 is a block diagram of one station and the P-Bus interface of two processors. When a processor wishes to transmit a packet, it raises a **Request** signal to the station manager. The manager arbitrates the processor requests and sends an **Enable** signal to the selected processor when the first empty packet arrives from the previous station. The requested transfer can be either a *local* transfer between processors in the same station or a *nonlocal* transfer between processors at different stations.

Local transfers. When the packet destination is local to the station, the manager tries to transmit it to that processor as soon as possible over the shared common-access bus. If the destination processor is unable to accept the packet (as

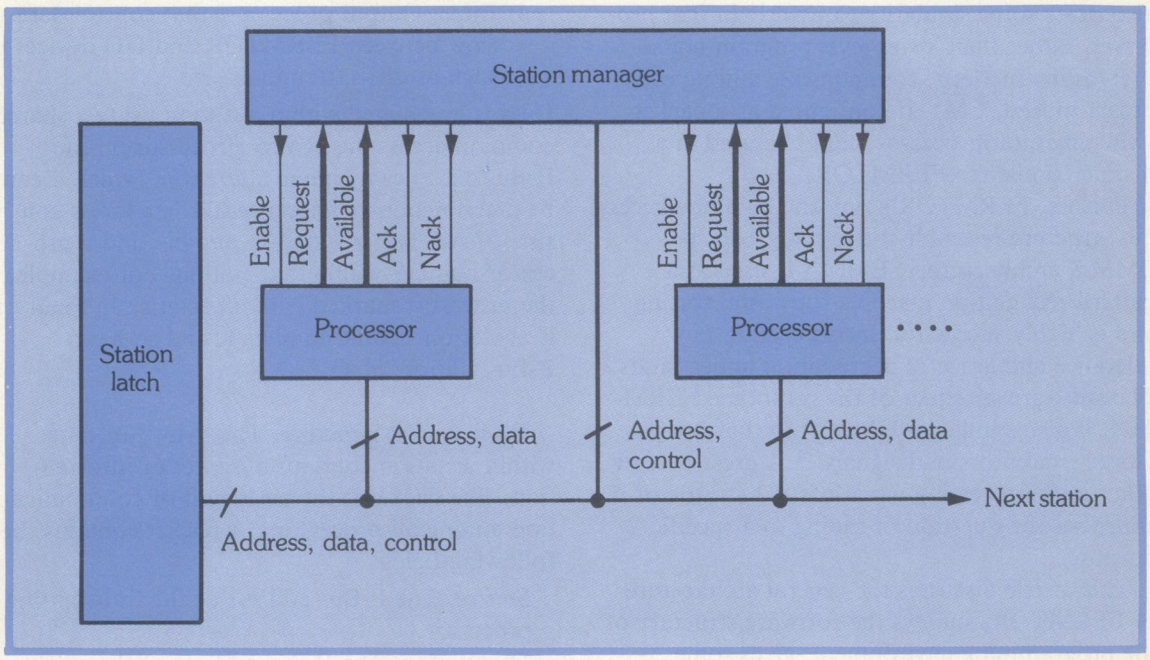


Figure 2. A typical FERMTOR station.

determined by its **Available** signal output) then a **Nack** signal (negative acknowledge) is sent to the transmitting processor. If the packet is accepted, then an **Ack** signal (acknowledge last transfer) is sent to the transmitting processor. The packet slot occupied by this transaction (whether successful or not) is then marked **Empty**. This means that the slot is immediately available for use by succeeding stations on the ring—i.e., a local transfer is always concluded in the time it takes one packet to go through a station.

Nonlocal transfers. When the packet destination is not local to the station, then the packet is transmitted to the next station along the ring. Each station that the packet passes through checks to see if it is the destination station. When the destination station is reached, it tries to transmit the packet to the destination processor exactly as described above for local transfers. If the transmission is successful, the packet is marked **Received**; if not, the packet is marked **Not Received**, using one of the appended flags. When the packet returns to the source station, the station manager sends either an **Ack** or a **Nack** signal to the source processor, depending on how the packet was marked.

Note that, from the processor's point of view, there is no difference between intrastation transfers and interstation transfers, other than the time it takes to transmit the packets. For any unsuccessful transfers, the processor simply regenerates a **Request** signal until the transfer succeeds. The P-Bus protocol allows a processor to have only one packet active on the ring at one time, to prevent it from dominating ring traffic.

Flexibility and extendibility of the architecture. The flexibility of the FERMTOR architecture facilitates the addition of stations and processors. When a processor becomes a bottleneck in a given computation, a second processor of this type can be added to take on a share of those computations, provided the problem is divisible. This is part of the process of tuning the FERMTOR architecture to a special-purpose application.

The P-Bus ring structure makes the hardware complexity of FERMTOR directly proportional to the number of processors. If a processor is added to an existing station, the increase in communication hardware is minimal. When a new station is added, the full station-manager hardware must be included.

The exact configuration of a FERMTOR implementation can be tuned to achieve sufficient interprocessor communication speed. There is a trade-off between having fewer processors at a

station and thus many stations, or many processors at a station and few stations. If there are only a few processors at a station, then they can communicate quickly among themselves. However, because there are many stations, the latency (the time required for a packet to traverse the ring) becomes larger. Conversely, with many processors per station and fewer stations the latency is small. In this case the intrastation bus contention is larger because many processors share one bus. Thus, the optimal number of processors per station is application-dependent. Loucks[9] found that three or four processors per station was best for general-purpose computation.

If the FERMTOR structure is viewed as a hierarchy, we can put the question of the number of processors per station in perspective. Further levels of hierarchy can be added to the P-Bus by using multiple rings, each communicating via inter-ring bridges. Thus an implementation could be a tree of rings or even a ring of rings. The farther the destination of a packet is from its source (in terms of levels of the hierarchy it must travel) the greater the time the packet transportation takes.

If it is apparent from the application that two or more processors communicate very frequently, then they should be placed at the same station. Groups of processors that communicate less frequently, but still at a significant rate, should be placed on the same ring. If two distinct groups of processors have little need to communicate, then they should be placed on separate rings. In this manner a FERMTOR configuration can partition the processors as required by the application.

The hierarchy of packet transportation is similar to CM*[15]; but, since it can be extended indefinitely, it is cleaner and more symmetrical.

Software environment

The viability of a multiprocessor is determined not only by its architecture, but by the software infrastructure as well. The communication scheme for the prototype FERMTOR allows versatile packet level communication between processors.

Granularity of parallelism. The use of the packet communication structure depends upon the choice of processor. Microprogrammed bit-slice processors[9] allow very fast interaction with the P-Bus. General-purpose microprocessors using memory-mapped I/O to access the P-Bus registers are significantly slower. Indeed, it requires roughly 12 microprocessor instructions (at 3 to 4 microseconds each) simply to load the P-Bus latches and request a transmission. It takes a great deal more time to determine the contents of that packet.

A microprogrammed machine could access the P-Bus on an instruction basis—i.e., every machine-level instruction would use the P-Bus. But this is not possible for the slower microprocessor elements. In this case we must ensure that each processor uses the P-Bus less frequently. Thus for such microprocessors, the basic unit of parallelism is the *process* and not the instruction. The process is constrained to executing local object code and performing intraprocessor communication at a rate compatible with the P-Bus interface. Later in the article we discuss how to relax these constraints by means of a faster interface.

FERMTOR uses processes as its basic unit of parallelism. The coarse granularity of this parallelism increases the likelihood that a processor is left idle for a long period of time while waiting for another process to finish a calculation. This could happen, for example, when one processor requests data from another that requires significant computation—the requesting processor would be idle during the computation. To make use of this idle time, each processor in FERMTOR is itself multiprocessing. That is, every processor can have several active processes in it so that it is busy as much as possible. While one process awaits data from another processor, other processes can use the microprocessor. The implementation of multiprocessing was done using the concurrent programming language Concurrent Euclid.[19] It is a dialect of Pascal and provides multiprocess synchronization using Hoare's monitors and signals.[18] Note that the programs are compiled individually for each processor, not as one big program for the entire multiprocessor.

Implementing software in a concurrent language has another benefit: the interprocessor communication can be run by separate processes of which the user need have no knowledge.

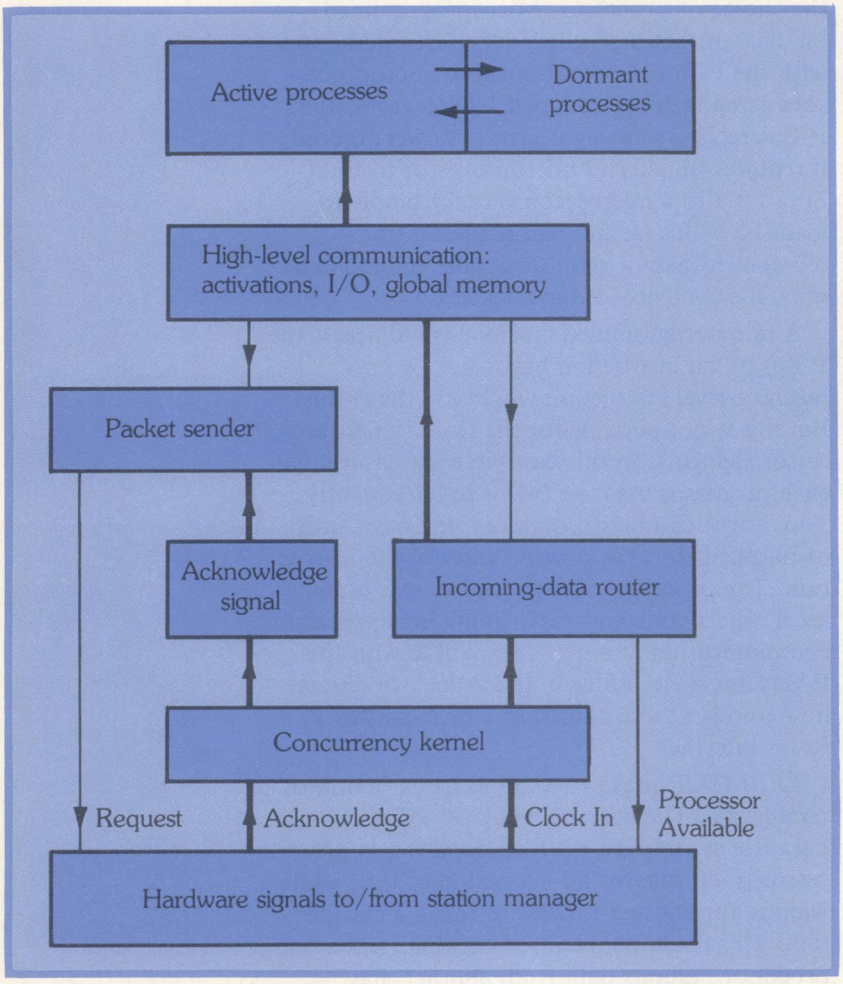


Figure 3. The communication hierarchy along the P-Bus.

Coarse granularity. The decision to choose the process as the granularity of parallelism is a significant one. The choice was between a fine granularity such as instruction-wise parallelism and a coarse granularity such as parallel processes.

Instruction-wise parallelism occurs when a very small amount of computation is done between interprocessor transfers. For example, this could be the amount of computation involved in a typical high-level-language arithmetic statement. In this kind of parallelism, each processor is explicitly told to execute such a statement by a separate scheduling processor. The scheduler communicates with the executing processor over the communication bus. Many statements are scheduled into several processors and ordered into a "chain" of instructions to be sequentially executed. The first processor/instruction is activated, then executed, and that processor activates the second instruction in the chain, and so on.

Instruction-wise parallelism incurs a great deal of overhead per instruction. Loucks[9] investigated this and found scheduling to be a significant bottleneck. Scheduling will always be a problem if the amount of computation per interprocessor transfer bus is small.

Process-wise parallelism occurs when a large amount of computation is performed between interprocessor transfers, such as would be needed to solve a large system of linear equations, for example. Typically, one processor acts as a scheduler, assigning these large pieces of the overall task to slave processors.

Parallel processes greatly reduce the P-Bus traffic because the processors spend much more time calculating between bus transfers than they do with instruction-wise parallelism. In addition, process-wise parallelism makes it relatively more easy to schedule large chunks of processor time.

In FERMTOR, we chose process-wise parallelism. The issue of scheduling is discussed further below.

Note that parallelism granularity is a continuum rather than a discrete choice between coarse and fine. It is an open problem to discover exactly how much computation should be done between interprocessor transfers. The answer is most likely both application- and situation-dependent.

Communication hierarchy. Processes can communicate at many levels. They can simply send a byte of data or transfer an entire file. They can also send messages which control program flow directly. To support these levels of communication there must exist a coherent substructure of software. This idea is not unlike the protocol levels described in the ISO model for Open System Interconnection,[20] commonly used in local-area networks. In FERMTOR, the coupling between processors is much greater than with a LAN, but similar ideas still apply. Figure 3 is a block diagram of the communication hierarchy.

Low-level packet communication. At the lowest level in the hierarchy is the actual hardware that performs the physical data transfer

and handles signalling between the processor and station manager. Here a packet is the basic unit of transaction. The first level of software interface is the *concurrency kernel*. The kernel, which is part of the Concurrent Euclid programming language, implements concurrency within the microprocessor. This includes handling hardware interrupts from the processor's P-Bus interface. The kernel polls the station manager after an interrupt to determine the cause, and then dispatches the associated Euclid process.

The **Ack** (acknowledge) signal from the station manager, shown in Figure 3, indicates that the previously requested P-Bus transfer has been completed. This interrupt is handled by the kernel, which invokes the acknowledge process. The acknowledge process sends a software signal[18] to the *packet sender*, which is then free to initiate another transmission. The packet sender raises the hardware request signal to transmit a packet.

The **Clock In** signal from the station manager indicates that a new packet has arrived for the processor. Again, this interrupt is handled by the kernel, which then invokes the *incoming-data router* process. Any process that expects to receive a packet must inform the incoming-data router of the type of packet it expects.

The incoming-data router keeps a table of all the packet types in the processor, along with the associated processes. When the router receives a packet, it signals the associated process and gives it the packet's contents.

High-level packet communication. In addition to low-level communication, Figure 3, described above, also depicts three kinds of medium- to high-level communication:

1) *Data input/output.* One or more processors are usually designated as the I/O processors, and are dedicated to that purpose. All other processors transmit data (for output) and request data (for input) from the I/O processors.

2) *Global memory.* One or more processors are designated to act as global memory. The memory processor performs four functions:
   - Allocate a block of storage.
   - Deallocate a block of storage.
   - Write to global memory.
   - Read from global memory. The read/write functions are similar to the I/O functions.

3) *Activations.* A process in the processor can be dormant waiting for another calculation to finish. An activation packet will wake that process and tell it where to find the data that it is waiting for; often such data is stored in global memory. Note that this feature has overtones of the data-flow concept.

Scheduling. Scheduling the processing resources of a multiprocessor is a crucial and difficult task. In the prototype FERMTOR, the process itself must be scheduled. This is the task of deciding which process goes to which processor. The schedule is currently static, but could be made dynamic if an efficient method of transporting code were developed (see our discussion of enhancements below). In the present version of FERMTOR, there is no automatic scheduling. The user partitions the processes, attempting to get maximum performance from the processors.

Using Concurrent Euclid, it is a simple matter to determine how much time each processor is idle. This knowledge can help the user to partition a multiprocessor program. Note that the work required to produce a good partition is only justified if the application is going to be used over a long period of time. This style of scheduling is applicable to special-purpose machines that have processors dedicated to permanent tasks.

Implementation

We have constructed a prototype FERMTOR that is sufficiently large to contain all the salient features of the architecture. It has provided a test vehicle to assess the viability of the architecture.

The prototype consists of three stations that are capable of supporting four processors each. We currently have six processors, which can be distributed arbitrarily among stations. Five of the processors are Motorola 6809 general-purpose microprocessors with their own local memory and P-Bus interface hardware. The sixth processor is a PDP-11/34 running the UNIX operating system. A special P-Bus interface to the PDP-11 was constructed, using a standard parallel port on the UNIBUS.

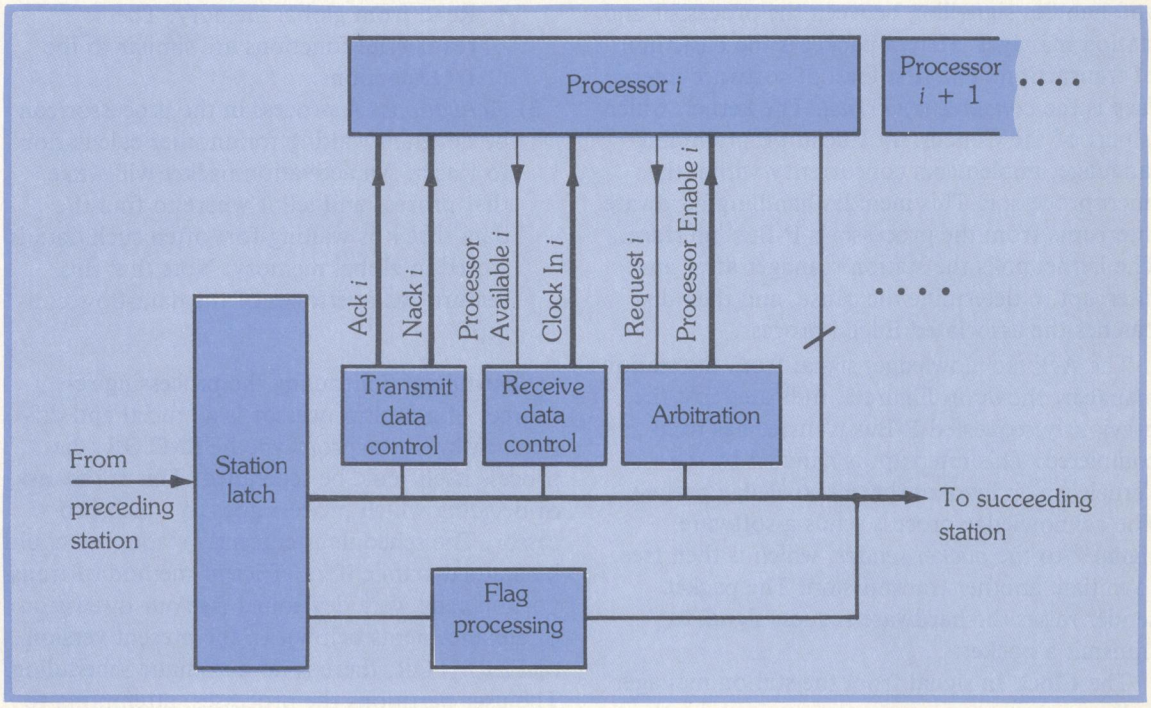


Figure 4. A detailed view of a station manager.

Station manager. A block diagram of the prototype station manager is shown in Figure 4. It consists of four principal parts. The *station latch* holds the data packets transmitted between the stations. The *arbitration unit* decides, every cycle, which packet is enabled onto the local bus. If the incoming ring packet is full, then it takes precedence. If it is empty, then the arbitration unit arbitrates processor requests for the bus. The *receive data control unit* transmits any packets destined for this station to the addressed processor by clocking the bus data into the processor's buffer. It also sends the **Clock In** interrupt signal to the processor. The *transmit data control unit* determines the success or failure of transmissions from this station and informs the sending processor with an **Ack** interrupt or a **Nack** signal.

The P-Bus is synchronously clocked. The maximum speed is determined by the arbitration unit and is roughly 2.5 MHz. A two-phase clock is generated centrally and is distributed to all the station managers. Other timing signals are generated local to each station from this clock.

Processors. Figure 5 is a block diagram of the 6809 processor and its P-Bus interface. The P-Bus interface has two parts: the buffers that contain incoming and outgoing data from the station bus, and the control circuit that handles the following control signals.

- **Request:** an output request to the station manager, indicating that there is data in the output buffer to be transmitted.
- **Processor Available:** an output-status flag indicating that the processor is able to receive another packet. This circuit is made difficult because the processor can be either much slower or much faster than the station manager.
- **Processor Enable:** an input from the station manager that causes the processor to place a requested transmission onto the P-Bus.
- **Clock In:** an input from the station manager that sends a packet from the P-Bus to the processor. This signal interrupts the processor.
- **Acknowledge:** an input from the station manager indicating that the last request was successful. It is wired to cause an interrupt to the processor.
- **No Acknowledge:** an input from the station manager indicating that the last re-

quest was not successful. Currently, this signal is wired to generate a new request. It could also be wired to input directly into the processor so that multiple transmission failures could be detected, and ring faults deduced.

Results and performance

The prototype FERMTOR consists of three stations and six processors. Each station Manager fits on one wire-wrapped board. One 6809 processor is a wire-wrapped prototype, and the other four are implemented as printed circuit boards. There is also an interface board that connects a PDP-11/34 with the P-Bus.

The P-Bus and associated processors have worked error-free in all tests. We have exercised the multiprocessor in a number of ways, testing communication speed alone as well as its multiprocessing capabilities.

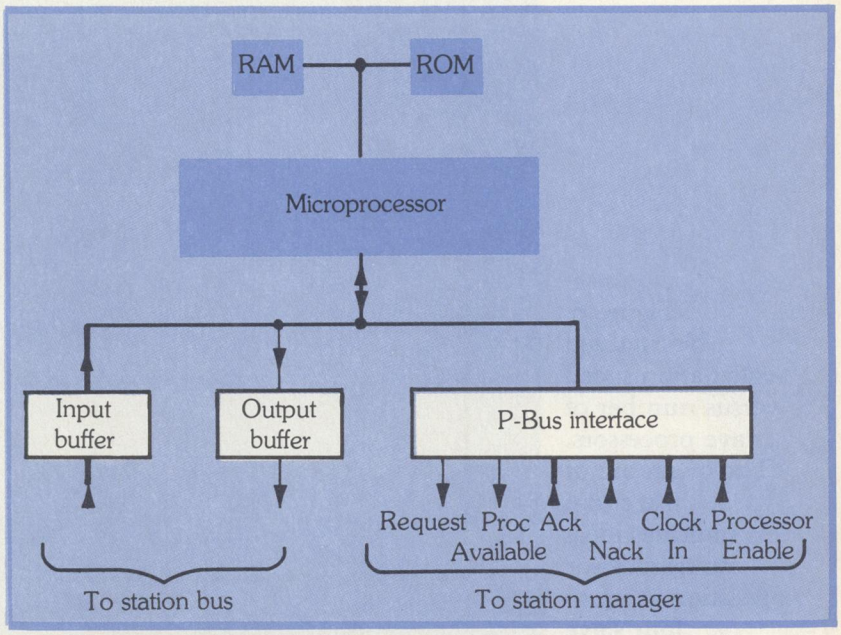


Figure 5. A detailed processor diagram.

P-Bus speed test program. The station manager clock on the prototype FERMTOR is set at 1 MHz. The effective bit-transfer rate of the P-Bus, which carries 24 bits of data, is 24M bits per second. The hardware is capable of supporting a 2.5 MHz clock, and so the maximum P-Bus transfer rate is 60M bits per second. This is the maximum amount of data that can be transmitted around the ring by all of the processors. A higher transfer rate can be achieved if there is a significant amount of intrastation transfers. Loucks [9] gives an analysis of the effective P-Bus transfer rates.

Standard microprocessors that use memory mapped I/O require only a small fraction of this bandwidth, as discussed immediately below. Therefore, the P-Bus is capable of supporting a very large number of such processors, in the range of several hundred per ring.

Transfer rates per microprocessor. P-Bus input/output by the microprocessor is done using latches that are mapped into processor memory. Every P-Bus transfer requires the loading and unloading of these latches, at the same rate as a load/store memory access.

We measured the maximum number of transfers that the microprocessor can perform. For both transmission and reception, the maximum rate is roughly 4000 packets per second. Since the P-Bus can support 2.5 million transfers per second, this suggests that several hundred processors can be supported on a one ring P-Bus.

If a higher per-processor transfer rate is required, the enhancements discussed further below could be implemented.

A simple test. To test the basic multiprocessing capability of FERMTOR, we wrote a simple application program. The program tests a range of integers for primeness. In each slave processor, a program waits for two packets specifying a number range to be tested. It uses the simple algorithm of dividing the number under test by all the odd numbers up to one third of that number and checking for a remainder. This inefficient algorithm makes the calculation highly CPU-intensive. The master (or scheduling) processor sends out the number ranges to the slaves. When a slave determines that a certain number is prime, it transmits that number to the master.

The large amount of calculation required by the test makes P-Bus transfers very infrequent. As a result, FERMTOR exhibits an ideal multiprocessing property: the computation speed is linear with respect to the number of processors.

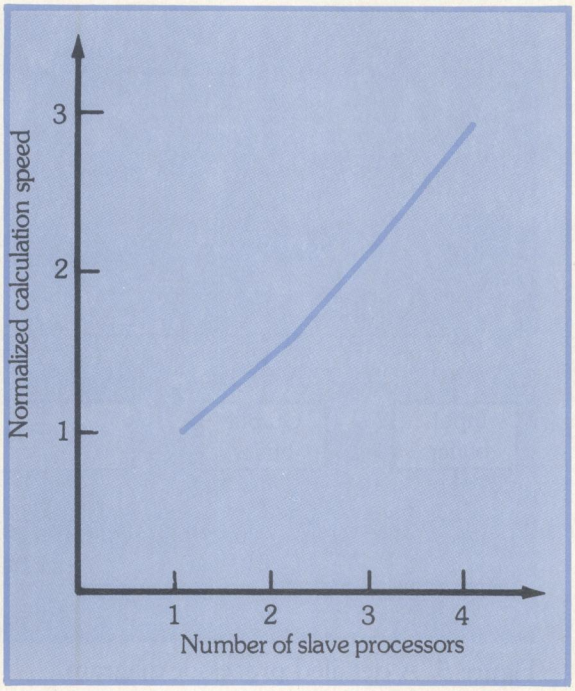


Figure 6. Normalized calculation speed versus number of slave processors. The linear arc of the curve indicates ideal multiprocessor operation at up to four slave processors.

That is, if for two processors (one master and one slave) the time taken is t, then for three, four or five processors it is $t/2$, $t/3$ and $t/4$, respectively. This performance is due entirely to the fact that the processors are loosely coupled.

A closely coupled multiprocessing test. It is generally true that the more closely coupled a multiprocessor architecture allows its processors to be, the more applicable that multiprocessor structure will be. Thus we wished to test FERMTOR with a multiprocessor program that frequently uses the P-Bus.

We again chose the prime number example but adapted the algorithm for more cooperation between processors. We wished to determine all the primes between 1 and some number X. As the system discovers these primes, at first using the same algorithm as above, the master processor records them and distributes them back to the slave processors. Thus the slaves need only divide the number under test by the prime numbers less than one third of the number. Note that if there are not sufficient prime numbers computed to that point, this algorithm again reverts to the former one. The transmission of primes to the master and their subsequent rebroadcast places a much heavier load on the P-Bus than does the simple test.

The size of the number range that each processor tests at a time becomes a factor in this algorithm. The smaller the range, the more frequently the master must present a new range, but the larger the range, the longer it takes to transmit discovered primes to all the slaves.

The results obtained with this closely coupled example are shown in Figure 6. The figure is a plot of normalized calculation speed versus the number of slave processors. The normalized calculation speed for n slave processors is found by dividing the calculation time of one slave processor (for the entire task) by the time required for n processors to do the entire task. Ideally, the normalized calculation speed increases linearly with the number of processors. The curve in Figure 6, which is indeed linear, demonstrates that we achieved the ideal multiprocessor result for up to four processors. The difference between this example and the less closely coupled example above is that the slope of the line is less than 1. For this example the slope is 0.64. This is due to the interprocessor communication overhead. There is increased interprocessor communication when more than one slave processor is calculating. This overhead is constant per processor, as indicated by the linear curve in Figure 6.

It is interesting to note how the interprocessor communication was optimized in this example. By measuring how much each processor was idle, and determining why it was idle, it was possible for us to discover bottlenecks in the multiprocessor program. For example, we found that the packet transmission by the scheduler was a bottleneck. Instead of having the scheduler transmit discovered primes to all the slaves, we altered it to transmit to only one slave. That slave then forwards the number to the next slave and so on until all slaves receive the prime number. Thus in the four-slave case, the scheduler need only send out one packet per prime instead of four. This eliminates the packet transmission bottleneck in the scheduler.

This kind of optimization produces a constant interprocessor-communication overhead per processor. The technique of idle-time measurement tells not only how idle each processor is, but exactly why it is idle. Provided a task can be divided

into many parts, then this idle-time analysis permits *fine tuning* to get good utilization of the multiprocessor.

Applications

We propose FERMTOR as a multiprocessor architecture tunable to other special-purpose applications. We are investigating three such applications: simulation, VLSI layout, and signal processing.

Simulation. Computer simulation is a task well-suited to the application of parallelism. Many physical systems that are simulated by computer are inherently parallel. At the same time, simulation may require many computation hours to complete, so that parallelism can reduce this considerably. Intuitively, it appears that FERMTOR would lend itself to implementing a language like SIMULA.[21] Some work has already been done on using MIMD systems and SIMULA.[22] In FERMTOR, the flow of data (such as packets around a ring) and control (such as activations) is very similar to the communication between SIMULA classes.

We are developing a few simple constructs to add to Concurrent Euclid to aid in simulation. Processes are activated by packets containing data, upon which the process operates. There are two kinds of processes:

- *Nonqueued processes,* in which any number of activations can be active at a given time (using reentrant code); and
- *Queued processes* (representing a single shared resource), in which only one activation is active at a given time. Subsequent activations are placed in a FIFO queue.

Perhaps the most difficult problem in parallel simulation is that of simulation-time synchronization. Every processor must agree that a certain piece of data or state is associated with the same simulation time. This may mean that all processors must wait for the slowest process to be completed before the next time period begins. An alternative is the use of *time stamps.*[23, 24] Here any message of data or activation contains a stamp indicating the time at which the sending processor is operating. The receiving processor cannot use the message until it reaches that time. Some synchronization problems remain; these are discussed by Reynolds.[24] The work on parallel simulation is in its infancy and requires a great deal of work on compiler generation and the issue of synchronization.

VLSI layout. The automatic layout of VLSI circuits, already a time-consuming task, promises to grow evermore compute-bound as the density of VLSI circuits increases. The partitioning, placement, and routing of VLSI circuits is a prime candidate for the application of multiprocessing. Much of the existing work concentrates on the use of SIMD structures for placement and routing.[25, 26, 27] We intend to use MIMD structures for dealing with layout problems. MIMD structures can make better use of existing layout algorithms than SIMD architectures because of the greater similarity between MIMD multiprocessors and uniprocessors. For example, the natural hierarchy of VLSI circuits can be used as a technique for partitioning the circuit, so that individual processors can do traditional placement on a small section of the circuit. We are also investigating the implementation of an algorithm like Soukup's Global Router.[28] Here one processor would be responsible for one interconnection net, giving the potential of a large amount of parallelism.

Signal processing and synthesis. The prototype P-Bus is currently being used to connect a number of Motorola 6809 and TMS 320 processor boards for the purposes of both signal processing and signal synthesis.[29] The TMS 320 is a high-speed, limited-memory processor. Several TMS 320s are being used with the P-Bus to cascade high-speed calculations for good-quality digital filters and for music synthesis. The versatility and tunability of the P-Bus permits easy addition of the TMS 320 processors.

Enhancements

We are currently working on several aspects of FERMTOR. It is clear that, using process-wise parallelism, faster individual processors will speed up the multiprocessor in a cost-effective way. We are now implementing a FERMTOR with National 32016 system microprocessors.[30]

To speed up the interprocessor communication, we are using DMA to implement two features:

- *Implicit shared memory*. Part of the local processor's memory will be mapped to another processor's memory, and any access to that memory will automatically use the P-Bus to get the remote data. This will eliminate explicit processor transfer requests.
- *Block transfer requests*. Hardware will use DMA to transfer large blocks of data and code between processors very quickly. Here the processor will explicitly initiate the request but will not be involved in each individual packet transfer. This is similar to DMA transfer between magnetic disks and processors.

These new features will be added to the 32016-based processors through the MULTIBUS on each processor.

Future application development work will concentrate on the VLSI layout machine. This promises to be an extremely useful application of FERMTOR.

We have presented a multiprocessor architecture that can be easily tuned to fit many applications. Through our experiments on the prototype, we found the potential for achieving a great degree of parallelism, provided a task is well partitioned. We are continuing to develop applications such as simulation and VLSI layout. The FERMTOR architecture provides a viable structure for developing powerful machines using standard microprocessor components.

References

1. H. Morkoc and P.M. Solomon, "The HEMT, A Superfast Transistor," *IEEE Spectrum,* Vol. 21, No. 2, Feb. 1984, pp. 28-35.
2. J.H. Hennessey, et al., "Design of a High Performance VLSI Processor," *Proc. VLSI 83,* pp. 1-21.
3. D.A. Patterson and C.H. Sequin, "A VLSI RISC," *Computer,* Vol. 15, No. 9, Sept. 1982, pp. 8-21.
4. H.T. Kung, "Why Systolic Architectures?" *Computer,* Vol. 15, No. 1, Jan. 1982, pp. 37-46.
5. G.M. Barnes, et al., "The ILLIAC IV Computer," *IEEE Trans. Computers,* Vol. C-17, No. 8, Aug. 1968, pp. 746-757.
6. R.A. Lougheed and D.L. McCubbrey, "The Cytocomputer: A Practical Pipelined Image Processor," *Proc. 7th Ann. Symp. Computer Architecture,* May 1980, pp. 271-277.
7. P.B. Berra and E. Oliver, "The Role of Associative Array Processors In Data Base Machine Architectures," *Computer,* Vol. 12, No. 3, March 1979, pp. 53-61.
8. W.M. Loucks and Z.G. Vranesic, "FERMTOR: A Flexible Extendible Range Multiprocessor," *Proc. Canadian Information Processing Soc. Conf.,* 1980, pp. 134-145.
9. W.M. Loucks, "FERMTOR: A Flexible Extendible Range Multiprocessor," PhD thesis, University of Toronto, 1980.
10. J.S. Rose, "An Implementation of FERMTOR: A Flexible Extendible Range Multiprocessor," MASc thesis, University of Toronto, 1982.
11. J. Rumbaugh, "A Data Flow Multiprocessor," *IEEE Trans. Computers,* Vol. C-26, No. 2, Feb. 1977, pp. 138-146.
12. J.B. Dennis, "The Varieties Of Data Flow Computers," *Proc. Ist Int'l Conf. Distributed Computing Systems,* Oct. 1979, pp. 430-439.
13. I. Watson and J. Gurd, "A Practical Data-Flow Computer," *Computer,* Vol. 15, No. 2, Feb. 1982, pp. 51-57.
14. E.P. Farrel, N. Ghani, and P.C. Treleaven, "A Concurrent Computer Architecture and a Ring Based Implementation," *Proc. 6th Ann. Symp. Computer Architecture,* April 1979, pp. 1-10.
15. R.J. Swann, S.H. Fuller, and D.P. Siewiorek, "CM*—A Modular Multiprocessor," *AFIPS Conf. Proc.,* Vol. 46, 1977, pp. 637-644.
16. L. Stringa, "EMMA: An Industrial Experience On Large Multiprocessing Architectures," *Proc. 10th Symp. Computer Architecture,* June 1983, pp. 326-333.
17. K.G. Shin, Y-H Lee, and J. Sasidhar, "Design of HM2P—A Hierarchical Multimicroprocessor for General Applications," *IEEE Trans. Computers,* Vol. C-31, No. 11, Nov. 1982, pp. 1045-1053.
18. C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM,* Vol. 21, No. 8, Aug. 1978, pp. 666-677.
19. R.C. Holt and J.R. Cordy, *Specification of Concurrent Euclid,* Computer Systems Research Group Technical Report CSRG-133, University of Toronto, Aug. 1981.
20. H. Zimmerman, "OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection," *IEEE Trans. Communications,* Vol. COM-28, April 1980, p. 425-432.

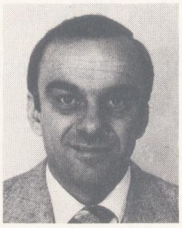
21. G. G. Birtwistle, L. Enderin, M. Ohlin, and J. Palme, *DEC System 10 SIMULA Language Handbook, Part I,* Swedish National Defence Research Institute and Norwegian Computing Centre, Stockholm, Sweden, 1978.
22. P. Georgiadis, M.P. Papazoglow, and D.G. Maritsas, "Towards a Parallel SIMULA Machine," *Proc. 8th Ann. Symp. Computer Architecture,* May 1981, pp. 263-278.
23. L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Comm. ACM,* Vol. 21, No. 77, July 1978, pp. 558-565.
24. P.F. Reynolds, "A Shared Resource Algorithm for Distributed Simulation," *Proc. 9th Ann. Symp. Computer Architecture,* April 1982.
25. M.A. Breuer and K. Shamsa, "A Hardware Router," *J. Digital Systems,* Vol. IV, Issue 4, 1981, pp. 393-408.
26. R.A. Rutenbar, T.N. Mudge, and D.E. Atkins, "A Class of Cellular Architectures to Support Physical Design Automation," *IEEE Trans. Computer-Aided Design,* Vol. CAD-3, No. 4, Oct. 1984, pp. 264-278.
27. K. Ueda, T. Komatsubara, and T. Hosaka, "A Parallel Processing Approach For Logic Module Placement," *IEEE Trans. Computer-Aided Design,* Vol. CAD-2, No. 1, Jan. 1983, pp. 39-47.
28. J. Soukup, "Global Router," *Proc. 16th Design Automation Conference,* June 1979, pp. 481-484.
29. J. Kitamura, MASc thesis, University of Toronto, in progress.
30. *NS32000 Data Book,* National Semiconductor Corporation, 2900 Semiconductor Drive, Santa Clara, CA 95051.

Jonathan Rose is currently working on his PhD in electrical engineering at the University of Toronto, Ottawa, Canada. His research interests include multiprocessing hardware and operating systems and their application to automatic VLSI layout. During the summer of 1983, Rose was with Bell-Northern Research, Ltd, Ottawa, in the Integrated Circuits CAD/CAM group.

Rose received the BASc in engineering science in 1980 and the MASc in electrical engineering in 1982 from the University of Toronto. A junior fellow of Massey College of the University of Toronto, he is a member of the IEEE and the ACM.

Wayne Loucks is an assistant professor in the Department of Electrical Engineering at the University of Toronto. Prior to joining the university staff, he was a research associate involved in the development of a local-area computer network. His research interests are in computer architecture, multiprocessors, processing arrays, and LANs.

Loucks received the BASc in 1975 from the University of Waterloo, Ontario, and the MASc and PhD degrees from the University of Toronto in 1977 and 1980, respectively. He is a member of the IEEE and the ACM.



Zvonko Vranesic received the BASc, MASc, and PhD degrees from the University of Toronto in 1963, 1966, and 1968, respectively. In 1968 he joined the faculty of the Departments Electrical Engineering and Computer Science at the University of Toronto. During academic year 1984-85 he was on research leave at the Institut de Programmation, Universite de Paris 6.

Vranesic's research interests include computer architecture, multiprocessor systems, fault-tolerant computing, local-area networks, and many-valued switching systems. He has served as chairman and technical program chairman for the International Symposium on Multiple-Valued Logic. Vranesic is a senior member of the IEEE.

Questions about this article may be addressed to the authors at the Department of Electrical Engineering, University of Toronto, Toronto, Canada M5S 1A4.