A new minicomputer/multiprocessor for the ARPA network*

by F. E. HEART, S. M. ORNSTEIN, W. R. CROWTHER, and W. B. BARKER

Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

INTRODUCTION

Since the early years of the digital computer era, there has been a continuing attempt to gain processing power by organizing hardware processors so as to achieve some form of parallel operation.[1,2] One important thread has been the use of an array of processors to allow a single control stream to operate simultaneously on a multiplicity of data streams; the most ambitious effort in this direction has been the ILLIAC IV project.[3,4] Another important thread has been the partitioning of problems so that several control streams can operate in parallel. Often functions have been unloaded from a central processor onto various specialized processors; examples include data channels, display processors, front-end communication processors, on-line data preprocessors—in fact, I/O processors of all sorts. Similarly, dual processor systems have been used to provide load sharing and increased reliability. Still another thread has been the construction of pipeline systems in which sub-pieces of a single (generally large) processor work in parallel on successive phases of a problem.[5] In some of these pipeline approaches the parallelism is "hidden" and the user considers only a single control stream.

In recent years, as minicomputers have proliferated, groups of identical small machines have been connected together and jobs partitioned quite grossly among them. Most recently, our group and several others have been investigating this avenue further, attempting to reduce the specialization of the processors in order to employ independent processors with independent control streams in a cooperative and "equal" fashion.[6,7,8]

This paper describes a new minicomputer/multiprocessor architecture for which a fourteen-processor prototype is now (February 1973) being constructed. The hardware design and the software organization include many novel features, and the system may offer significant advantages in modularity and cost/performance. The system contains an expandable number of identical processors, each with some "private" memory; an expandable amount of "shared" memory to which all processors have equal access; and an expandable amount of I/O interface equipment, controllable by any processor. The system achieves unusual modularity and reliability by making all processors equivalent, so that any processor may perform any system task; thus systems can be easily configured to meet the throughput requirements of a particular job. The scheme for interconnecting processors, memories, and I/O is also modular, permitting interconnection cost to vary smoothly with system size. There is no "executive" and each processor determines its own task allocation.

A key issue throughout most of the attempts at parallel organization has been the difficulty of partitioning problems in such a way that the resulting computer program(s) can really take advantage of the parallel organization. This issue is raised in its most serious form when the parallel machine is expected to work well on a great diversity of problems as, for example, in a time-sharing system. Our machine design has been developed under the highly favorable circumstances that (1) the initial application, and a prior software implementation in a standard machine, was well understood; (2) the initial application lent itself to fragmentation into parallel structures; and (3) the design would be deemed successful if it handled only that one application in a meritorious fashion. However, we now believe that the design is advantageous for many other important applications as well and that it may herald a broadly useful new way to achieve increased performance and reliability.

The machine has been designed to serve initially as a modular switching node for the ARPA Network[9] and, in the following section, we briefly describe the ARPA Network application and the requirements that the network imposed upon the machine design. In subsequent sections we discuss our choice of minicomputer, describe our system design in some detail, discuss certain of the more interesting characteristics of multiprocessor behavior, and summarize our present status and plans for the near future.

* *This work was sponsored by the Advanced Research Projects Agency under Contracts DAHC15-69-C-0179 and FO8606-73-6-0027.*

ARPA NETWORK REQUIREMENTS

The ARPA Network, a nationwide interconnection of computers and high bandwidth (50 Kb) communication circuits, has grown during the past four years to include over 35 sites, with more than one computer at many sites. The computers at each site, called Hosts, obtain access to the net via a small communications processor known as an Interface Message Processor or IMP.[10] In order to permit groups without their own computer facility to access this powerful set of computer resources, a version of the IMP called a Terminal IMP allows, in addition, attachment of up to 63 local or remote terminals of a wide range of types.[11]

As a considerable simplification, the job to be handled by an IMP is that of a communications processor. Arriving messages must pass through an error control algorithm, be inspected to some degree (e.g., for destination), and generally be directed out onto some other line. Some incoming messages (e.g., routing control messages) must be constructed or digested directly by the IMP. The IMP must also concern itself with flow control, message assembly and sequencing, performance and flow monitoring, Host status, line and interface testing, and many other housekeeping functions. To perform these functions an IMP requires memory both for program and for message buffers, processing power for executing the program, and I/O units of various sorts for connecting to a variety of lines and devices. The original IMP, built around a Honeywell 516 processor with a 1 μs cycle time, could handle approximately three-quarters of a megabit per second of full duplex communications traffic. A later, smaller and cheaper (Honeywell 316) version handles about two-thirds as much traffic.

As the network has grown and as usage has increased, a number of demands for improvement have led to the need for a new "line" of IMP machines. Our intent is to provide a modular arrangement of flexible hardware from which it will be possible to construct both smaller and less expensive IMPs as well as far more powerful IMPs. An important specific objective is to obtain an IMP whose communications bandwidth could be at least an order of magnitude greater than the 516 IMP; such a high speed IMP would permit the direct connection of satellite circuits or land T-carrier circuits operating at approximately 1.3 megabits/second.

It is also desirable to improve the present IMP design in a number of other areas, as follows.

- Expandability of I/O: The present IMPs permit connection to a total of only seven high-speed circuits and/or Host computers. We would like to permit a much greater fanout so that an IMP might be connected to as many as 20 or more Host computers or to hundreds of terminals. This means that the number of interface units should be expandable over a wide range.
- Modularity: A number of groups have wished to make a network connection from a single Host at a considerable distance (miles) from the nearest IMP. We feel that such Hosts should be locally connected to a very small IMP in order to preserve consistency and standardization throughout the network. Therefore, a goal of this new hardware effort is the provision of a small and inexpensive but compatible IMP which could serve to connect a single, distant spur Host.
- Expandability of Memory: The new line of equipment is required for use in connection with satellite links (or longer faster links in general) and must therefore be able to expand its memory easily to provide the much greater buffer storage requirements of such links.
- Reliability: The new line of processors should be more reliable than the existing IMPs and ought to permit better self-diagnosis and simple isolation and replacement of failing units.

Of the requirements posed by the ARPA Network application, the most central was to obtain an order-of-magnitude traffic bandwidth improvement. We first considered meeting this requirement with highly specialized hardware, but the need to allow evolution of the communications algorithms, as well as the "bookkeeping" nature of much of the IMP task, militate against hardwired approaches and require the flexibility of a stored program computer. Thus we need a machine with an effective cycle time of 100 nanoseconds, a factor of ten faster than the present 1 μs IMP. Realizing that a single very fast and powerful machine would be difficult to build and would not give us compatible machines with a wide spectrum of performance, we began to consider the possibility of a minicomputer/multiprocessor in order to achieve the flexibility, reliability, and effective bandwidth required.

With the idea of a multiprocessor in mind we considered the IMP algorithm to determine which parts were inherently serial in nature and which could proceed in parallel. It seemed difficult to process a single message in a parallel fashion: the job was already relatively short and intimately coupled to I/O interfaces. However, there was much less serial coupling between the processing of separate messages from the same phone line and no coupling at all between messages from different phone lines. We thus envisage many processors, each at work on a separate message, with the number of processors carefully matched to the number of messages we expect to encounter in the time it takes one processor to deal with one message. With this simple image there seems to be no inherent limit to the parallelism we can achieve—the ultimate limit would be set by the size of the multiprocessor we can build.

CHOICE OF THE PROCESSOR

In designing a multiprocessor for the IMP application, we found ourselves iteratively exploring two related but distinct issues. First, assuming that the problem of interconnection could be solved, what minicomputer would be

a sensible choice from the price/performance and physical points of view? Second, and much harder: for any specific machine, how did the CPU talk to memory, how would multiple CPUs, memories, and I/O be interconnected to form a system, and how would the program be organized?

Since the program for the existing IMPs was well understood, it was possible to identify key sections of that program which consumed the majority of the processing bandwidth. Then, for each sensible minicomputer choice, we could ask how many CPUs of this type would be needed to provide an effective 100 nanosecond cycle time; and given a price list, physical data, and a modest amount of design effort, we could define the physical structure and the price of the resulting multiprocessor. With this general approach, we examined the internal design of about a dozen machines, and actually wrote the key code in many cases. Using the fastest available minicomputers it was possible to arrive at configurations with only three or four processors; using the slowest choices, systems with 20 CPUs or more were required.

If we defer the interconnection and contention problems for a moment, it is interesting to note that "slow and cheap" may win over "fast and expensive" in this kind of multiprocessor competition to achieve a stated processing bandwidth. This is an especially happy situation if, as in our case, a spectrum of configurations is needed, including a very tiny cheap version.

In considering which minicomputer might be most easily adaptable to a multiprocessor structure, the internal communication between the processor and its memory was of primary concern. Several years ago machines were introduced which combined memory and I/O busses into a single bus. As part of this step, registers within the devices (pointers, status and control registers, and the like) were made to look like memory cells so that they and the memory could be referenced in a homogeneous manner. This structure forms a very clean and attractive architecture in which any unit can bid to become master of the bus in order to communicate with any other desired unit. One of the important features of this structure is that it made memory accessing "public"; the interface to the memory had to become asynchronous, cleanly isolable electrically and mechanically, and well documented and stable. A characteristic of this architecture is that all references between units are time multiplexed onto a single bus. Conflicts for bus usage therefore establish an ultimate upper bound on overall performance, and attempts to speed up the bus eventually run into serious problems in arbitration.[12]

In 1972 a new processor—the Lockheed SUE[13]—was introduced which follows the single bus philosophy but carries it an important step further by removing the bus arbitration logic to a module separate from the processor. This step permits one to consider configurations embodying multiple processors and multiple memories as well as I/O on a single bus. The SUE CPU is a compact, relatively inexpensive (approximately $600 in quantity), quite slow processor with a microcoded inner structure. This slowness can be compensated for by simply doubling or trebling the number of processors on the bus; performance is limited largely by the speed of the bus. With this bus architecture it becomes attractive to visualize multibus systems with a "bus coupling" mechanism to allow devices on one bus to access devices on other busses.

Similar approaches can be implemented with varying degrees of difficulty in systems with other bus structures, and we examined several approaches in some detail for those processors whose cost/performance was attractive. Rather fortuitously, the minicomputer which exhibited the most attractive bus architecture also was extremely attractive in terms of cost/performance and physical characteristics. This machine, the Lockheed SUE, would require fourteen processors to achieve the effective 100 nanosecond cycle time, and we embarked on the detailed design of our multiprocessor on that basis.

SYSTEM DESIGN

Although our design permits systems of widely varying size and performance, in the interest of clarity we will describe that design in terms of the particular prototype now under construction. Our overall design is represented in Figure 1. We require fourteen SUE processors to obtain the necessary processing bandwidth, and we estimate that 32K words of memory will be required for a complete copy of the operational program and the necessary communication buffer storage. The I/O arrangements must allow easy connection of all the communications interfaces, appropriate to the IMP job (modem interfaces, Host interfaces, terminal interfaces) as well as standard peripherals and any special devices appropriate to the multiprocessor nature of the system.

Some of the basic SUE characteristics are listed in Table I. From a physical point of view, the SUE chassis represents the basic construction unit; it incorporates a printed circuit back plane which forms the bus into which 24 cards may be plugged. From a logical point of view this bus simply provides a common connection between all

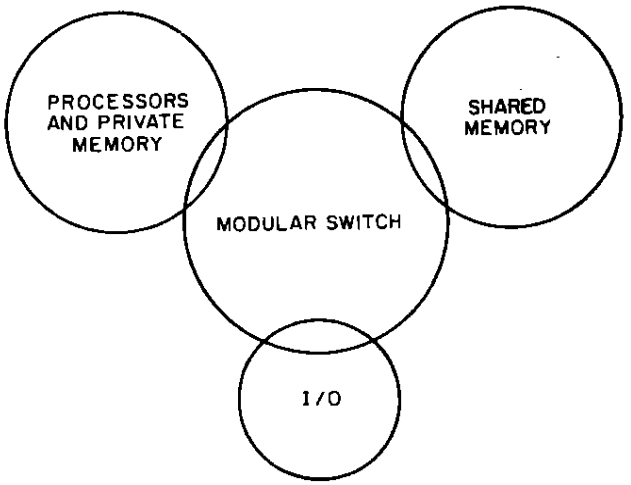

Figure 1—System structure

TABLE I—SUE Characteristics

16-bit word
8 General Registers
△3.7 μs add or load time
Microcoded
Two words/instruction typical
8-½"×19"×18" chassis
64K bytes addressable by a single instruction
~$3K for: 1 CPU+4K Memory+Power, Rack, etc.
200 ns minimum bus cycle time
850 ns memory cycle time
425 ns memory access time

units plugged into the chassis. We are using these chassis for the entire system: processor, memory, and I/O. All specially designed cards as well as all Lockheed-provided modules plug into these bus chassis. With this hardware, the terms "bus" and "chassis" are used somewhat interchangeably, but we will commonly call this standard building unit a "bus." Each bus requires one card which performs arbitration. A bus can be logically extended (via a bus extender unit) to a second bus if additional card space is required; in such a case, a single bus arbiter controls access to the entire extended bus.

We can build a small multiprocessor just by plugging several processors and memories (and I/O) into a single bus. For larger systems we quickly exceed the bandwidth capability of a single bus and we are forced to multi-bus architecture. Then, from a construction viewpoint, our multiprocessor design involves assigning processors, memories and I/O units to busses in a sensible manner and designing a switching arrangement to permit interconnection of all the busses. Of course, the superficial simplicity of this construction viewpoint completely hides the many difficult problems of multiprocessor system design; we will try to deal with some of those issues in the following sections.

Resources

A central notion in a parallel system is the idea of a "resource," which we define to mean a part of the system needed by more than one of the parallel users and therefore a possible source of contention. The three basic hardware resources are the memories, the I/O, and the processors. It is useful to consider the memories, furthermore, as a collection of resources of quite different character: a program, queues and variables of a global nature, local variables, and large areas of buffer storage.

The basic idea of a multiprocessor is to provide multiple copies of the vital resources in the hope that the algorithm can run faster by using them in parallel. The number of copies of the resource which are required to allow concurrent operation is determined by the speed of the resource and the frequency with which it is used. An additional advantage of multiple copies is reliability: if a system contains a few spare copies of all resources, it can continue to operate when one copy breaks.

It may seem peculiar to think of a processor as a resource, but in fact in our system the parallel parts of the algorithm compete with each other for a processor on which to run. We take the view that all processors shall be identical and equal, and we go to some trouble to insure that this is in fact so. As a consequence no single processor is of vital importance, and we can change the number of processors at will. A later section will describe how the processors coordinate to get the job done without a master of some sort.

Processor busses

A SUE bus can physically and logically support up to four processors. As more processors are added to a bus, the contention for the bus increases, and the performance increment per processor drops; but the effective cost per processor also drops, since the cost for the chassis, power supply, bus arbitration, etc., is amortized over the number of processors.

Roughly speaking, using two processors per bus loses almost nothing in processor performance, using three processors per bus loses significant efficiency, and adding a fourth processor gains less than half an "effective processor." After careful examination of the logical, economic and physical aspects of this choice, we decided to use two processors per processor bus, and we thus require seven processor busses in our initial multiprocessor system.

The next question was how the processors should access the program. In our application, some parts of the program are run very frequently and other parts are run far less frequently. This fact allows a significant advantage to be gained by the use of private memory. When a processor makes access to shared memory via the switching arrangement, that access will incur delays due to contention and delays introduced by the intervening switch. We therefore decided to use a 4K local memory with each processor on its bus to allow faster local access to the frequently run code; these local memories all typically contain the same code. With this configuration and in our application, the ratio of accesses to local versus shared memory is better than three to one. This not only reduces contention delays for access to the shared memory but also cuts the number of accesses which suffer the delays.

The final configuration of a processor bus is shown in Figure 2(a). The units marked "Bus Coupler" have to do with our multiprocessor switching arrangement, which will be discussed below.

*Shared memory busses**

The shared memory of our multiprocessor is intended to contain a copy of the program as well as considerable storage space for message buffering, global variables, etc. Application-dependent considerations led us to select a

* *The terms "I/O bus" and "memory bus" as used here and henceforth are not the same as conventional I/O and memory busses.*

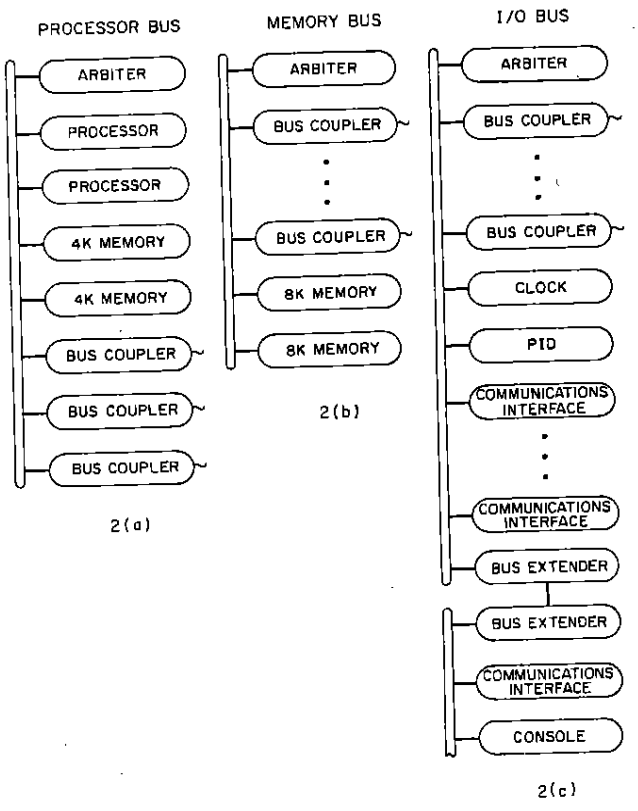

Figure 2—Bus structures

32K memory, but it is possible to configure this memory on a single bus or to divide the memory onto several busses. We first concluded that four logical memory units would be appropriate in order to reduce processor contention to an acceptable level. Then, since the bus is considerably faster than the memories, it is feasible to place two logical memory elements on a single bus with almost no interference. Thus, we are planning two memory busses in the initial multiprocessor; the configuration of a common memory bus is shown in Figure 2(b).

I/O busses

The I/O system of the multiprocessor employs standard SUE busses with standard bus arbitration units on those busses. Into the bus will be plugged cards for each of the various types of I/O interfaces that are required, including interfaces for modems, terminals, Host computers, etc., as well as interfaces for standard peripherals. Our initial system has a single I/O bus and Figure 2(c) shows its configuration; the specialized units shown (a "Clock" and "Pseudo Interrupt Device") are system-wide resources that are used to control the operation of the multiprocessor. The I/O bus will also be the access route for the multiprocessor console; we plan to use a standard alphanumeric display terminal which can be driven by code in any processor, and no conventional consoles will be used.

Interconnection system

Our prototype multiprocessor is now seen to contain seven processor busses, two shared memory busses and an I/O bus. To adhere to our requirement that all processors must be equal and able to perform any system task, these busses must be connected so that all processors can access all shared memory, so that I/O can be fed to and from shared memory, and so that any of the processors may control the operation and sense the status of any I/O unit.

A *distributed* inter-communication scheme was chosen in the interest of expandability, reliability, and design simplicity. The atom of this scheme is called a Bus Coupler, and consists of two cards and an interconnecting cable. In making connections between processors and shared memory, one card plugs into a shared memory bus, where it will request cycles of the memory; the other card plugs into a processor's bus, where it looks like memory. When the processor requests a cycle within the address range which the Bus Coupler recognizes, a request is sent down the cable to the memory end, which then starts contending for the shared memory bus. When selected, it requests the desired cycle of the shared memory. The memory returns the desired information to the Bus Coupler, which then provides it to the requesting processor, which, except for an additional delay, does not know that the memory was not on its own bus. Note that the memory access arbitration inherent in any memory switching arrangement is handled by the SUE Bus Arbiter controlling the shared memory bus, while the Bus Coupler itself is conceptually straightforward.

One additional feature of the Bus Coupler is that it does address mapping. Since a processor can address only 64K bytes (16 bit address), and since we wished to permit multiprocessor configurations with up to 1024K bytes (20 bit address) of shared memory, a mechanism for address expansion is required. The Bus Coupler provides four independent 8K byte windows into shared memory. The processor can load registers in the Bus Coupler which provide the high-order bits of the shared memory address for each of the four windows.

Given a Bus Coupler connecting each processor bus to each shared-memory bus, all processors can access all shared memory. I/O devices which do direct memory transfers must also access these shared memories. These I/O devices are plugged into as many I/O busses as are required to handle the bandwidth involved, and bus couplers then connect each I/O bus to each memory bus. Similarly, I/O devices also need to respond to processor requests for action or information; in this regard, the I/O devices act like memories and Bus Couplers are again used to connect each processor bus to each I/O bus. The path between processor busses and I/O busses is also used in a more sophisticated fashion to allow processors to examine and control other processors; this subject is described in a later section.

The resulting system is shown in Figure 3. One is struck by the number of bus couplers: P*I+I*M+P*M bus couplers are required for a system with P processor bus-

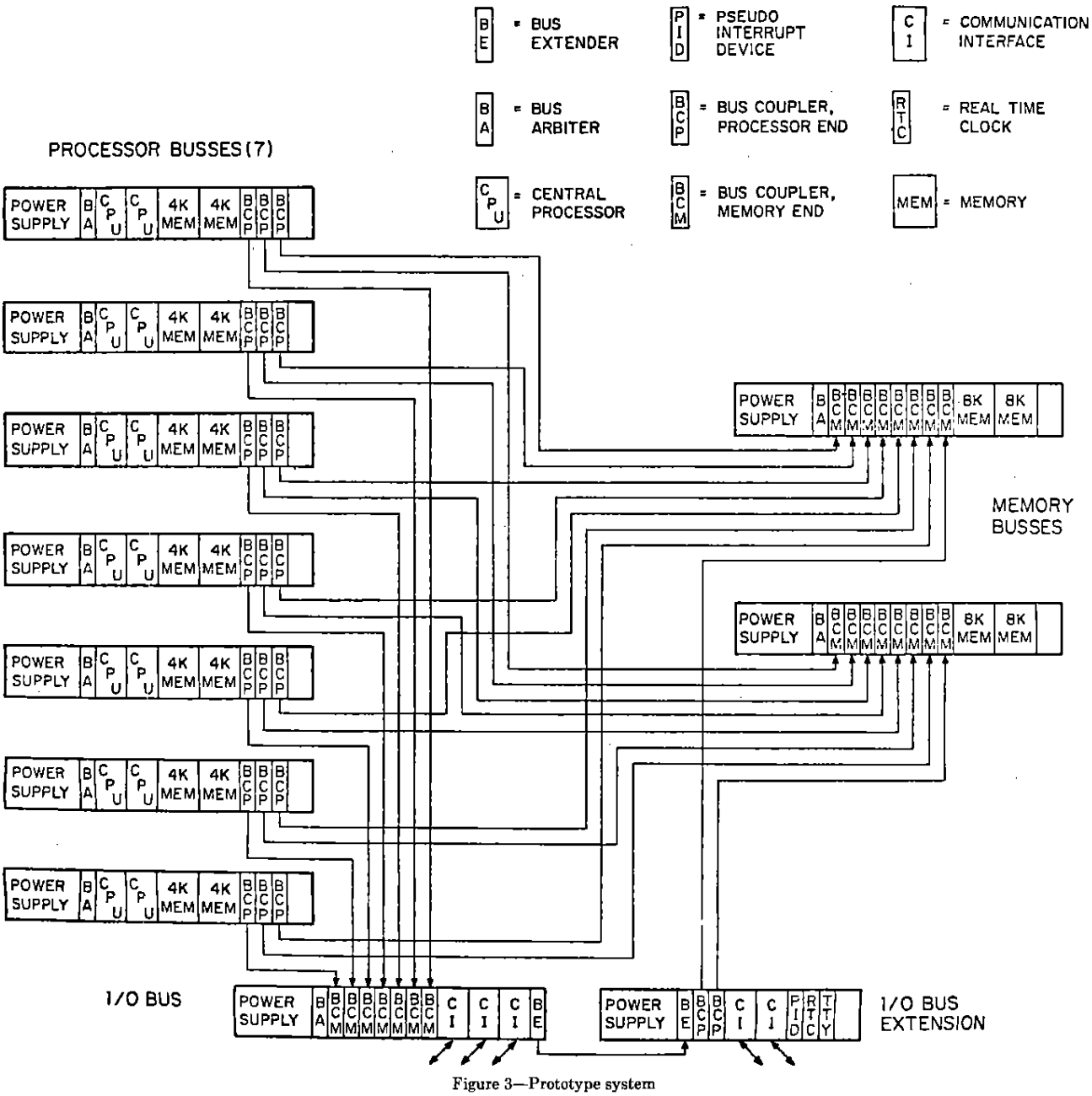

Figure 3—Prototype system

ses, I I/O busses, and M memory busses. In the case of our initial multiprocessor, 23 are needed.

This modular interconnection approach clearly permits great flexibility in the number and configuration of busses, and allows interconnection cost to vary smoothly with system size. We believe that this modular interconnection scheme also permits a complex hierarchical arrangement of busses. Actually the system exhibits a pronounced hierarchical structure already. A processor accesses the local memory when it needs instructions or local variables. Two such processor-memory combinations form a dual processor, which can be regarded as a unit and which needs access to shared resources, such as global variables, free buffers, and I/O interfaces. When one copy of a resource can only support a limited number of users, it seems sensible to provide only the corresponding limited number of connections. If a multiprocessor of this type were to grow larger, the physical number of bus couplers as well as increasing contention problems might not permit the connection of each processor to all of common memory, but might instead require a multi-level structure where groups of processors were connected to an

intermediate level bus which was in turn connected to a centralized common memory. We have not explored this domain but feel it is an interesting area for future work.

MULTIPROCESSOR BEHAVIOR

Until the processors interact, a multiprocessor is a number of independent single processor systems: it is the interaction which poses the conceptual as well as the practical problems. If the various processors spend their time waiting for each other, the system degrades to a single processor equivalent; if they can usefully run concurrently, the processing power is multiplied by the number of processors. If the failure of a single processor takes the system down, the system reliability is only the probability of all processors being up; if working processors can diagnose and heal or amputate faulty processors and proceed with the job, the system reliability approaches the probability of *any* processor being up. We now consider how to keep processors running concurrently, and then how to keep the system running in the case of module failure.

The first problem in making the machines run independently is the allocation of runnable tasks to processors, so that the full requisite power can be quickly brought to bear on high priority tasks. Our scheme for doing this rests on four key ideas: (1) We break the job up into a set of tiny tasks. (2) Our processors are all identical, asynchronous, and capable of doing any task. (3) We keep a queue of pending tasks, ordered by priority, from which each processor at its convenience gets its next task. (4) For speed and efficiency, we use a hardware device to help manage the queue.

By breaking the job up into smaller and smaller tasks until each one runs in under 300 μs, we effectively determine the responsiveness of our system. Once started, a task must run to completion, but there will be a reconsideration of priorities at the beginning of each new task. We have chosen 300 microseconds as the maximum task execution time because this compromise between efficiency and responsiveness is well matched to the execution time of key IMP functions.

By making the processors identical, we can use the same program in systems of widely varying size and throughput capability. Any processor can be added to or removed from a running system with only a slight change in throughput. The power of all processors quickly shifts to that part of the algorithm where it is most needed.

By queuing pending tasks, we keep track of what must be done while focusing on the most important tasks. By using a passive queue in which the processors check for a new task when they are ready, we avoid some nasty timing problems. Tasks may be entered into the queue at any time, either by a processor or by the hardware I/O devices. This approach is an extremely important departure which avoids the use of conventional interrupts and the associated costs of saving and restoring machine state. Further, this approach neatly sidesteps the problem of routing interrupts to the proper processor.

We could not afford a software queue both because it was slow to use and because processors would have been waiting for each other to get access to the queue. Instead we use a special hardware device called a Pseudo Interrupt Device (PID), which keeps in hardware a list of what to do next. A number can be written to the PID at any time and and it will be remembered. When read, the PID returns (and deletes) the highest number it has stored. By coding the numbers to represent tasks, and keeping the parameters of the tasks in memory, a processor can access the PID at the end of each task and determine very rapidly what it should do next.

Contention

Clearly, the PID must give any task to exactly one processor. This is guaranteed because the PID is on a bus that can be accessed by only one processor at a time and because the PID completes each transaction in a single access. This is an example of the more general problem that whenever two users want access to a single resource there must be an interlock to let them take turns. This is true at many levels, from contention for a bus to processor contention for shared software resources such as a free list. When all the appropriate interlocks have been provided, the performance of the multiprocessor will depend rather critically on the time wasted waiting at these interlocks for a resource to become free. As discussed above, whenever conflicts become a serious problem one provides another copy of the resource. We studied our system behavior carefully, noting areas of conflict, in order to know how many additional copies of heavily accessed resources to provide. Table II provides examples of delays due to various conflicts. Practically speaking, the curve of delay vs. number of resources has a rather sharp knee, so that it is meaningful to make such statements as "a memory bus supports eight processors" or "a free list supports eight processors." Of course, these statements are application related and depend on the frequency and duration of accesses required.

With interlocks, deadlocks become possible (in both hardware and software). For example, a deadlock occurs

TABLE II—Expected System Slowdown Due to Contention Delays

Slowdown	Cause
5.5%	Contention for a Processor Bus.
3%	Contention for the Shared Memory Busses.
5%	Contention for the Shared Memories.
10%	Contention for a single system-wide software resource, assuming each processor wants the resource for 6 instructions out of every 120 instructions executed.
1.7%	Contention for one of two copies of a system-wide software resource, as above.
0.15%	Contention for the parameters of a single 1.3 megabit phone line, assuming the parameters will be used for 160 microseconds every 800 microseconds.

when each of two processors has claimed one of two resources needed by both. Each waits indefinitely for the other's resource to become available.[14] Unless there is a careful systematic approach to interlocks, deadlocks interlock, and require that a processor never compete for a resource when it already owns a higher numbered resource. It is not always practical or possible to do this, although we expect to be able to do so with the IMP algorithms.

An interesting example of a deadlock occurs in our bus coupling. To permit processors to access one another, for mutual turn on, turn off, testing, etc., the path connecting each processor bus with the I/O bus is made bi-directional. Thus processors access one another via the I/O bus. In a bi-directional coupler, a deadlock arises when units obtain control of their busses at each end and then request access via the coupler to the bus on the other end. Because the backward path is infrequently used, we simply detect such deadlocks, abort the backward request and try again.

Reliability

We have taken a rather ambitious stand on reliability. We plan to detect a failing module automatically, amputate it, and keep the system running without human intervention if at all possible. Critical to our approach is the fact that there are several processors each with private memory and thus each able to retreat to local operation in the face of system problems. To reduce our vulnerability further, power and cooling are provided on a modular basis so that loss of a single unit does not jeopardize system operation. We are only mildly concerned with the damage done at the time of a failure, because the IMP system includes many checks and recovery procedures throughout the network.

The first sign of a failure may be a single bit wrong somewhere in shared memory, with all units apparently functioning properly. Alternatively, the failure may strike catastrophically, with shared memory in shambles and the processors running protectively in their local memories. Against this spectrum we cannot hope for a systematic defense; instead we have chosen a few defensive strategies.

So long as a module is failing, recovery is meaningless. We must run diagnostics to identify the bad module, or see if cutting a module out at random helps things. We feel that identifying such a solid failure will be relatively easy. Since a processor without couplers is completely harmless, once we identify a malfunctioning processor, we amputate it by turning off its bus couplers. We considered the possibility of a runaway processor turning good processors off. This is unlikely to begin with but we decided to make it even less likely by requiring a particular 16-bit password to be used in turning off a coupler. A runaway processor storing throughout shared memory would need this password in its accumulator to accidentally amputate. Similarly we require a password for one processor to get at another's local memory.

Against intermittents we use a strategy of dynamic reinitialization. Every data structure is periodically checked; every waiting state is timed out; the code is periodically checksummed; memory transfers are hardware parity checked; memory is periodically tested; processors are periodically given standard tests. Whenever anything is found wrong, the offending structure is initialized. Using this scheme we may not know what caused a failure, but its effects will not persist. In the most extreme cases we will need to reload all the program in main memory. Fortunately we have a communications network handy to load from. This technique of reloading has worked remarkably well in the current ARPA Network. Each processor has a copy of the reload program in its local memory, thus making loss of reload capability unlikely.

We might seem to be vulnerable to memory or I/O failures, particularly those involving the PID and the clock. If these modules fail it does indeed hurt us more, but only because we have fewer modules of these types in our system. If we provide redundant modules, the system can reconfigure itself to substitute a spare module for a failed one. Our design allows multiple I/O busses with multiple PIDs and clocks, and we could even have separate backup interfaces to vital communication lines on separate busses.

To summarize, the mainstay of our reliability scheme is a system continually aware of the state of things and quickly responding to unpleasant changes. The second line of defense consists of drastic actions like amputation and reloading. Assuming we can make all this work, we will have quite a reliable system, perhaps even one in which maintenance consists of periodic replacement of those parts which the system itself has rejected.

STATUS AND NEAR FUTURE

In February 1973, as this paper is submitted, we are very much in the middle of our multiprocessor development. Much progress has been made and we are increasingly confident of the design, but much work remains to be done.

The broad design is complete; all Lockheed-provided units (CPUs, memories, busses, etc.) have been delivered; prototype wire-wrapped versions of the crucial special modules have been completed, including the Bus Couplers, Pseudo Interrupt Device, clock, and modem interfaces; and a multi-bus, multi-processor-per-bus assembly has been successfully tried with a test program. A substantial program design effort has been in progress and coding of the first operational program has been started. We are still doing detailed design of some hardware, and we are still learning about detailed organizational issues as the software effort proceeds. An example of such an

area is: exactly how is it best for processors to watch each other for signs of failure?

We currently anticipate the parts cost of the prototype fourteen-processor system, without communication interfaces, to be under $100K.

Hopefully, by the time this paper is presented in June 1973, we will be able to report an operational prototype multiprocessor system. Beyond that, our schedule calls for the installation of a machine in the ARPA Network by about the end of 1973. We also plan to construct many variant systems out of this kit of building blocks, and to experiment with systems of varying sizes. As part of this work, we plan to concentrate on the very smallest version that may be sensible, in order to provide a minimum cost IMP for spur applications in the ARPA Network.

As the design has proceeded, our attraction to the general approach has increased (perhaps a common malady), and we now believe that the approach is applicable to many other classes of problems. We expect to explore such other applications as time permits, with initial attention to two areas: (1) certain specialized multi-user systems, and (2) high bandwidth signal processing.

With our presently planned building blocks, although we do not yet know what will limit system size, we do not now see any intrinsic problem in constructing systems with fifty or a hundred processors. As improvements in integrated circuit technology occur, and processors and memories become smaller and cheaper, organization and connection become the paramount questions in multiprocessor design. We expect to see many attempts at multiprocessors, and are hopeful that the ideas embodied in this design will help to steer that technology. Perhaps minicomputer/multiprocessors will soon represent real competition for the various brontosaurus machines that now abound.

ACKNOWLEDGMENTS

Our new machine design is a product of many minds. We gratefully acknowledge the specific design contributions of M. Kraley, A. Michel, M. Thrope, and R. Bressler. Helpful criticism and an important idea about the Pseudo Interrupt Device were contributed by D. Walden. Assistance in planning and in the choice of building blocks was contributed by H. Rising. Helpful ideas and criticism were provided by J. McQuillan, B. Cosell, and A. McKenzie. Assistance with support software was provided by J. Levin.

We also wish to express appreciation for the support and encouragement provided by Dr. L. Roberts of the Advanced Research Projects Agency.

REFERENCES

1. Lehman, M., "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors", *Proc. IEEE*, Vol. 54, No. 12, pp. 1889-1901, December, 1966.
2. Lorin, H., *Parallelism in Hardware & Software - Real and Apparent Concurrency* Prentice-Hall, 1971.
3. Slotnick, J. L., Bork, W. C., McReynolds, R. C., "Solomon", *AFIPS Conference Proceedings*, FJCC 1962.
4. Barnes, G. H., et al, "The Illiac IV Computer", *IEEE Trans.* C-17, Vol. 8, pp. 746-757, August 1968.
5. Anderson, D. W., Sparacio, F. J., Tomasulo, R. M., "The IBM System/360 Model 91 - Machine Philosophy and Instruction Handling", *IBM Journal* No. 11, January 1967, pp. 8-24.
6. Cohen, E., "Symmetric Multi-Mini-Processors, A Better Way to Go?" *Computer Decisions*, January 1973.
7. Wulf, W. A., Bell, C. G., "C.mmp - A Multi-Mini Processor", *AFIPS Proceedings*, FJCC, Vol. 41, 1972.
8. Cosserat, D. C., "A Capability Oriented Multi-Processor System for Real-Time Applications", *Computer Communication Proc. ICCC*, pp. 282-289, October 1972.
9. Roberts, L. G., Wessler, B. D., "Computer Network Development to Achieve Resource Sharing" *AFIPS Proceedings*, SJCC, Vol. 36, 1970.
10. Heart, F. E., et al, "The Interface Message Processor for the ARPA Computer Network", *AFIPS Proceedings*, SJCC, Vol. 36, 1970.
11. Ornstein, S. M., et al., "The Terminal IMP for the ARPA Computer Network", *AFIPS Proceedings*, SJCC, Vol. 40, 1972.
12. Chaney, T., Ornstein, S., Littlefield, W., "Beware the Synchronizer", *Proc. COMPCON Conference*, 1972.
13. *SUE Computer Handbook*, Lockheed Electronics Company, Los Angeles, 1972.
14. Holt, R. C., "Some Deadlock Properties of Computer Systems", *ACM Computing Surveys*, Vol. 4, No. 3, pp. 179-196, September 1972.

SUPPLEMENTARY BIBLIOGRAPHY

Amdahl, G. M., *Engineering Aspects of Large High-Speed Computer Design - Part II Logical Organization*, IBM Tech. Report TR00.1227, December 1964.

Baskin, H. B., et al, "A Modular Computer Sharing System," *CACM*, Vol. 12, No. 10, October 1969, p. 551.

Bell and Newell, *Computer Structures*, McGraw-Hill, 1971.

Bell, G., et al, *C.mmp the CMU Multiminiprocessor Computer*, Dept. of Computer Science, Carnegie Mellon Univ., August 1971.

Burnett, G. J., et al., "A Distributed PROCESSING System for General Purpose Computing", *AFIPS Proceedings*, FJCC, Vol. 31, 1967.

Dijkstra, E. W., "Cooperating Sequential Processes", in *Programming Languages*, (Gennys, F., ed.), Academic Press, pp. 43-110, 1968.

Flynn, M. J., "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*", Vol. C-21, No. 9, September 1972.

Flynn, M. J., "Very High-Speed Computing Systems", *Proc. IEEE*, Vol. 54, No. 12, pp. 1901-1909, December, 1966.

Holland, J. H., "A Universal Computer Capable of Executing an Arbitrary Number of Sub-Programs Simultaneously," *AFIPS Proceedings*, FJCC, pp. 108-113, 1959.

McQuillan, J. M., et al, "Improvements in the Design and Performance of the ARPA Network", *AFIPS Proceedings*, FJCC, Vol. 41, 1972.

Ornstein, S. M., Stucki, M. J., Clark, W. A., "A Functional Description of Macromodules" *AFIPS Proceedings*, SJCC, Vol. 30, 1967.

Pirtle, M., "Intercommunication of Processors & Memory", *AFIPS Proceedings*, FJCC, Vol. 31, 1967.

Randell, B., "Operating Systems - The Problems of Performance and Reliability", *IFIP Congress 71*, Ljubljana, North Holland Pub. Co., 1972, pp. 281-290.

A Description of the Advanced Scientific Computer System, Texas Instruments, Inc., 1972.

Thornton, J. E., "Parallel Operation in the Control Data 6600", *AFIPS Proceedings*, FJCC, Vol. 26, 1964.

Wulf, W., et al, *Hydra— A Kernel Operating System for C.mmp*, Dept. of Computer Science, Carnegie Mellon Univ., 1971.