The BTl 8000—Homogeneous, general-purpose multiprocessing

by GEORGE R. LEWIS and J. SHIRLEY HENRY

BTl Computer Systems
Sunnyvale, California

and

BRIAN P. McCUNE*

Stanford University
Stanford, California

APPROACHES TO GROWTH IN COMPUTER USAGE

With the price of computer hardware decreasing steadily and the scope of data processing applications ever rising, the problem of upgrading a computer system is omnipresent. The myriad of potential pitfalls includes losing an investment in purchased hardware and software, reprogramming applications, reformatting data files, retraining personnel, operating two different systems in parallel during the conversion period and reoptimizing finely tuned applications.

Computer series

Most computer vendors attempt to ease the pain of upgrades by offering an entire series of similar computers spanning a wide range of price and performance. Such series (e.g., IBM 3601,2,8 and its successors, and PDP-119) usually consist of a relatively constant machine architecture with different underlying implementations which yield faster and faster central processing units (cpus) and memories (see Reference 7, Part 6, "Computer Families").

Unfortunately, re-implementation of an architecture may result in forced changes at the user level; for example, a program or even the operating system may run on one implementation and not another. At the least, re-implementation is costly in terms of the additional design and manufacturing efforts required.

In addition, minimal configurations of cpus, memories, and peripherals are designed with the typical user in mind. Thus, it is common for a user requiring much central processing power but little I/O to be forced into a configuration sporting many unneeded data channels in order to get a powerful enough cpu.

* Currently at Systems Control, Inc., Palo Alto, California.

Modular multiprocessor

An alternative to the typical computer series with multiple implementations is a multiprocessor architecture which offers systems of one to many identical cpus, memories and data channels, thereby spanning the desired price and performance range. Cpus, memories and channels can be added independently to meet a particular processing need and without changes to user programs.

It costs less to design a simple cpu once and replicate it within an elegant multiprocessor architecture than to design many different, possibly very complicated cpus. Increased reliability because of redundant resources is an additional benefit of such a modular approach. Indeed, we feel that computer systems of the future should not have to depend upon the availability of a single cpu.

BACKGROUND ON MULTIPROCESSING

There has been much interest in multiprocessing computer systems since the first, the Burroughs D 251,3 appeared in 1962. A multiprocessor system has more than one cpu, each with its own stream of instructions operating on its own data stream. Such an architecture is often termed "MIMD" for these "multiple instruction, multiple data" streams.19

Homogeneity of resources

There are many possible ways to classify MIMD architectures (Figure 1, after Reference 16). Perhaps the most useful is by their degree of homogeneity, or the degree to which resources of the system (processors, main memory, and peripheral units) can be shared indistinguishably (Reference 18, pages 104-105). This has also been defined as the quality that all resources of a particular type appear symmetric to the rest of the system.17
Adding capabilities to an existing system is straightforward when it is composed of homogeneous modules. The more homogeneous a system is, the more robust (i.e., less prone to catastrophic failure), because other equivalent resources are still available when one fails. Using only one basic design in each part of a multiprocessor also keeps design, manufacturing and programming costs to a minimum. And homogeneity allows more efficient utilization of resources, since each member of a particular resource pool is unrestricted as to what tasks it can do.

Most multiprocessors of the past have lacked homogeneity in at least one respect. The concept of homogeneous cpus, for instance, breaks down if a system has one general purpose processor and one or more special purpose processors, e.g., the fast floating point AP-1208 array processor by Floating Point Systems. In this case, programming for each type of cpu is obviously different and must be done explicitly. (Not to be considered a multiprocessor at all in this discussion is the typical computer with one general-purpose cpu and one or more special-purpose I/O processors, Reference 7, Part 5, Section 2, "Computers with One Central Processor and Multiple Input/Output Processors.")

Ideally, all cpus would have access to all of memory to make the sharing of programs and data simpler and more efficient, and thus cpus would not have local private memories (other than perhaps a small transparent cache for performance improvement). This type of cpu is often termed "closely coupled." The Tandem Computers T 16 NonStop computer, which has nonshared memory for each cpu, must be regarded as a network of computers (each consisting of a cpu, I/O processor and memory) with partial sharing of peripherals. Similarly, the Datapoint Attached Resource Computer (ARC) and IBM's loosely coupled attached support processor (ASP) (Reference 7, page 306) are networks (Figure 2). The Carnegie-Mellon University C.mmp (based on Digital Equipment PDP-11 cpus) and the Bolt Beranek and Newman PLURIBUS (based on Lockheed SUEs) are hybrids, having both shared memory and memory private to each cpu.

The sharing of all I/O capabilities is the next aspect of homogeneity, one which is lacking in C.mmp, for example, because an I/O device must be attached to the Unibus of one of its PDP-11s and is restricted to communicating with a local memory there. This is undesirable because the loss of either the cpu or its local memory isolates the I/O device from the rest of the system.

A final aspect of homogeneity concerns software, including both the operating system and user programs. Ideally, for efficient use of processor and memory resources there should be only one copy of one operating system which can be run on any of the cpus; this is the case with C.mmp. In practice, however, the typical multiprocessor system has resorted to having either separate copies of the same operating system running on each cpu or one copy restricted to always running on the same cpu (the common master-slave mode of many dual processor systems).

For reliability and efficiency it should be possible for any user program to run on any processor. Too often, applications are segregated to run on dedicated cpus. When segregation is done by entire classes of programs (e.g., interactive, batch, data base management), it may be a consequence of the lack of homogeneity in I/O. A common example of this is the restriction of interactive terminals to one cpu.

---

This text contains a table and a diagram.

### Table: Multiprocessing Aspects of Some Computers

<table>
<thead>
<tr>
<th>Computer</th>
<th>Full Homogeneity?</th>
<th>Main Memory</th>
<th>Peripherals</th>
<th>Operating System &amp; User Programs</th>
<th>Purpose</th>
<th>Cpu-Memory Interconnection</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTI 8000</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>General Purpose Timesharing</td>
<td>None</td>
</tr>
<tr>
<td>D 825</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Command &amp; Control</td>
<td>None</td>
</tr>
<tr>
<td>CLC</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Command &amp; Control</td>
<td>Multiported Memory</td>
</tr>
<tr>
<td>U 1108</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>General Purpose</td>
<td>Multiported Memory</td>
</tr>
<tr>
<td>PLURIBUS</td>
<td>No</td>
<td>Partial</td>
<td>Yes</td>
<td>Yes</td>
<td>Packet Switching</td>
<td>Multiple Buses</td>
</tr>
<tr>
<td>C.mmp</td>
<td>Yes</td>
<td>Partial</td>
<td>No</td>
<td>Yes</td>
<td>Realtime Artificial Intelligence</td>
<td>None</td>
</tr>
<tr>
<td>PRIME</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Timesharing</td>
<td>None</td>
</tr>
<tr>
<td>T 16</td>
<td>Yes</td>
<td>No</td>
<td>Partial</td>
<td>No</td>
<td>Transaction Processing</td>
<td>None</td>
</tr>
<tr>
<td>ARC</td>
<td>No</td>
<td>No</td>
<td>Partial</td>
<td>No</td>
<td>General Purpose</td>
<td>None</td>
</tr>
<tr>
<td>IBM 370 ASP</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Scientific</td>
<td>None</td>
</tr>
<tr>
<td>AP-1208</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
<td>None</td>
</tr>
</tbody>
</table>

### Figure 1: Multiprocessing Aspects of Some Computers

Adding capabilities to an existing system is straightforward when it is composed of homogeneous modules. The more homogeneous a system is, the more robust (i.e., less prone to catastrophic failure), because other equivalent resources are still available when one fails. Using only one basic design in each part of a multiprocessor also keeps design, manufacturing and programming costs to a minimum. And homogeneity allows more efficient utilization of resources, since each member of a particular resource pool is unrestricted as to what tasks it can do.

Most multiprocessors of the past have lacked homogeneity in at least one respect. The concept of homogeneous cpus, for instance, breaks down if a system has one general purpose processor and one or more special purpose processors, e.g., the fast floating point AP-1208 array processor by Floating Point Systems. In this case, programming for each type of cpu is obviously different and must be done explicitly. (Not to be considered a multiprocessor at all in this discussion is the typical computer with one general-purpose cpu and one or more special-purpose I/O processors, Reference 7, Part 5, Section 2, "Computers with One Central Processor and Multiple Input/Output Processors.")

Ideally, all cpus would have access to all of memory to make the sharing of programs and data simpler and more efficient, and thus cpus would not have local private memories (other than perhaps a small transparent cache for performance improvement). This type of cpu is often termed "closely coupled." The Tandem Computers T 16 NonStop computer, which has nonshared memory for each cpu, must be regarded as a network of computers (each consisting of a cpu, I/O processor and memory) with partial sharing of peripherals. Similarly, the Datapoint Attached Resource Computer (ARC) and IBM's loosely coupled attached support processor (ASP) (Reference 7, page 306) are networks (Figure 2). The Carnegie-Mellon University C.mmp (based on Digital Equipment PDP-11 cpus) and the Bolt Beranek and Newman PLURIBUS (based on Lockheed SUEs) are hybrids, having both shared memory and memory private to each cpu.

The sharing of all I/O capabilities is the next aspect of homogeneity, one which is lacking in C.mmp, for example, because an I/O device must be attached to the Unibus of one of its PDP-11s and is restricted to communicating with a local memory there. This is undesirable because the loss of either the cpu or its local memory isolates the I/O device from the rest of the system.

A final aspect of homogeneity concerns software, including both the operating system and user programs. Ideally, for efficient use of processor and memory resources there should be only one copy of one operating system which can be run on any of the cpus; this is the case with C.mmp. In practice, however, the typical multiprocessor system has resorted to having either separate copies of the same operating system running on each cpu or one copy restricted to always running on the same cpu (the common master-slave mode of many dual processor systems).

For reliability and efficiency it should be possible for any user program to run on any processor. Too often, applications are segregated to run on dedicated cpus. When segregation is done by entire classes of programs (e.g., interactive, batch, data base management), it may be a consequence of the lack of homogeneity in I/O. A common example of this is the restriction of interactive terminals to one cpu.

---

Figure 2: Architecture of a computer network.
Purpose

Another dimension of multiprocessors is their purpose. Most successful multiprocessors have not been designed for the general purpose commercial computing environment. The D 825 and the Bell Telephone Laboratories CLC (part of the SAFEGUARD ABM system)\textsuperscript{25} were designed for real-time military command and control applications. The T 16 was designed for transaction processing applications in industries, such as banking and airlines, in which continuous system availability is required. PLURIBUS was designed as a packet-switching node for the ARPA net. One-of-a-kind research prototypes have included C.mmp and the University of California at Berkeley PRIME,\textsuperscript{2} designed for real-time artificial intelligence applications and timesharing, respectively.

Many computer manufacturers have introduced multiprocessing to extend the high-end capabilities of a computer line. Examples are the Sperry Rand Univac 1108,\textsuperscript{27} IBM 370 MP,\textsuperscript{25,15} and DECSystem-10\textsuperscript{2} dual processors. These architectures were not designed from the start with a multiprocessor in mind. So we often observe systems which could theoretically support a number of processors, but in practice never have more than two.

Closely related to the purpose of a multiprocessor is the way in which user programs are expected to take advantage of its multiprocessing capability. There are two basic philosophies.

The first is most useful for special-purpose multiprocessors (e.g., C.mmp, PLURIBUS, CLC) dedicated to running a small number of large, known programs. In this case the user must explicitly segment an application into a number of independent programs. These programs, called processes, are designed so that they may run concurrently, thus taking advantage of the multiple cpus available. Segmenting a program currently must be done by hand by the programmer or analyst and is not a simple task.

The alternative used in general purpose multiprocessors, especially in time-sharing, is to rely upon the job mix of a multiprogramming environment to provide enough processes (one per job) to keep all cpus busy. This avoids the programmer headaches of program segmentation and also avoids the system overhead resulting from the interprocess communication needed to coordinate processes. Of course, it is desirable to allow such coordination, while not requiring it.

Processor-memory interconnection schemes

Probably the most critical design decision in a multiprocessor architecture is selection of the method by which main memory is interconnected to the cpus and peripheral processors (channels). The bandwidth of this interconnection switch provides an upper bound on the number of processors which can be handled, and thus on the throughput of the entire system.

Four basic schemes have been used. In decreasing order of complexity and potential throughput, they are

1. A crossbar switch or matrix connecting each memory unit with each processor.
2. Multiported memory, in which each memory provides a port for a connection from each processor.
3. Multiple buses, each of which connects some cpus, memories and other buses (which may in turn connect to other cpus, memories and buses).
4. One common time-shared bus, over which all data traffic must pass.

Although the first three alternatives provide concurrent transfers between more than one processor-memory pair (Figure 3), analogous to a telephone switching exchange in which more than one conversation can occur simultaneously, they are also relatively expensive and typically only used in large systems. The D 825 and C.mmp used crossbar schemes, while the dual processor IBM 370 MP systems, Univac 1108, and DECSystem-10 (along with most other multiprocessors) rely on multiported memory.

The complexity (and hence cost) of a switch is proportional to the number of hardware switching elements it has, which in turn depends on the number of unique interconnections allowed between processors and memories. If each of \( p \) processors is to be capable of "talking" to each of \( m \) memories in parallel, then there must be \( p \times m \) unique interconnections. If, on the other hand, each processor and memory need only be able to talk to a single common bus, then the number of interconnections is reduced to just \( p + m \).

For systems with more than two cpus and two independent memories, the difference in the complexity of the two schemes becomes immense.

Until now, no one has developed a bus fast enough to make a multiprocessor using a single timeshared bus feasible (Figure 4), analogous to a telephone exchange in which all subscribers are on a single party line. For example, limited by buses with 200-nanosecond data transfer rates, a PLURIBUS system requires multiple buses, each bus connecting up to two processors and two memories. Up to seven of these buses may then be connected to shared memory and peripherals by still other buses.

![Diagram](image-url)
ARCHITECTURE OF THE BTI 8000

The BTI 8000 computer system was designed to be a general-purpose system featuring inexpensive, modular resources for easy upgrading and high reliability, interactive time-sharing, an easy-to-use and constant virtual machine for the user and high-level programming languages.11

In light of these goals, the BTI 8000 was designed from the beginning with fully homogeneous, general-purpose multiprocessor as an objective. To our knowledge it is the first system to meet this goal, while also being the first general purpose multiprocessor minicomputer. The BTI 8000 is homogeneous with respect to CPUs, memory, peripherals, operating system and user programs.

A BTI 8000 system (Figure 5) is based on 32-bit words and consists of a number of modular resource units.

1. Computational processing units (CPUs). (This paper denotes a computational processing unit of a BTI 8000 by “CPU” and an arbitrary central processor by “cpu.”)
2. Memory control units (MCUs) and associated memory.
3. Peripheral processing units (PPUs) and associated I/O peripherals.
4. System services unit (SSU).

Up to 16 of these resource units (each of which is actually a microprogrammed processor) communicate via a single

---

**Figure 4**—Architecture of a multiprocessor with a single common time-shared bus.

**Figure 5**—Block diagram of the BTI 8000 architecture.
32-bit-wide directly connected common (or "global") time-shared (or functionally and physically "non-dedicated") bus. Its speed is the key to the success of the 8000 as a multiprocessor. The bus can transfer one 32-bit word every 66.7 nanoseconds, or 15 million words (60 million bytes) per second. This speed is possible because the passive, synchronous bus relies upon distributed (or "decentralized") control logic for fast resolution of the bus contention arising from simultaneous, independent bus requests from two or more resource units.

The hardware modularity (along with an operating system which requires no reprogramming to handle hardware reconfiguration) makes possible both easy system enhancement and graceful degradation. A small user may start out with the minimum system configuration, comprising one each CPU, MCU, PPU, and SSU and corresponding to a medium speed minicomputer. As system requirements increase, additional resource units can be added, resulting in a system which has a throughput rivaling that of many mainframes, but which is much more cost effective. For example, an additional CPU costs a fraction of the price of a complete system and consists of one 20-inch-by-23-inch printed circuit board ready to be plugged into the bus (Photograph 1). A typical large 8000 configuration might consist of six CPUs, six MCUs, three PPUs, and one SSU.

Because all CPUs must communicate via the same bus to the same homogeneous memory, the question arises, how badly will bus and memory contention degrade performance in a system with more than one CPU? In many multiported memory multiprocessors, for example, more than two cpus are impractical because the cpus are faster than memory and not enough memory ports are provided. In contrast, the number of CPUs and MCUs in a BT1 8000 can be flexibly chosen based upon the relative speeds of each type of resource. And the bus is fast enough so that contention for it is minor.

Simulation studies of various configurations of CPUs, MCUs and PPUs have been run assuming typical memory accesses for each type of processing unit. An 8000 system with six CPUs, six MCUs, and two PPUs should have the throughput of approximately five separate 8000 systems, each having one CPU, one MCU and one PPU (Figures 6, 7). Each of these five systems would, of course, require its own SSU, bus, and peripherals. So for equivalent throughput, the multiprocessor configuration requires one additional CPU and MCU, but eliminates the redundant three PPUs, four SSUs, four buses, four sets of peripherals, four sets of power supplies, four system cabinets, etc. The multiprocessor will become even more attractive as the cost of CPUs and MCUs continues to drop with respect to the cost of other system components.

**Bus protocols**

The common bus is a number of parallel lines which may be grouped into

1. Four lines identifying the bus slot to which the current message on the bus is being routed.

<table>
<thead>
<tr>
<th>Typical Multiported Memory Multiprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cpus</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

* = not practical

**Table 6**—Throughput of multiprocessors.

<table>
<thead>
<tr>
<th>BT1 8000</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUs,MCUs,PPUs</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>1,1</td>
</tr>
<tr>
<td>2,2</td>
</tr>
<tr>
<td>3,3</td>
</tr>
<tr>
<td>4,4</td>
</tr>
<tr>
<td>5,5</td>
</tr>
<tr>
<td>6,6</td>
</tr>
<tr>
<td>7,7</td>
</tr>
</tbody>
</table>

All traffic on the bus takes place in terms of messages between a source and a destination resource unit. Furthermore, all resource units use the same types of messages.

There are three message types—command, data and abnormal data. A command message is a request from one resource unit to another to carry out a specified function. A data message has 32 bits of data on the data lines (e.g., a word read out of memory by an MCU and being sent to a CPU). An abnormal data message is sent instead of a data message if an error occurs in trying to provide data. In this case the data lines contain error information.

A command message has a type code on the four high-order data lines, has an address or request code on the lower 22 lines, and leaves the other six lines unused. There are five command types—information request (known as "who are you"), read, write, read/modify/write, and self test (which starts microcode diagnostics).

Depending upon the request code, an information request can ask a resource unit to return a data message identifying its resource unit type, interrupt status, bus error status, error count, etc. This could then be used, for instance, by the SSU to determine the system configuration upon reload (e.g., how many CPUs are now on-line and occupying which bus slots).

A read command specifies the 22-bit address of the requested word to the destination resource unit. The destination responds with either a data message containing the
Photograph 1—BTI 8000 CPU board and chassis for up to 16 resource units.
The BTI 8000—Homogeneous, General-Purpose Multiprocessing

Figure 7—Throughput graph for multiprocessors.

A write command specifies the 22-bit address to be stored into the destination unit. This is followed by a data message containing the word to be stored.

A read/modify/write command is only sent to an MCU. The MCU returns the contents of the addressed word in memory in a data message and then waits for the source resource unit to send another word to be written back into the same location. In the interim the MCU denies other access to that memory location. This is the hardware feature which makes process interlock mechanisms possible on the 8000.

Computational processing unit

Because the system relies upon multiple CPUs working concurrently to achieve high throughput, the CPU design was purposefully kept straightforward and inexpensive. Each CPU is an identical microprogrammed processor with a program state described by eight 32-bit general-purpose registers, a 17-bit program counter, a 17-bit current console area register, a 15-bit process status register, a 32-bit monitor status register and a page map of 256 20-bit words.

The current console area register points to the 10-word block of memory which stores the user's program state during an interrupt. This register, instructions which load and store the entire console area, and interrupt firmware which stores the area automatically all enhance the speed of context switching.

The process status register contains flags for arithmetic faults and condition bits set by compare operations.

The monitor status register contains information accessible only to the monitor, such as the processor number (i.e., the bus slot the processor is plugged into), interrupt enabling flags and paging control flags.

Every virtual memory address is transformed into a physical address by a calculation involving the page map. The page map is divided into two 128-word halves, one for the monitor and the other for the current user. Which half is used is controlled by bits in the monitor status register.
Memory is divided into pages of 1024 words. Of a 17-bit virtual address, the upper seven bits select a word in the page map and the lower 10 bits a word within the physical page. The physical page is determined by the selected word of the page map. Of this 20-bit word, 12 bits determine the physical page and four bits the bus slot. This scheme allows for many MCUs, each containing \(2^{22} = 12 + 10\) words of memory, and means that PPU's and MCUs are addressed identically. The remaining four bits in the page map word determine the legal types of access to this page (e.g., execute only, read only, read/write) and note whether the page has been read or written.

**Memory control unit**

An MCU is a "slave" resource unit in that it operates only in response to commands from other units, i.e., it does not originate commands itself. Its microprocessor selects one of two memory devices, handles error conditions occurring in memory and locks out other references to the MCU during a read/modify/write cycle (discussed under "Bus Protocols"). The data paths within an MCU are 32 bits wide.

**Peripheral processing unit**

A PPU handles up to four I/O controllers, two for high-speed devices (disk or high-speed tape drives) and two for slow devices (e.g., low-speed tape drives, line printers, and interactive terminals). Data transfers between a PPU and its controllers occur in eight-bit bytes and are buffered by FIFO queues.

A CPU initiates I/O by writing control information into the appropriate PPU. Some of this information is passed on to the controller being addressed. Other information (e.g., the MCU memory starting address for a block transfer) is stored in the local RAM of the PPU's microprocessor. Once a DMA (direct memory access) transfer is begun between the MCU and PPU, the originating CPU is free to do other tasks until completion of the transfer.

The disk and communications controllers are also microprocessor-based.

**System services unit**

The SSU performs a number of functions which need only be done in one place in the 8000. Among these are (1) providing the system-wide clock for synchronizing bus operations, (2) providing a real-time clock readable by other resource units, (3) sensing abnormal environmental conditions (e.g., temperature, humidity), (4) providing a local system status and reload capability via an extremely simple front panel and (5) providing remote diagnostic and preventive maintenance capabilities for BTI personnel via a dial-up communications port.

**INSTRUCTION SET ARCHITECTURE**

The instruction set processor of the CPU was designed to (1) keep bus traffic due to instruction fetches to a minimum and (2) to enhance the efficiency of the operating system, system utilities, compilers and compiler-generated code. These two goals have led to a rather high-powered instruction repertoire—there are 10 data types, approximately 200 different operation codes, and about 50 addressing modes.

**Data types**

Data types supported include 32- and 64-bit fixed point numbers, 64-bit floating point numbers, eight-bit ASCII characters, one-bit Boolean values, 32-bit pointers, one- to 32-bit bytes, arrays of all of the preceding types, linked lists and pushdown stacks. Arithmetic is two's complement. There is a special "undefined number" value (a leftmost one bit followed by 31 or 63 zero bits) which may cause an error trap when used. A byte may cross the boundary between two words.

**Instruction set**

All instructions comprise one 32-bit word in which the leftmost 10 bits define the operation code, sometimes specifying a general register to be used, and the rightmost 22 bits specify the operand. The instruction set is fairly complete, providing most useful variants of each instruction, e.g., reverse subtract and reverse divide.

As a precautionary measure to prevent runaway programs, operation codes which arise in common data words are illegal. These include words containing all zeros or all ones, containing the undefined number, or starting with an ASCII space.

Arithmetic, Boolean, and compare instructions all have both register-to-register and register-to-memory modes. Also, certain arithmetic operations store the result into both a register and memory. Arithmetic reverse subtraction and division, as well as Boolean subtraction and reverse subtraction, are provided.

To support process synchronization, a number of instructions can read and write a word of memory in one indivisible step. In this class of instructions are the four Boolean operations to memory, the set and test operation, an operation which exchanges a register with a word in memory, and the seven single-word fixed point add and subtract to memory instructions.

**Instructions to enhance systems software**

A number of instructions simplify common tasks done by systems software. Two instructions jump based on the value of a particular bit of a register. Other instructions store in memory useful constants such as 0, 1, -1, and the "unde-
defined" value. One instruction jumps to itself without requiring any subsequent instruction fetches from memory, effectively causing the CPU to do nothing until interrupted.

The permutation operation computes the exclusive OR of those words in a contiguous 32-word block of memory which are indicated by a one in the corresponding bit position of a register. Thus, the bits in a register may be permuted according to a pattern defined in the 32-word block of memory. A checksum of a block of memory is computed by initializing the controlling register to all ones. By loading the register with a data word and memory with 32 words containing the integer 1, the parity of the register is computed.

Operations on linked lists

Linked lists are supported by instructions which advance to the next element of a list, testing for the end of the list. Both one-way and two-way lists may be used.

A one-way list consists of a number of list elements, or blocks of memory of arbitrary size. The address field (rightmost 17 bits) of the first word of each element contains the address of the successor element in the list, or zero if there is none. A pointer to the current place in the list is kept in a register. If a list is to be left unmodified, this is all the information necessary to traverse it. If, however, elements are to be inserted or deleted, an optional pointer to the predecessor of the current element is maintained in an adjacent register.

In place of the address of its successor, an element of a two-way list has the exclusive OR of the addresses of its left and right neighbors. Call this lORr. An important property of exclusive OR is that (lORr) OR l = r and (lORr) OR r = l. Thus, given the addresses of an element and one neighbor, the address of the other neighbor can be calculated. So the instruction to move ahead in a two-way linked list requires that pointers to both the current and previous elements be in adjacent registers. An example Exclusive OR Link and Jump on Non-Zero Address instruction is

HERE: XLINA i THERE

The effect of executing this instruction is defined by the following program, and is shown graphically in Figure 8.

```
temp := registeri+1;
registeri+1 := registerj;
registerj := contents (registeri+1);
if registeri ≠ 0 then pc := THERE;
```

Also, an instruction is available which searches a one-way linked list for an element containing a specified key. The key may be a word, character, or byte which is offset a fixed amount from the first word of the list element, as specified in the instruction. One register is loaded with a pointer to the first element to be examined. After a successful search, it points to the desired element. Another register holds the key being searched for.

```
if register + registeri = key then pc := THERE;
```

Subroutine linkage instructions

A group of 27 instructions is provided for subroutine linkage, including parameter passing and type checking and ensuring that a legitimate subroutine is called. The following example shows a typical pair of calling and entering sequences:

```
| C: CALL S | S: ENTR REG7 |
| PAR A1  | STP F1 |
| PAR A2  | STPV F2 |
| PAR A3  | STP2 F3 |
| PARV A4 | STPV F4 |
| PARL A5 | STPL F5 |
```

Here the main program on the left calls subroutine S, passing it five actual parameters, A1 through A5. After type checking and storing the parameters into local locations F1 through F5, the body of the subroutine is executed, and then control is returned to the main program at C+6.

These instructions use general registers R0 for passing single-word parameters, R0 and R1 for double-word parameters and R7 for holding the next address and parameter specifications.

The CALL Subroutine instruction ensures that the instruction located at S is some type of Enter Subroutine (ENTR being the simplest of these), saves the contents of R7 in memory location REG7 and puts S+1 in R7.

The PARameter instruction at C+1 stores its type (in this case, single-word call by reference) along with C+2 in R7, loads R0 with the address A1, and jumps to the address previously in R7 (S+1).

The STP Store Parameter instruction at S+1 ensures that its desired type is the same as that of the actual parameter, as defined in R7 (it is). Then the contents of R0 (A1) are stored in F1, S+2 put in R7, and control returned to C+2.

The rest of the corresponding Parameter and Store Parameter instructions are executed in interleaved order until a Store Parameter Last (STPL) is encountered. This instruction marks the last parameter, so control passes into the body of the subroutine instead of returning to C+6 for more parameters.

At the end of subroutine execution, the LEAVE Subroutine instruction returns to C+6 (assumed to still be in R7) after restoring R7 from REG7.

Suffixes "2" and "V," singly or together, on PAR and STP operation codes define double word and call by value types. For example, PARV2 A6 will load the 64-bit value located at A6 and A6+1 into R0 and R1. There are three other type bits available to distinguish, e.g., fixed and floating point operands.

Some type coercion is possible. The PAR at C+2 is call by reference, while the corresponding STPV at S+2 specifies call by value. The STPV is smart enough to load indirect through R0 to get the value stored at A2.
Before:

![Diagram](image)

After:

![Diagram](image)

Figure 8—Registers and memory snapshots of “Exclusive OR Link and Jump on Non-Zero Address” instruction.

**Instructions for compiled code**

To improve compiler efficiency, an instruction exists to load an effective address. It performs the address calculation done for a load, but loads a register with this computed address instead of the value contained there.

Four instructions trap if an array index is out of prescribed bounds. The instructions handle single word integer, double word integer and floating point indices.

**Addressing modes**

Much of the power of the BTI 8000 instruction set arises from its wealth of ways to address data. As discussed in the
previous section, each 32-bit instruction has its operation code in the upper 10 bits. This leaves 22 bits to define the way in which the operand, operands and/or the location to store the result will be determined. Two instruction classes don't use the addressing modes. Jump instructions use the lower 22 bits as a five-bit number defining which bit to test and a 17-bit jump address. Character instructions have a five-bit extended operation code, and the Character Fill instruction also has an eight-bit immediate operand.

Completely separating designation of the operation codes from the address modes implies that any operation may utilize any address mode which is well defined for that operation. For example, in a single instruction one could add to a register an integer which is stored as a three-bit byte, split across the boundary of two words of memory. For a 32-bit add, the three-bit byte would be right justified and padded with 29 zero bits; for a 64-bit add, an additional word of padding would be provided. Conversely, storing a register into such a three-bit byte location in memory would only use the rightmost three bits of the register.

Addressing mode formats

There are six different formats for addressing modes (Figure 9). The format, as well as the specific address calculation

![Addressing Modes Diagram]

Figure 9—Instruction addressing modes and formats.
to be done, is determined by the mode field of the instruction.

Address mode format A is the simplest one. It is used for direct and indirect access of the memory word specified in the address field. In addition, the address field may be used as an immediate operand in three ways—right-justified with zero fill, right-justified with one fill, and left-justified with zero fill.

Format B specifies a register to be used to index the result of a direct or indirect address calculation. For instructions dealing with 64-bit operands and results, the index register is added twice.

Format C specifies a base register and a constant index or offset. Depending on the submode, six calculations are possible. The simplest of these uses the specified register directly, adding the offset to the contents of the register to form an operand. When addressing a result, the offset is subtracted from the result before the result is stored in the register. The second submode uses the register as an indirect pointer. The next three submodes use the register to point to the base of an array and the offset to specify the desired array element. The array can be of words, characters, or formal parameters, which are indirect pointers to the actual parameters of a procedure. The sixth submode uses the register as a pointer to the top of a stack (pushdown list), growing downward from high memory addresses to low. The register is decremented by the offset prior to storing a result, thus pushing the stack. After fetching an operand from the top of the stack, the register is incremented by the offset to pop the stack. For an instruction which loads an operand and stores a result, the stack is both popped and pushed.

Format D has four calculations analogous to format C, but with the added capability of indexing with a register the final address calculated. An extra mode is used because the submode field is limited to one bit to accommodate the index register field.

Format E specifies type conversion to be done upon loading or storing one or two consecutive registers. For loading an operand from the register(s), three submodes specify converting either a 32- or 64-bit integer into either a 64-bit integer or floating point quantity. The register(s) are unchanged. When storing a result into the register(s), the inverse conversion is done. The type field has bits specifying whether integers are considered signed or unsigned and whether to round or truncate upon conversion.

Format F is used for two byte-accessing modes which allow the 8000 to omit shift instructions. In the first the register is considered to be a circular list of bits, and a byte is referenced starting at the specified bit and extending for the specified length. The offset is unused in this mode. The second mode uses the register as the bit address of a bit array. The desired byte in this array starts with the bit indexed by 32 times the offset plus the bit field. The size of the byte is specified by the byte length field. The byte may cross a word boundary, hence the name "zigzag byte" for this addressing mode.

Format of pointer words

Words used as pointers to fields in memory use the format shown in Figure 10. A base or index register for a word array uses only the address field. A base or index register for a character array in addition uses the character field to select a particular character of the word addressed. The base register for a bit array also uses the bit field to specify a bit within the character.

The mode field is used in pointer words in a register or memory which are specified by instructions in an indirect addressing mode. Four of the pointer's own modes specify direct addressing and three forms of immediate operands, as discussed earlier under address mode format A. In these modes the character, bit and byte length fields are unused. The fifth pointer mode uses the character field to select a character in the word addressed. The sixth pointer mode references a word of memory and a zigzag byte within that word. The byte starts at the bit specified by the bit field within the character specified by the character field. Indexing a pointer in zigzag byte mode causes the index register to be multiplied by the length field before being added in, thus allowing indexing of an array with elements of arbitrary byte size (not greater than 32 bits).

IMPLEMENTATION

Little has been said about the implementation of the 8000. This has been deliberate. Because memory and logic circuits are continuing their well established trend toward smaller size, lower cost and higher performance, the design philosophy has been to separate specific implementation decisions from the system architecture as much as possible. Thus, a future implementation of a system component should be able to take advantage of advances in integrated circuit technology.

![Figure 10—Format of pointer words.](chart)
Especially important in this regard was the decision to make every resource unit be a microprogrammable computer with its own internal microprocessor, memories, data paths and I/O connections (current architecture shown in Figures 11-14). From the viewpoint of the resource unit, these I/O connections include the common system bus plus main memory (in the case of an MCU), standard I/O controllers (in the case of a PPU) and special devices such as a thermometer (in the case of the SSU). Each of these computers may now be treated as possessing a flexible architecture, easily changed when desirable, as long as the I/O characteristics of each computer do not change.

SYSTEMS SOFTWARE

A user of the BTI 8000 is presented with a virtual machine which is independent of a particular physical configuration. For example, if an additional CPU is required to meet increasing system load, it is merely plugged into the bus and the operating system is reloaded at the push of a single button. The system automatically recognizes the hardware reconfiguration; no user reprogramming is required. In fact, a user never knows—and need not care—on which CPU his program was run. Similarly, a failing CPU in a multiprocessor configuration is merely unplugged from the bus, followed by the same reload. The only difference noticed by a user may be a degradation of response time. Because of virtual memory mapping, physical memory can be similarly reconfigured without affecting the user's access to 131,072 (128K) words (512K bytes) per user process.

The 8000 is designed to facilitate the use of high-level languages, including PASCAL, FORTRAN, BASIC, COBOL and RPG. The use of vendor-supported PASCAL is promoted because of its applicability to structured programming and the specification of concurrent processes. Unlike uniprocessor systems, a software process on the BTI
Figure 13—Peripheral processing unit.
8000 may actually be run concurrently, communicating via shared synchronization regions which are kept in main memory. To promote the use of advanced high-level languages, the PASCAL compiler is the only one bundled with the hardware. By contrast, the assembler is available only to educational institutions for instructional purposes.

POTENTIAL PROBLEMS

The high reliability and availability of the BTI 8000 depend strongly on the unique elements of the system. For example, each system has only one active SSU, which includes the system clock. However, a spare SSU can be kept ready to go online if the primary SSU fails.

There is only one bus, the failure of which would be catastrophic. By distributing all of its active logic to each of the connected resource units, most of the potential for such a failure is alleviated. However, distributing the bus logic to resource units means that each such device is more complicated.

Software complexity is usually higher on multiprocessors than uniprocessors. Most of the added complexity has been handled in the 8000 instruction set and operating system. The issue will be further ameliorated by the availability within high-level languages of a capability for writing concurrent processes.
A final problem is the continual desire for larger and larger contiguous addressing spaces for user programs. Because each machine instruction takes one 32-bit word and many bits were allocated to the operation codes and addressing modes of the 8000, virtual addresses are 17 bits. Thus, 128K words (512K bytes) are directly addressable by each user process. While this is actually a rather large address space for a minicomputer, it may be extended even further in the future. In addition, user tasks can be divided into a large number of communicating concurrent processes, each of 128K.

ACKNOWLEDGMENTS

We acknowledge the inspiration and hard work that the BTI Research and Development group has contributed to the 8000 project. We are especially grateful to the management of BTI Computer Systems for its continuing faith in a project of such high risk. Robert Adams, Ron Crandall, Bill Quackenbush, and Don Quaintance provided important details of the 8000 and advice on their presentation during the writing of this paper. Finally, special thanks are due Carol Hjerpe for countless hours spent preparing this document.

REFERENCES