Burroughs

BSP

PARALLELISM — THE DESIGN STRATEGY FOR THE BSP

BSP

BURROUGHS SCIENTIFIC PROCESSOR

PARALLELISM — THE DESIGN STRATEGY FOR THE BSP

CONTENTS

	Page
AN OVERVIEW	A-1
BSP Objective	A-1
BSP System	A-2
BSP Key Features	A-3
BSP Organization	A-3
BSP Characteristics	A-4
Parallel Processor	A-4
Conflict-free Memory Access	A-4
Vector Performance	A-5
Performance Optimization	A-5
File Memory	A-6
Vectorizing FORTRAN Compiler	A-6
BSP Design	A-7
BSP Superiority	A-7
IN PERSPECTIVE	A-9
The BSP – A New Approach	A-9
Linear Vectors	A-10
A Different Kind of Supercomputer	A-11
System Manager	A-11
Overlapped Instruction Mode	A-11
Linear Vector Approach to Parallelism	A-12
Scalar Operations	A-13
BSP Approach to Scalars	A-14
The BSP Design	A-15
I/O Subsystem	A-15
Computational Envelope	A-16
File Memory	A-16
Summary	A-17
PARALLEL ARCHITECTURE	A-19
Parallelism	A-19
Templates	A-31
Arithmetic Elements	A-22
Conflict-free Memory Access	A-25
Parallel Processor Control Unit	A-27
Scalar Processing Unit	A-29
BSP Software	A-30

BURROUGHS SCIENTIFIC PROCESSOR

PARALLEL ARCHITECTURE

PARALLELISM

The capability of the Burroughs Scientific Processor (BSP) to sustain high processing rates is achieved via unique parallel designs. The BSP comprises multiple processors arranged to operate in parallel. The combined potential of multiple processors is brought to bear on large computational applications.

Figure 3 illustrates the overall architecture of the Burroughs Scientific Processor (BSP). Four types of parallelism are featured within this architecture; that is, four different classes of computation occur simultaneously. They are:

1. The arithmetic performed by the 16 arithmetic elements (AE's),

2. Memory fetches and stores, and the transmission of data between memory and the AE's,

3. Indexing, vector length, and loop control computations in the parallel processor control unit,

4. The generation of linear vector operation descriptions, which takes place in the scalar processor unit (SPU).

The BSP is analogous to an efficiently operated business. The SPU and its control memory are the executive suite. The executive's instructions are passed to the administrative clerks in the parallel processor control unit. This unit then does the bookkeeping and keeps all the major components of the business as busy and as efficient as possible.

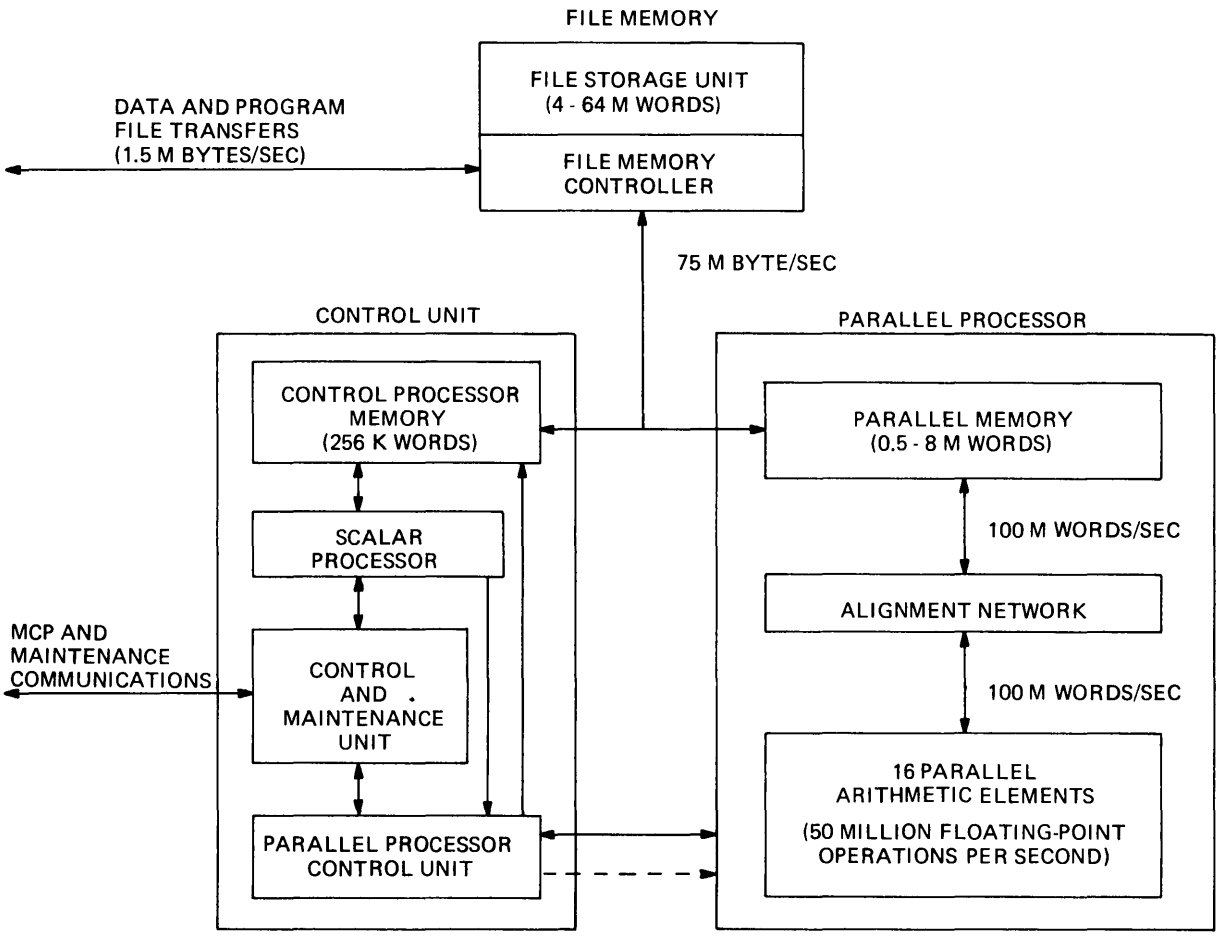


Figure 3. BSP Block Diagram

A fallout from the use of CCD's is excellent reliability. While disc errors are likely to be multiple-bit errors, CCD errors, with proper partitioning, are typically single bits, and, therefore, easily corrected and bypassed using Hamming codes. The BSP file memory features single-error correction/double-error detection (SEC/DED) with all storage and data paths.

The maximum size file memory available on the BSP is 67,108,864 words (nominally 64 million words, where a "million" is 2^{20}). The smallest file memory size is 4 million words. In certain circumstances, some files may overflow file memory. For this reason, an additional file attribute is provided, allowing the user to specify that a file is to be "chaptered", with only one chapter available on file memory at any given time. The operating system automatically transfers chapters between the file memory and the discs on the system manager when the user "releases" a given chapter. The operating system assumes that such files are sequential and it double-buffers the chapters, unless the user asks for a chapter out of sequence.

SUMMARY

Figure 1 shows the BSP connected to a B 7800 or a B 7700 system manager and illustrates that the BSP is the realization of the computational envelope (Figure 2). The high-speed I/O transfers occur inside the BSP between main memory and file memory. New jobs are staged to the file memory, and output from finished jobs is staged to the system manager from the file memory.

Figure 1 also shows some specialized communication paths between the BSP and the system manager. These are used for operating system communications, for performance logging, for hardware error logging, and for maintenance and diagnostic purposes.

The connection to the B 7700 or B7800 is through a standard I/O port. Hence, if a B 7700 owner wished to attach a BSP, he would install the BSP, connect the cables to a B 7700 I/O processor, recompile the operating system with the BSP option set, and go.

It is evident from the way in which the BSP is connected to the system manager, and the arguments upon which the computational envelope approach is based, that normal job flow through the BSP is first-in/first-out. However, priority overrides are provided. These are primarily for job debug purposes, because the system manager will be providing the text editing, compiling, file management, etc., that constitute the usual time-sharing load on a scientific processing system.

The file memory controller is the key to fast file memory response, user control, low operating system overhead, and file security. On a file-open command by a user, the operating system in the BSP is invoked. This system determines the user's access rights to the file and then sets status bits in the file memory controller to correspond with these access rights. Subsequent references to the file by the user are done with in-line code in user mode, since the file memory

controller will not respond to an improper request. There are two potential "users", the current job running on the BSP, and the system manager. Both are treated in essentially the same way. Although, in the case of dealings with the system manager, the BSP operating system will also have to manage file memory space allocations before it responds to a system manager request and space deallocation after the system manager has removed a file from file memory. The file memory is paged and file references are logical addresses, which the file memory controller translates to physical addresses. Hence, a file need not occupy contiguous pages in file memory.

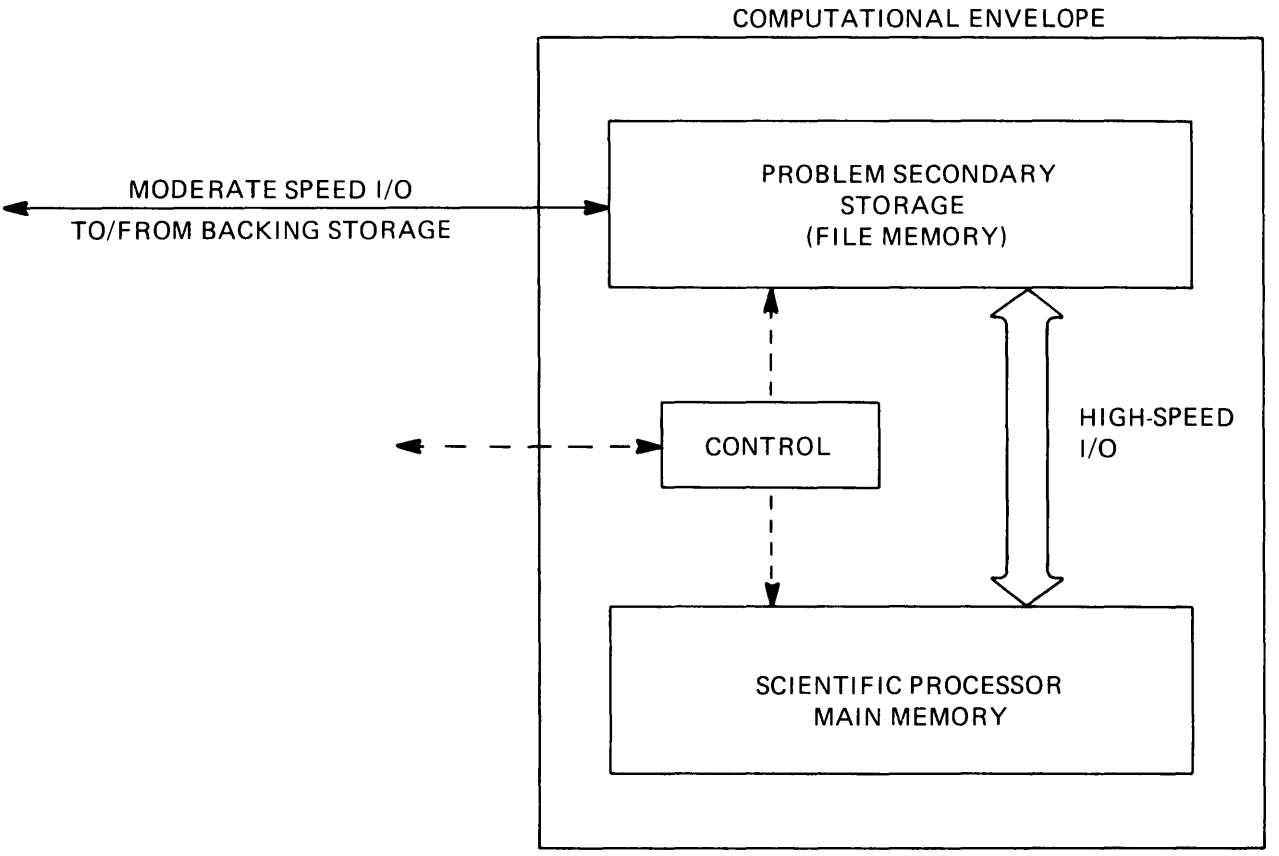


Figure 2. Scientific Problem I/O Characteristics

processed with reasonable efficiency. The idea is that a conversion may be done in manageable stages, with useful effect for one's efforts at each stage.

In summary, the BSP approach was to design a more general vector processor, and to forego the very fast scalar hardware. Is the science of parallelism too young for such a design? No one can say for sure. But the next few years should be revealing.

THE BSP DESIGN

The major BSP design elements include the system manager, I/O subsystem, parallel main memory, arithmetic elements, and scalar processor, parallel processor control, and the control and maintenance processor. Also included are BSP software, job flow, and the various user interfaces.

I/O Subsystem

In scientific computations, the dominant I/O patterns differ radically from those in the business data processing arena. With business data processing, small numbers of operations are performed on a very large data base. Also, the amount of main memory required to efficiently process a business data job is relatively small. Hence, in business data processing, I/O becomes a bottleneck, because of the limited number of operations performed on data while it resides in main memory. But, short of placing the entire data base in main memory, a given job does not demand too much memory to execute with adequate efficiency. This is an ideal environment for fostering multiprogramming. Many problems may reside in main memory at once. A few will be in an active state; the rest will be waiting in I/O.

The situation is quite different in the case of scientific computations. A given job usually requires a large amount of memory before it can execute efficiently. With present processor speeds and memory sizes, the larger bread-and-butter jobs execute best if each one has main memory to itself. In the case of many scientific jobs, some of the data on secondary storage is best regarded as an overflow of main memory — this data is what would not fit in main memory, but the programmer really wishes it were there. Hence, this overflow data is quite tightly coupled to the processing of data in main memory, and the programmer may want to exercise a great degree of control over the I/O process.

Compare such a situation with business data processing. In business data processing, the programmer is delighted to have sophisticated operating systems doing his I/O for him. And he is not concerned if the operating system is trying to optimize I/O for all the jobs in the mix. The scientific programmer resents such a situation. He wants as much memory as he can get, and then he wants direct control over I/O whenever feasible. For this reason, and due to details of particular hardware systems, many scientific programmers have reported spending the bulk of their programming effort trying to optimize I/O.

Such a state of affairs is unfortunate, because the overall flow of the high-speed I/O in most scientific problems is very simple. If the scientific programmer were not simultaneously battling an operating system, as well as often inadequate I/O devices, he could describe his I/O needs with a few simple statements.

In contrast with these difficulties, the scientific programmer has certain advantages which are irrelevant to the commercial programmer. For example, his file sizes are relatively small. Of course, immense file sizes may be a consideration in both cases for large problems. In general, however, scientific problems require much smaller files. Also, the scientific problem performs more operations on each data word it retrieves from secondary storage.

Further, the scientific problem programmer can typically state the flow of I/O. That is, the high-speed I/O flow is not usually data-dependent. In other words, efficient double-buffering I/O schemes are normally applicable.

Computational Envelope

How did all this affect BSP I/O design? The BSP design is based on the premise that the high-speed I/O and storage requirements be specified in what is called the computational envelope. The performance of the secondary I/O subsystem is designed to be sufficient to support the processor and main memory. This performance is completely under user control. Finally, for simplicity, a single I/O device, rather than a hierarchy, is used for this secondary storage system. (See Figure 2.)

Although the scientific problem program makes more extensive use of I/O data than does the business data program, the speed of present supercomputers is so great that no conventional I/O device can support them. Also, the access times associated with conventional devices are much too long. Because access time is a discontinuity that must be smoothed in I/O operations, slow access times imply large I/O buffers. If, at the same time, the transfer rate must be increased, then the buffers must be still larger. For many problems simulated in designing the BSP, cache buffer sizes would have approached half a million words, if disc technology were used for the secondary storage.

File Memory

Hence, the BSP secondary storage, called file memory, is based on semiconductor technology – 64-bit charge-coupled device (CCD) technology, to be specific. The results are average latencies well under one millisecond and sustainable transfer rates over 60 megabytes per second. Buffer sizes are reasonable and optimum performance is attained with simple standard FORTRAN statements. In other words, straightforward programming gets all the performance there is to get, and this performance is adequate to the task.

The BSP's memory system handles problem 2. The solution to problem 3 may be inferred from the reference already made to the very high level instruction set in the BSP. This same instruction set is part of the solution to problem 4. The needed high system utilization rate implied by problem 1 is gained in part by the parallel processor control unit, which is described later. And the BSP does take advantage of the emerging science of parallelism to help it gain an unusual speed on linear recurrences.

Due to what has become known as the "scalar problem", there is a substantial difficulty implicit in the simultaneous solution to problems 1, 4, and 6. The problem is easily described, but difficult to resolve. For example, imagine a linear vector processor that could process vectors at infinite speed, but could process scalars no faster than one operation every microsecond. Then, if the total problem comprised 90% vector and 10% scalar processing, the vectors would be done in no time at all, but the scalars would be done one operation per microsecond. Because only 10% of the problem would be scalars, one operation per microsecond would be divided by 0.1 to obtain an overall speed of 10 operations per microsecond on the example problem.

This is not especially fast because users now want at least 20 floating-point operations per microsecond. Yet the example is not unreasonable, because many vector machines, with maximum speeds over 50 floating-point operations per microsecond, have a difficult time sustaining 10 floating-point operations per microsecond.

Scalar Operations

Before discussing potential solutions to the problem of how to do scalars fast, it is beneficial to first explain what a scalar operation entails. This, however, is no simple task. First of all, some apparent scalars are not evident. For example, the memory indexing hardware on most vector computers fetches the entire linear vector, based only on some simple information such as start of vector, address difference between vector elements, and length of vector. Similarly, the execution of the vector operation is the same as executing an inner loop of a program. This means that many indexing operations, and much of the loop overhead present in an ordinary scalar machine, are eliminated as a result of the basic idea of the linear vector processor.

But certainly, some work remains, for example, generation of the simple vector descriptors referred to previously. Is this a sequence of scalar operations? Perhaps it is. On some vector machines, nothing else can happen while a vector is processed. The instruction processor can be busy retrieving a description of the next vector operation, while the present vector operation is executing. On the BSP, the SPU can describe and queue a sequence of vector operations, while a given vector operation executes. Vector setup operations are countable scalars on some machines, while on other machines, setups are counted only if they can not be overlapped with a vector operation already executing.

There are other situations in which machines are difficult to compare directly. For example, on the BSP the DO loop:

```
DO I = 2, N
A(I)  = C(I)  * A(I-1) + B (I)
END DO
```

is executed in parallel, with a maximum speed well over 10 operations per microsecond. On other vector machines, this common construct must be executed as a scalar sequence. And, if it is to execute rapidly, the vector machine must also contain a very fast scalar processor.

BSP Approach to Scalars

This is where the BSP parts company with the other recent vector machines. To solve this recurrence, and some other problems, conventional wisdom, at present, says a fast scalar processor must be included in the design.

But there are three major problems with this viewpoint. The first is that the fast scalar processor may be a high cost item. The second problem is more insidious, but probably more severe. To the extent that the compiler must choose between the use of the scalar hardware and vector hardware, the compiler has the job of compiling to two machines. This is probably sufficiently difficult that the compiler will be unable to generate object code for all the parallelism it has found. For example, if the scalar code is intimately dependent on the vector code, or vice versa, either the hardware must have extremely clever synchronizing mechanisms to tie the processors together, or the compiler must decide that some mixed code will arbitrarily be categorized as all being of one type.

The third problem is also insidious, and possibly, the most costly. This problem is that the arbitrary inclusion of a fast scalar processor, to solve a problem in an *ad hoc* way, almost guarantees that a successor machine from the same manufacturer will require a substantial reconversion effort. The successor machine is not likely to retain the structure of its predecessor.

For these reasons, although the BSP FORTRAN compiler will use the SPU for selected scalar operations, the BSP compiler is likely to treat a floating-point scalar as a vector of length one — or to treat a sequence of floating-point scalars as a non-linear vector operation sequence. This enables the BSP to forego the mixed blessing of the ultra-fast scalar unit. It allows the compiler to concentrate on capitalizing on detected parallelism. And it guarantees upward compatibility with a successor machine, recompilation being the maximum conversion penalty.

This approach also permits a smooth initial conversion to the BSP. In the beginning, a conversion may leave an undesirable amount of scalars. But, with uniform treatment of operands, a scalar does not have to be made part of a vector of length 100 to be processed efficiently. If it becomes part of a vector of length 3, then it is processed three times as fast as before. Vectors of length on the order of 10 are

A DIFFERENT KIND OF SUPERCOMPUTER

So far, this section has attempted to explain the basic rationale behind the current crop of supercomputers, namely, the linear vector. And, further, because of this basic rationale, the use of parallel arithmetic elements in the BSP and in the ILLIAC IV does not cause them to be fundamentally very different from the pipeline-based supercomputers. However, one important difference has been identified, that is, from the beginning, the BSP was intended to be paired with another processor, namely, the Burroughs B 7700/B 7800.

System Manager

In this respect, the BSP is somewhat akin to the IBM 3838 signal data processor. The IBM 3838, however, only executes functions or subroutines passed to it by its manager, whereas the BSP executes either entire programs or substantial portions of programs. Thus, the prime motivation for attaching an IBM 3838 to its manager is to enhance the power of the manager by off-loading. The basic motivation for attaching the BSP to a system manager, on the other hand, is to free the BSP for concentrating on processing large computational problems. A second motivation is to create a total system that features application and throughput capabilities not economically feasible with a single specialized processor.

Overlapped Instruction Mode

The BSP differs from its supercomputer alternatives in another important respect. Its instruction processor is loosely coupled to the parallel arithmetic elements. The approach is a generalizetion of the overlapped instruction execution mode in ILLIAC IV. ILLIAC IV runs more than twice as fast in the overlapped mode than in a nonoverlapped mode.

In order to achieve this overlap, the BSP has a queue between the instruction processor and the unit that drives the arithmetic elements. The queue is comparable to the ILLIAC IV implementation. In contrast, however, it contains hardware features that check for out-of-bound array references and optimize the choice between inner and outer FORTRAN DO loops. The latter feature facilitates such functions as the Fast Fourier Transform (FFT), which has an inner loop whose length is decreasing, while the next outer loop's length is increasing. In the BSP, this loop length optimization maintains a 256-point (or larger) FFT running at over 75% of maximum machine speed. This is because all vectors will be of length 16 or more (and hence efficient on 16 AE's), even though the programmer wrote a structure that implied vector lengths of 8, 4, 2, and 1 in the final FFT stages.

The BSP's ability to run fully overlapped surpasses the ILLIAC IV's ability to run fully overlapped. Whereas the ILLIAC IV's instruction processor must call on the parallel section for instruction storage and for some arithmetic operations, the BSP's instruction processor, called the scalar processing unit (SPU), has full arithmetic capability. The SPU is also equipped with local memory called control

memory (CM), which is used for storage of indexing parameters, and vector descriptors. In total, these features further the overlap implementation between vector instruction processing and vector instruction execution introduced with the ILLIAC IV.

Linear Vector Approach to Parallelism

The last basic difference between the BSP and supercomputer alternatives is perhaps the most controversial. It stems from the BSP's timing in the evolution of linear vector-based supercomputers.

In designing the BSP, some experience had been accumulated relative to the ways in which the linear vector approach to parallelism could be applied to real world problems. In this respect, it is not unreasonable to assert that the BSP is the forerunner of a second generation of supercomputers.

What substantiation is there for this rather strong assertion? The following is a list of some ideas or problems that were understood when the BSP design started:

1. Maximum speed is not nearly as important as sustainable speed.

2. A one-dimensional memory – one that is efficient only for linear vectors whose elements are packed adjacent to one another – is not sufficiently general.

3. Assembly language level programming is almost incompatible with linear vector programming. Even the set of FORTRAN primitives cannot directly express many simple linear vector constructs. If the programmer is to think effectively about his problem at the linear vector level, he must be insulated from concern with machine details.

4. It is possible to construct FORTRAN program analyzers which find a large percentage of the intrinsic parallelism in programs. However, if the target machine structure is not simple and general at a high level, an analyzer cannot create useful object code from the parallelism it has found.

5. Although the use of parallelism still has many vestiges of black art practice, a science is beginning to emerge. In particular, linear recurrence relations are now known to be susceptible to parallelism.

6. Conversion to a linear vector machine should be accomplished once. Any new design should consider the future, so the user will not confront difficulties converting to a successor machine.

BURROUGHS SCIENTIFIC PROCESSOR

IN PERSPECTIVE

THE BSP – A NEW APPROACH

Early in 1973, Burroughs assembled a select team to design a commercial supercomputer. In 1977, the team's efforts resulted in the Burroughs Scientific Processor (BSP) – a product that presents a new approach to large-scale computational processing. This section places the BSP in perspective, discusses its more interesting features, and describes various design trade-offs.

The BSP uses a large-scale Burroughs B 7700/B 7800 general-purpose computer as a system manager. As a result, the global structure of the BSP itself is simple. It consists of an instruction processor, a set of parallel arithmetic elements, a main memory, an instruction or control memory, and a single I/O device (Figure 1).

This I/O device is called file memory. It is controlled by the BSP's instruction processor. It functions as a high-speed I/O device for programs running on the BSP and as a staging point for lower-speed I/O going to or coming from the system manager.

The BSP parallel processor has 16 parallel arithmetic elements (AE's) driven in lock-step by a single instruction stream. Hence, the BSP is a single-instruction, multiple-data stream (SIMD) architecture. In this respect, it is comparable with other large pipeline or parallel scientific processors.

LINEAR VECTORS

Single-instruction, multiple-data stream (SIMD) machines were designed to process "linear vectors". A vector is an operand consisting of a series of numbers or values. A linear vector is a vector that is mapped into storage in a linear fashion; the addresses of the constituents differ by a constant. Such vectors are the most elementary vectors that can be formed by looping structures in programming languages (DO loops, etc.). Linear vectors are naturally generated when programming language array element index expressions are linear functions of loop parameters.

It is this latter fact that has caused the SIMD architecture to emerge as the front-runner in attempts to gain increased scientific processing speed through parallelism. That is, once parallelism is selected, it must be noted that the bulk of scientific processing involves processing contained within looping structures. The simplest array element describable in looping structures is a single quantity, a scalar. However, parallelism requires operations on more than one object at once. This being so, the simplest data type susceptible to parallelism is the linear vector.

The linear vector has two significant advantages relative to other parallel operands. First, it is straightforward to design hardware that will efficiently fetch linear vectors from memory, under the control of a simple vector descriptor. The second advantage is that, inside a loop structure, the same operation is specified between all the consecutive element pairs of a pair of vector operands. Together, these two advantages imply that, while operations between linear vectors can be done using parallel hardware, the control of such operations can be from a single instruction using a simple data descriptor. Consequently, the relatively simple SIMD architecture provides sufficient control capability to exploit this particular kind of parallelism.

The SIMD architecture has previously appeared in several forms: 64 processing elements, with their private memories, driven by a single instruction processor in the ILLIAC IV; sets of pipelines, each performing a part of an arithmetic operation, as in the CDC STAR, TI ASC, and CRAY-1. Regardless of the nature and method of implementation, however, all of these machines, including the BSP, have been designed to function optimally with linear vectors as operands. Hence, it is reasonable to categorize all of them as linear vector machines, or, more commonly, vector machines.

Because the linear vector is the basic programming entity, the BSP's instruction set is designed around the concept of linear vectors of arbitrary length. The granularity in vector operations, caused by the fact that 16 arithmetic elements do 16 identical things at once, as well as the need to properly manipulate vectors whose lengths are not integer multiples of 16, is handled automatically by the control hardware. The BSP FORTRAN compiler is unaware that there are 16 AE's. The compiler simply deals with vectors and issues vector instructions.

BSP DESIGN . . . meets the specific requirements of large-scale scientific processing

What are these requirements? First, the performance level of supercomputers requires some type of concurrent computational capability. Second, the bulk of operations characterizing scientific computation are floating-point numerical operations, indexing and branching. Third, many large codes have execution times best measured in terms of hours; some require days. Fourth, a key characteristic of scientific programs (and one that distinguishes them from commercial business codes) is that they generate and regenerate their own data bases, often in a very regular way. This feature confines high-speed I/O to an envelope containing the floating-point processor and a fast secondary store. Fifth, the scientific marketplace is FORTRAN-dominated with codes involving man-years of preparation and tens of thousands of source statements.

The BSP has been designed to meet all these requirements.

BSP SUPERIORITY . . . is based on several significant advantages over other systems in the supercomputer class

Clearly, the BSP is a superior performer. It is competitively priced. The machine derives its performance capabilities from a number of considerations. The BSP is a total system, combining a most advanced general-purpose processor with a floating-point system of exceptional speed. Its design philosophy is such that extensibility is an integral part of it.

Another significant feature of the BSP is its reliability. The system has been constructed from standard BCML circuits and packages. All paths to and from memory are equipped with error-correcting capability (SECDED). In addition, there is residue checking on all arithmetic operation, instruction retry, and an extensive set of on-line system device diagnostics.

Because of these features offered by the BSP, Burroughs can expand its market potential and extend its competitive range.

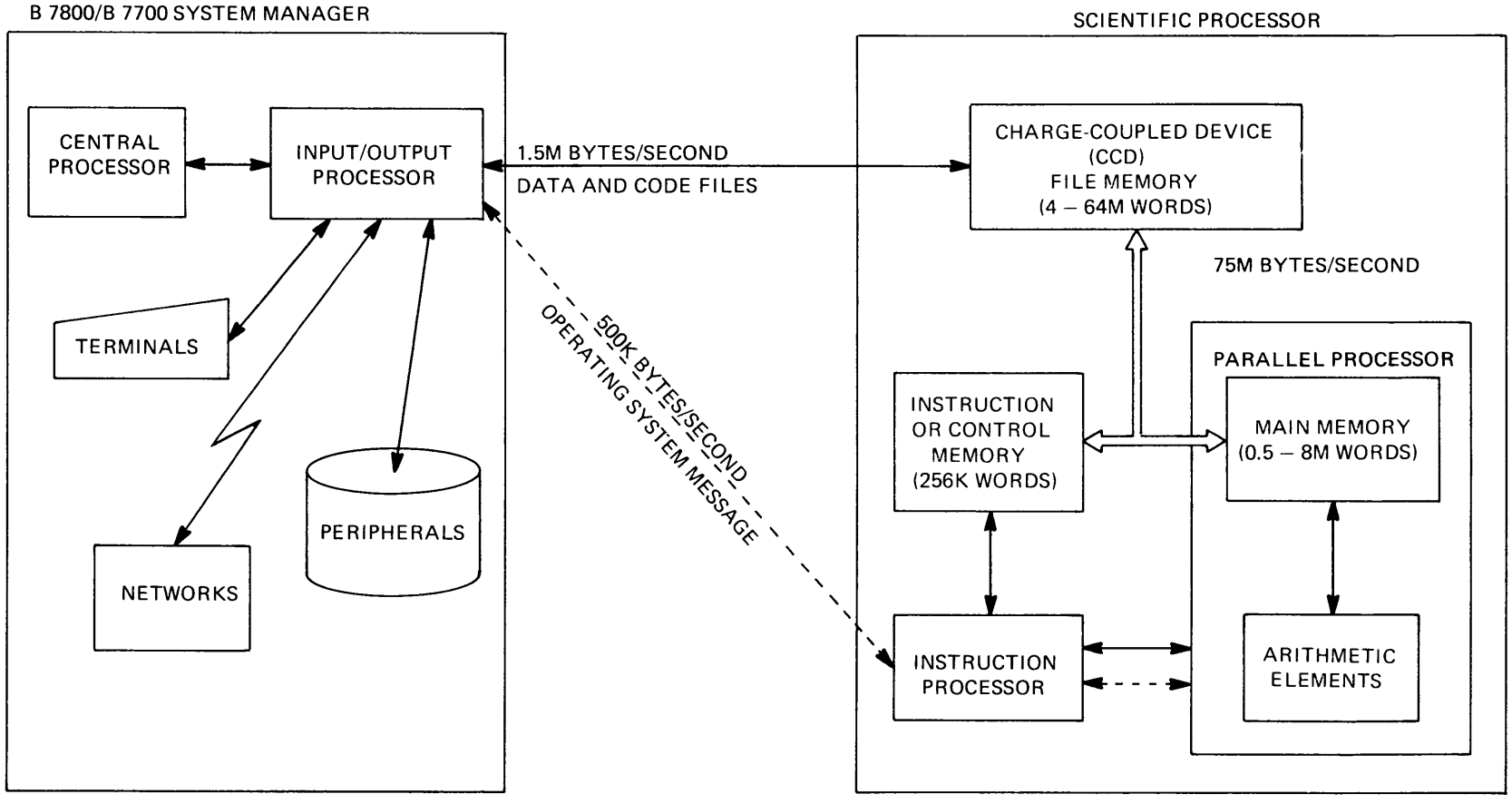


Figure 1. BSP System Block Diagram

Vector Performance

The parallel architecture equips the BSP with an outstanding performance capability. A commonly used figure of merit in scientific computations is the number of million floating-point operations per second (MOPS). For vector lengths greater than 16, the system has the performance potential of 50 MOPS.

Performance Optimization

The BSP has three particular hardware and software design features that influence performance.

First, the BSP is equipped with the capability of handling recurrences. The system can detect and transform expressions of the form:

```
A(I) = A(I-1) * B(I)
```

This is a particularly useful capability because such expressions appear to be scalar in nature.

Second, the indexing hardware on the system is able to reorder DO LOOPs. This is important because long vector computations are more efficiently processed than short vector computations. For example, the expression,

```
      DO 4  I = 1, 70
      DO 4  J = 1, 5
4     A (I, J) = B(I, J) * C(I, J)
```

as it appears here consists of 70 computations on vectors of length 5. But there are no reasons (data dependencies, special sequencing) why these loops could not be inverted to:

```
      DO 4  J = 1, 5
      DO 4  I = 1, 70
4     A (I, J) = B(I, J) * C(I, J)
```

so that there are now five computations on vectors of length 70.

Finally, the system has the capability to handle conditional statements in parallel by using "bit" vectors. These are sequences of ones and zeros that can be used to mask out unwanted results.

FILE MEMORY

One of the truly significant innovations in the BSP is the file memory. It serves as the secondary storage system for the parallel processor memory, and is important because of the greatly enhanced performance capability it gives the BSP.

On most systems (even supercomputers), secondary storage is provided by discs. In supercomputers this can be a problem because the rate at which information can be transferred from secondary storage to main memory is simply not matched to the tremendous processing capability of the CPU. In fact, for several classes of problems where the program and data spill out of main memory onto discs, overall system performance can be very seriously degraded.

The most important feature about the file memory for BSP performance is that it sustains a transfer rate to parallel memory of 10 M words/second, complementing the processing capability of the AE's well and providing system balance.

VECTORIZING FORTRAN COMPILER

One of the very strongest assets of the BSP is its software. The BSP is the first supercomputer developed as a total system, and that concept extends to BSP software. The BSP is provided with a mature operating system (the MCP) and a vectorizing FORTRAN compiler.

What does vectorizing mean? It is merely the recognition of computational sequences that can be done simultaneously.

On a serial or scalar processor, the sequence of computations

```
DO 10 I = 1, 100
10 A(I)  = B(I) + C(I) * D(I)
```

would be done one at a time.

In examining a code, the vectorizing compiler recognizes that such sequences can be done simultaneously. It is, therefore, a means of converting scalar or sequential programs into parallel programs.

Users will also be able to program in FORTRAN exclusively. No assembly language programming will be necessary to achieve the performance of the BSP.

For new program development, the language will also be equipped with vector extensions that will allow for the introduction of parallel computing concepts from the beginning.

BSP KEY FEATURES . . . include a system manager, the BSP elements, and a vectorizing FORTRAN compiler

The system manager is responsible for overall BSP job scheduling and control. Through it, program preparation and data input and output are accomplished. It serves as the device for interactive program preparation and debugging and provides archival storage.

The control processor portion of the BSP is a high-speed, asynchronous unit that controls the parallel processor and performs scheduling, file allocation, and I/O management. It is characterized by an 80-nanosecond cycle time (12.5-megaHertz clock) and is equipped with 262K words of 4K MOS memory with an access time of 145 nanoseconds. The control processor also serves to interface the BSP with maintenance and diagnostic devices.

Programs to be run on the BSP are compiled on the system manager using a vectorizing FORTRAN compiler, which is a significant part of the system software. It is used to maximize the efficiency of the BSP across a wide spectrum of scientific applications.

BSP ORGANIZATION . . . consists of three basic units: control unit, parallel processor, file memory

The control unit is made up of a "scalar" processor unit that handles setup of vector operations for the parallel processor, 262 K words of memory in which the program to be executed is stored, a parallel processor control unit that sets up vector calculations, and a control and maintenance unit that is used to interface the maintenance features of the system manager to the BSP.

The parallel processor is made up to 16 arithmetic elements (AEs) connected to a parallel processor memory by means of alignment network. The network is a cross-bar switch that connects the parallel memory banks to the AEs and is used to guarantee conflict-free memory access.

The BSP is completed by the file memory that consists of charge-coupled device (CCD) storage media and a file memory control unit.

BSP CHARACTERISTICS . . . include the parallel processor, file memory, and vectorizing "FORTRAN compiler"

PARALLEL PROCESSOR

The parallel processor portion of the BSP is designed to perform "vector" oriented computations at a very high rate. The BSP itself is a single instruction stream/multiple data stream computing device. The high execution rate is achieved by partitioning computations onto the 16 arithmetic elements of the parallel processor.

Consider the following FORTRAN statement:

```
DO IO I = 1, 1000
A(I)    = B(I) + C(I) * D(I).
```

The sequence of computations performed is:

```
A(1)    = B(1) + C(1) * D(1)
A(2)    = B(2) + C(2) * D(2)
A(N)    = B(N) + C(N) * D(N).
```

Quite obviously, there is no dependence in these expressions of A(N) on (N-1). That is, the computations are independent of one another. There is, therefore, no reason not to perform these computations simultaneously. That is, if there were an ensemble of arithmetic elements (AE_1, AE_2, AE_3, etc.) then at the same time that A(1) was being computed in AE_1, A(2) could be computed in AE_2, A(N) in AE_n, and so forth. This is the basic idea behind the computational philosophy of the BSP. What makes the philosophy truly usable is that large classes of scientific problems exhibit this type of computational concurrency.

Conflict-free Memory Access

One of the key reasons the BSP is able to sustain such tremendous computation rates is the conflict-free memory access. The system is designed so that the number of memory banks is relatively prime to the number of processing elements. With this design decision, it is possible to map data into the parallel memory so that rows, columns, diagonals (in fact, any sequence of elements in a two-dimensional array that are regularly spaced) can be accessed at full memory bandwidth. This situation contrasts with other supercomputers in which only rows or columns or diagonals can be accessed at the full rate.

BURROUGHS SCIENTIFIC PROCESSOR

AN OVERVIEW

BSP OBJECTIVE . . . to extend Burroughs product excellence into the domain of very high-speed scientific computing

Traditionally, Burroughs has been very strong in covering the entire spectrum of general-purpose data processing, from the B 80 to the B 7800. With the BSP, Burroughs is complementing these general-purpose capabilities with a most innovative and powerful scientific system.

The demands of large-scale scientific processing impose vigorous requirements on the machine that supports it. The BSP has been designed to meet and surpass these requirements. In particular, the BSP is a very high-performance system. It is in the "supercomputer" class of machines and will deliver floating-point results up to 50×10^6 operations per second. In contrast with other currently available supercomputers, the BSP is a total system. It combines the general-purpose processing capability of a Burroughs large-scale or very large-scale system with exceptional floating-point capability.

Burroughs has chosen to build the scientific processor from a new circuit family, BCML (Burroughs Current Mode Logic). As a result, the BSP enjoys high reliability, availability and maintainability and is exceptionally cost-effective.

Finally, there is a large degree of extensibility inherent in the BSP design. The system has an impressive potential for modular growth.

BSP SYSTEM . . . consists of the system manager and the scientific processor

The system manager (typically, a B 7800) schedules and allocates tasks to the scientific processor. It supports networking and data communications and has a complete set of peripherals (printers, disks, tapes) that can be extended to include archival storage.

The scientific processor consists of the control processor, the parallel processor, and the file memory.

There are three basic models of the BSP. For the user who already has a B 7700 or B 7800, there is the basic BSP. For other users, the basic configurations are the BSP/7811 and BSP/7821.

- Basic BSP - 16 arithmetic elements, 524K words of parallel processor memory, a control processor with 262K words of memory, and 4 M words of file memory.

- BSP/7811 - a BSP with a B 7811 system manager. The B 7800 has one central processor, one I/O processor, one maintenance processor, and an operator display console with dual displays.

- BSP/7821 - a BSP with a B 7821 system manager. The B 7821 has two central processors, two I/O processors, one maintenance processor, and an operator display console with dual displays. Dual BSP interface adapters provide a connection between the BSP and both B 7800 I/O processors. However, only one interface adapter is active at any one time; the other is used for backup or system reconfiguration.

TEMPLATES

The problem of keeping the AE's, the alignment network, and the memory simultaneously busy is interesting. Collectively, these global elements form a pipeline. That is, data is first fetched from memory, transmitted to the AE's over the alignment network, processed by the AE's, transmitted back to memory, and stored. In total, this is a five-step process, executed with four major elements (memory, input alignment network, AE's and output alignment network). The parallel processor control unit keeps these system elements as busy as possible relying on precoded microinstructions called "templates".

A template is a description of an entire sequence of operations that a group of associated sets of 16 numbers follows. For example, such a group of sets of numbers could be the addition of 16 elements of "A" to 16 elements of "B" to form 16 elements of "C". In other words, one template can be used to control 16 arithmetic operations which can be done simultaneously by the 16 AE's, plus all the associated memory and data transmission operations.

The problem that the parallel processor control unit must solve is the interleaving of one such set of operations, or template, with the next set. In general, one template will not be identical to the one that follows it. For example, a succeeding template may be generating the first 16 elements of Z = Q * R + P , while the forerunner template is doing the last 16 (or less) elements of C = A + B.

The reason for wanting to interleave dissimilar types of templates is that, if it is not done, then the pipeline formed by memory, the alignment networks, and the AE's must be emptied between the completion of one vector operation and the start of the next. If this were to happen, then the BSP would suffer from the same vector startup problem that has plagued other pipeline machines. The manifestation of this problem to the user is that the machine is inefficient for operation on short vectors because of the startup idle time.

Given that a template is a microsequence specified at system design time, the job of the parallel processor control unit is substantially simplified. Instead of having to efficiently interleave the control of several dissimilar units, the parallel processor control unit simply has to select the next stored sequence. BSP templates can be characterized satisfactorily by two numbers. One number specifies the clock count between a template's last fetch from memory and its last store into memory. In other words, the template leaves this many memory clocks available for the next template. The other number is the number of memory clocks the template needs at its beginning. This number must be less than or equal to the number of clocks left by the preceding template.

For example, let a template be T_1 (2, 3). This means the template needs two contiguous memory clocks to start up, and it leaves three contiguous memory clocks between its last fetch from memory and its last store into memory. If another template is T_2 (3, 5), then the sequence T_1 (2, 3) followed by T_2 (3, 5) would have complete overlap between T_1 and T_2, with no wasted cycles. If one used the sequence T_1 (2, 3) followed by another T_1 (2, 3), there would be one

clock lost in the interface between the two templates. And, of course, if a T_1 (2, 3) is followed by a T_3 (4, 2) there are four wasted clocks, because 4 is not less than or equal to 3. In the BSP, an adequate number of template choices exists so that the average time lost between two dissimilar templates is small.

Template control entails the selection between templates already in control storage. The criterion of the selection is optimized efficiency of the system. Clearly, the power of the processor required to make this selection is miniscule compared with the power required to attempt dynamically to optimize system utilization.

There is an extra bonus attributable to using templates in the parallel processor control. This is the ability to implement retry on vector operations. Upon detection of a noncorrectable error, the control lets the last successful template finish, resets any partially started templates back to the start point, and retries the template which failed. The BSP is the only supercomputer that features vector operation retry.

A problem can occur in a system that has this much overlap. The problem is called a vector "hazard". For example, if A = B + C is followed by Q = A * R, then the elements of A must be stored by the first statement before they are used in the second. If the vector operations are long, no problem exists. If they are short, it may be necessary to hold up the second operation until the first has finished, even though this costs some lost cycles. The parallel processor control unit in the BSP automatically detects and solves this problem situation. The template control processor adopts a different strategy in this case. Instead of maximizing overlap, it selects templates which minimize time lost between operations.

ARITHMETIC ELEMENTS

All 16 arithmetic elements (AE's) in the parallel processor are identical. The set of 16 is driven from a single microsequence, in keeping with the SIMD nature of the machine.

Each arithmetic element is quite soft, in the sense that only the most primitive operators are hardwired. The control word is over 100 bits wide. In part, this large control word width is due to direct access to primitive functions; it is large because the arithmetic element has an unusual range of processing capability. That is, besides being a floating point machine, the AE has substantial nonnumeric capability. A comprehensive set of field manipulation and editing operators is available. Additionally, a special set of operators is available specifically for FORTRAN format conversion. While the BSP is marked as a floating-point number processor, in actuality, with its charge-coupled device (CCD) file memory, exceptionally large main memory, and versatile AE's, it may well represent the largest available nonnumeric processor.

Floating point add, subtract, and multiply each take two memory clocks in an AE. The use of two clocks enables memory bandwidth to be balanced exactly with AE bandwidth for triadic operations. A triadic operation is defined as having three operands and one result. Evidently, this does result in balance, because the four memory clocks required to handle the operands and result are balanced by the four clocks required by the two arithmetic operations, which convert three operands into one result.

The BSP memory system cycle time is 160 nanoseconds. The AE cycle time is the same. In this day of cycle times much shorter than 160 nanoseconds, it is reasonable to wonder at such a long cycle time.

A shorter AE clock would have allowed more effective utilization of AE parts because they would have been used more often per operation. However, offsetting factors were the high level of integration in the ECL circuits, and the desire for ease of manufacturing and straightforward maintenance through absence of need to fine tune the clock signal. To some extent, accumulated gate delays at the level of integration required a long clock (although certainly not as long as 160 nanoseconds), but primarily the level of integration made the long clock affordable. The extra ICs did not substantially impact total system size.

Floating-point divide is implemented by generating the reciprocal in a Newton-Raphson iteration. The square root is performed in the hardware as well. It is also implemented via the Newton-Raphson algorithm. ROMs exist in each AE to give the first approximations for the divide and square root iterations. One advantage of using parallel AE's, instead of pipeline implementation, is the relative ease with which a full-length divide and square root can be generated.

The single-precision, floating-point format is 48 bits long. It has 36 bits of significant exponent and 10 bits of binary exponent. This gives a range of 10 ± 307 and about 11 decimal digits of precision. The floating point arithmetic is done using guard bits and efficient rounding algorithms. Even for large problems, it is rare for more precision to be needed. Double precision is available, however, should added precision be required.

The AE has double-length accumulators and double length-registers in key places. This permits direct implementation of double-precision operators in the hardware. The AE also permits software implementations of triple-precision, etc.

Note that with 16 AE's, each generating an add, subtract, or multiply in 320 nanoseconds, and with parallel-processor control fully overlapped, the maximum speed of the BSP is 50 million floating-point operations per second. While sustained operation at this maximum speed will be infrequent, it should be evident that overall design philosophy has been to produce a machine which can sustain a reasonable fraction of its maximum operation rate.

4 X 5 ARRAY

n

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}

STANDARD FORTRAN COLUMNWISE MAPPING

ARRAY ELEMENTS	a_{11}	a_{21}	a_{31}	a_{41}	a_{12}	a_{22}	a_{32}	a_{42}	a_{13}	a_{23}	a_{33}	a_{43}	a_{14}	a_{24}	a_{34}	a_{44}	a_{15}	a_{25}	a_{35}	a_{45}
MEMORY ADDRESS = $\underline{a}$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

LINEAR VECTOR COMPONENTS ARE SEPARATED BY A CONSTANT ADDRESS DISTANCE $\underline{d}$. FOR EXAMPLE, COLUMNS HAVE $\underline{d} = 1$, ROWS HAVE $\underline{d} = n$, FORWARD DIAGONALS HAVE $\underline{d} = n + 1$, ETC..

Figure 4. Standard Array Memory Mapping Features

M = THE NUMBER OF MEMORY BANKS
N = THE NUMBER OF ARITHMETIC ELEMENTS

CHOOSE M TO BE A PRIME NUMBER.
CHOOSE $N \leq M$.

THEN, FOR ADDRESS $\underline{a}$:

MEMORY MODULE NUMBER: $\mu = \left|\underline{a}\right|_M$

OFFSET IN THE MODULE: $i \left\lfloor \frac{\underline{a}}{N} \right\rfloor$

FOR EXAMPLE, IF M=7, N=6, THE 4 X 5 ARRAY IS MAPPED:

ARRAY ELEMENTS	a_{11}	a_{21}	a_{31}	a_{41}	a_{12}	a_{22}	a_{32}	a_{42}	a_{13}	a_{23}	a_{33}	a_{43}	a_{14}	a_{24}	a_{34}	a_{44}	a_{15}	a_{25}	a_{35}	a_{45}
a =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
μ =	0	1	2	3	4	5	6	0	1	2	3	4	5	6	0	1	2	3	4	5
i =	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3

Figure 5. The BSP Memory Mapping Algorithm

CONFLICT-FREE MEMORY ACCESS

A unique feature of the BSP is its memory system which delivers a useful operand to each AE, per each memory cycle. That is, the distance in memory between elements of a vector need not be unity. Therefore, DO loops may contain nonunity increments, or the program may access rows, columns, or diagonals of matrices without penalty.

This kind of memory capability has not been available before with memory parts of modest speed. Supercomputer designers have elected either to use memories with severe access restrictions, or have used expensive fast memory parts to attain a degree of conflict-free access through bandwidth overkill.

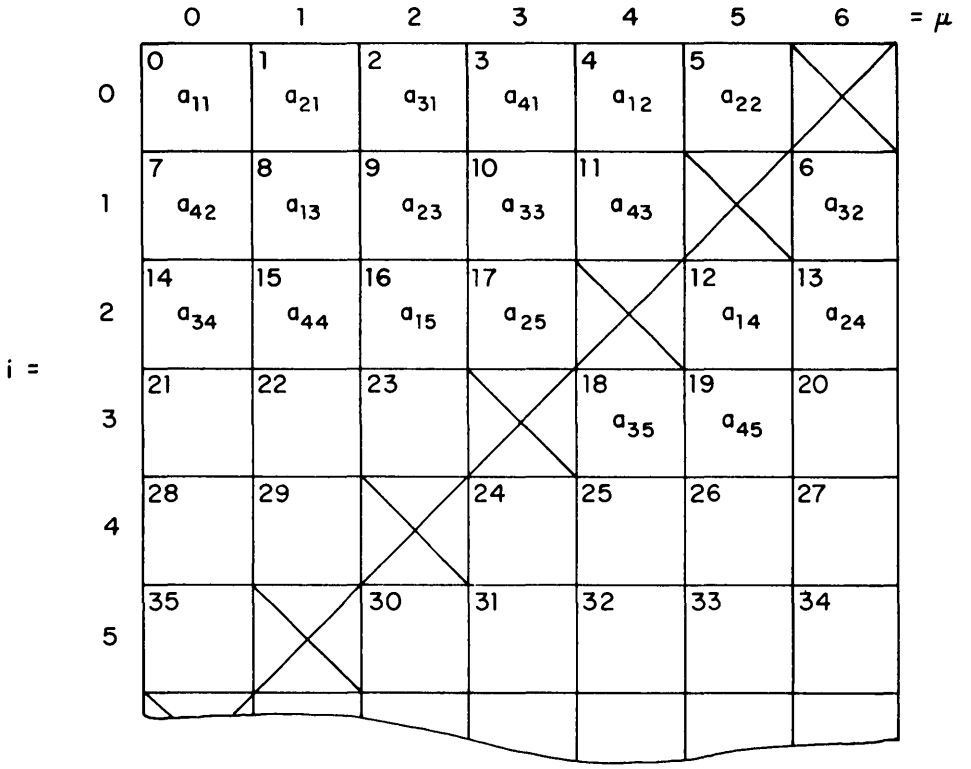
The hardware techniques used to ensure conflict-free access are a prime number of memory ports, full crossbar switches between the memory ports and the AE's, and special memory index generation along with crossbar switch tag generation.

The index and tag generators compute the proper addresses for a particular address pattern. This address pattern is the one used by orthodox serial computers. That is, each higher memory address refers to the "next" word in memory. With this pattern, the parallel memory is completely compatible with all the constructs of present programming languages. In particular, FORTRAN EQUIVALENCE, COMMON, and array parameter passing can be implemented in the same way as on a conventional computer.

As an example, consider Figure 4. This shows a 4 by 5 matrix mapped columnwise into the memory of a serial machine. For purposes of illustration, assume a 6 AE, 7 memory bank parallel machine. (The BSP has 17 memory banks.) The index and tag equations are shown in Figure 5. The index is the floor of the integer quotient of the address $\underline{a}$ divided by the number of AE's. Thus, the index will remain constant for a cycle equal to the number of AE's; then it will jump by one value. On the other hand, the tag (or memory bank number in which the value associated with address $\underline{a}$ is stored) will be $\underline{a}$ modulo the number of memory banks.

Hence the tags will be repeated cycles of the same values, with no value repeating in one cycle, and the length of the cycle equal to the number of memory banks. As long as the number of AE's is less than or equal to the number of memory banks, the sequence of tag values will cause a different memory bank to be connected to each AE. Thus, each AE may receive or send a unique value. The particular storage pattern produced in this 6 AE, 7 memory bank system for the 4 by 5 example array is shown in Figure 6. Figure 7 shows index and tag calculations for the second row of the array.

Note that the equations yield an AE centrist vantage point. That is, the first logical element of the vector goes to the first AE, etc. There is nothing special about this approach beyond a certain confort in thinking. The important point is the following: As long as the same set of equations is always applied to the data, from the first time it comes in as I/O onward, then the storage pattern is completely invisible to the user. This applies to program dumps, etc., as well because the hardware always obeys the same rules.



NOTE THAT FOR $|\underline{d}|_\mu = 0$, ALL ADDRESSES ARE IN THE SAME MEMORY BANK. OTHERWISE, THERE IS NO CONFLICT AT ALL FOR A LINEAR VECTOR.

Figure 6. The Physical Mapping of the Example Case

IF ROW 2 IS WANTED, THEN:

START ADDRESS IS 1.
SKIP DISTANCE $\underline{d}$ IS 4.

$$\mu = |1|_7, \quad |5|_7, \quad |9|_7, \quad |13|_7, \quad |17|_7,$$

$$= 1, \quad 5, \quad 2, \quad 6, \quad 3$$

$$i = \left\lfloor \frac{1}{6} \right\rfloor, \quad \left\lfloor \frac{5}{6} \right\rfloor, \quad \left\lfloor \frac{9}{6} \right\rfloor, \quad \left\lfloor \frac{13}{6} \right\rfloor, \quad \left\lfloor \frac{17}{6} \right\rfloor$$

$$= 0, \quad 0, \quad 1, \quad 2, \quad 2$$

REFER TO FIGURE 4 TO SEE THAT ARRAY ELEMENTS 0,1,2,3,4 RECEIVE $a_{21}, a_{22}, a_{23}, a_{24}, a_{25}$ RESPECTIVELY.

Figure 7. Index and Tag Calculations Used to Fetch Row 2 in the Example

The unused memory cells are the result of having one less AE than there are memory banks. For the example case, this is not a useful occurrence. For the real BSP, with 16 AE's and 17 memory banks, division by 16 is much simpler and faster than division by 17. However, one then pays the penalty of supplying some extra memory to reach a given usable size.

Note that conflict does occur if the addresses are separated by an integer multiple of the number of memory banks. In this case, all the values one wants are in the same memory bank. For the BSP, this means that skip distances of 17, 34, 51, etc., should be avoided. In practice, 51 is a likely problem skip. This is because it is the skip of a forward diagonal of a matrix with column length 50. If conflict occurs in the BSP, the arithmetic is performed correctly, but at 1/16 the normal speed. The system logs the occurrence of conflicts and their impact on the total running time. This information is given to the programmer for corrective action, if the impact was significant.

BSP memory reliability has been mentioned. Diagnosibility is also an important feature. Instead of placing the Hamming code generators, detectors, and correctors on the memories, as is usual, they are placed at the AE's. This way the entire loop, from the AE's to the memory and back again, is Hamming code corrected. A side benefit of the conflict-free access hardware is that control information can be buried in the Hamming code in such a way that failing control elements can easily be detected and identified. Hence, not only are memory elements and data paths checked in the BSP design, but control logic is checked as well.

PARALLEL PROCESSOR CONTROL UNIT

Many of the functions of the parallel processor control unit have been mentioned. The unit is essentially a pipe, with each stage concerned with translating or checking input instructions into accurate control signals for the parallel processor.

The first stage in this pipeline is called the vector input and validation unit (VIVU). The vector and array instructions, assembled by the SPU, are inserted at this point. The VIVU assembles a sequence of such instructions into a single global description of an operation defined over a set of program arrays. It also establishes the relationship between this operation and the one before, to guard against vector hazards.

The form of instruction inserted by the SPU into the VIVU is interesting. In keeping with Burroughs policy of designing standard product computers as language processors, it was determined early in the BSP design cycle that the machine would process assignment statements. Arithmetic assignment statements are the basic ingredient to most numerical processing. The templates are memory-to-memory entities, because assignment statements are memory-to-memory statements.

In the case of Burroughs stack machines, such as the B 7800, the source language translates almost directly into object language with little need for run-time processing to aid in the description of the object language. Hence, in the B 7800 this run-time processing is automatically invoked in the hardware.

It was not possible to take the same approach with the BSP. The most general BSP parallel processor construct is a sequence of vectors, essentially a nested pair of DO loops. Because in the general case, all of the parameters used to describe an operation could be run-time variables, so much language processing is involved that it makes no sense to try to do it in automatic hardware. For example, parametrized descriptors of operations on sets of vectors, where the parameters are computed at run time, involve a great deal of run-time processing to convert the parametrized source language into object code.

This general consideration defines the need for the SPU as well as the point in the processing sequence at which the SPU passes instructions to the parallel processor control unit. This point is where run-time source language processing ceases, and all subsequent vector operation processing can be done in a uniform way. From a language point of view, this point is the time at which actual parameters can be inserted into a formal assignment statement.

This is, consider the triadic formal assignment statement

$$RBV\ ,\ Z = A\ \ OP_1\ \ B\ \ OP_2\ \ C,\ \ OBV\ \ ;$$

where Z, A, B, and C are vector descriptors, op_1 and op_2 are operators, and OBV/RBV are the optional operand and result bit vector descriptors.

The executing this operation, the SPU issues the following sequence to the parallel processor control unit:

VFORM	TRIAD, OP_1, OP_2
OBV	
RBV	
VOPERAND	A
VOPERAND	B
VOPERAND	C
VRESULT	Z

The information in the VFORM instruction will name the actual operators to be used, designate the level of loop nesting, indicate presence of bit vectors, etc. The OBV and RBV descriptors will give a start address and a length. The starting addresses are not referenced to an array, because bit vectors are not FORTRAN concepts. However, the vector operand and result descriptors give the start of the vector relative to an array location, the location of the array, the volume of the array, the skip distance in the location of the array, the volume of the array, the skip distance in memory between vector elements, and the optional skip distance between the start of vectors in a nested loop pair. Some consideration should convince the reader that this is the point where run-time language processing has ceased. The remaining processing, for example, array bounds checking, will be constant for all operations. Hence, it is seen that the BSP is a FORTRAN language processor.

After the VIVU has processed the vector form and its associated operand and result descriptors, the finished package description is stored in the template descriptor memory (TDM). The TDM functions as a queue, thereby permitting many vector forms to be present in it at once. The template control unit (TCU) fetches information from the TDM, updates it, and stores the updated information back in the TDM. The function of the TCU is to drive the parallel processor. It does this by selecting an appropriate template and then using the address information in the TDM to generate tags and indices for the quantities dealt with by the template. Because a template normally processes sets of 16 elements, the TCU normally updates addresses by 16 and stores the new addresses back into the TDM. However, for a memory conflict case, or compress/expand/merge operations, the TCU adds the correct updating number to addresses before restoring them to the TDM.

For retry, checkpointing, and arithmetic fault identification, the TDM also contains a copy of the original vector form, as received from the VIVU. The program-address counter value of the instruction which issued the vector form is also stored in the TDM. In the case of program termination due to arithmetic faults, the programmer is given the index of the first operand pair causing a fault in a vector form, the nature of the fault, the line number of the associated source statement, and the calling sequence used to reach that statement.

If the TCU is processing the last template of a vector form, there will generally not be sets of 16 elements involved. This is because the last template will require only enough elements to process the vector length specified by the program. For this case, the TCU will pad the partial lengths out to 16 using "null" operands. The null operands are generated by the input alignment network aid have the following properties in the AE's:

operand ◄——— "null" (operator) operand

operand ◄——— operand (operator) "null"

Also, memory bank will refuse to store a null. Hence, "null" times 1 equals 1, "null" plus 1 equals 1. And, of course, two nulls produce a null. So, a vector that is not an integer multiple of 16 is padded out with a quantity which results in no improper stores occurring. But the null is more than that. For example, in a sum reduction, where numbers are being transmitted between AE's, the correct sums are formed because the nulls do not intrude into the result. The same is true for a product reduction, or for a reduction using "maximum" or "minimum" as the operator, in which case a search is occurring.

SCALAR PROCESSING UNIT

The SPU is a conventional, register-oriented processor in most respects. It operates on an 80-nanosecond cycle. It has 16 48-bit, general-purpose registers, a full complement of numeric and nonnumeric operators and an instruction processor which includes content-addressable, memory-controlled, in-buffer, looping

capability. The unusual features of the SPU relate to subroutine entry and to instructions issued to the parallel processor control unit.

In addition to the 16 general purpose registers, there are 16 120-bit vector data buffers (VDB's). The 120-bit width is the maximum width of any parallel processor control unit instruction. If a vector instruction or descriptor has been completely assembled at compile time, then it will be fetched from control memory into a VDB. From there, a single clock will be sent into the VIVU input buffer. If some run-time vector instruction or vector descriptor processing is necessary, then as many as four of the general-purpose registers can be referenced to insert fields into the 120-bit wide VIVU input buffer. This data will overwrite the associated fields in the 120-bit word coming from the VDB set. This facility permits the formation of skeletal descriptors at compile time, with the expectation that missing fields will be filled in at run time. It also permits a compile-time-generated descriptor to be overwritten at run time, which can be used to optimize VDB usage and memory references.

The SPU uses a push-down stack for subroutine linkage. The data which goes on the stack is the return address and a register address translation table. The 48-bit and 120-bit registers are referred to via an address translation unit. This capability assists in parameter passing and in the minimization of register saves and restores during subroutine entry and exit.

A second stack is maintained for the SPU executive state. Control memory is divided into the executive area, user read/write area, and user read-only area. The user stack is at one end of the user read/write area.

BSP SOFTWARE

Wherever possible, the BSP takes advantage of existing B 7700/B 7800 software. Necessary additional items are a small operating system to run on the SPU, a BSP compiler and the associated linkage editor and intrinsic functions, and the diagnostic package.

Perhaps the most interesting aspect of the SPU operating system is its facility for staying out of the way. For example, I/O is done in user mode, and assuming I/O is overlapped by computations, the SPU spends less than a microsecond in managing a transfer. Overlay management and chaptered file management are the major operating system functions performed for a running program. The hardware assists in overlay management by allowing presence bits to be assigned to each phase of a program. Hence, the operating system is dropped into automatically, if and only if the program attempts to enter a routine in a phase which is not present.

The FORTRAN compiler has a number of interesting features. The most important is the vectorization pass over the program, which converts serial FORTRAN into vector constructs. The BSP vectorization pass is a more complete parallel analysis

of a program than has been previously inserted into product compilers. The usual approach has been to attempt to vectorize only loops which did not contain branches or cyclic dependencies. The BSP vectorizer issues vector instructions even for branching cases, as long as the branch paths depend on vectors which are known at branch time. The vectorizer also detects cyclic dependencies. If an analysis of the dependency shows it is equivalent to a linear recurrence, then appropriate vector instructions are issued.

These types of parallelism are the most frequent which can be detected using rigorously defined algorithms. There are some important cases, such as when a user has undertaken to manage indices, for which only ad hoc vectorization techniques are known. These will be vectorized as well, but clear-cut statements about the extent of vectorization are not possible.

The FORTRAN compiler also contains the facility to directly express vector and array constructs. Assignment statements like A = A + 1, where A is an array, are permitted. Index expressions are permitted on array identifiers. For example, if A is a 3-dimensional array, then A (10:50:2, 6, 1:10) would refer to a 21 by 10 array which is a subspace of A. Clearly, this reference need only generate a descriptor, which is pushed into the VIVU queue. No actual subset of A is generated as a temporary.

An interesting special feature of the compiler is a source code regenerator. This regenerator creates standard FORTRAN statements. Hence, a programmer can indulge in the use of array extensions, but retain full compatibility with standard FORTRAN.

Because of the way in which the B 7700/B 7800 is connected to the BSP, it is possible to use standard B 7700/B 7800 terminals or network connections to run BSP diagnostics. A field engineer can invoke a sequence of routines which will result in a printout of suspect IC locations. He can quickly replace the suspect IC's and bring the BSP back up again, if it is down. This idea is extended to the ability to use BSP development engineers on-line in the diagnostic process. Thus, the field engineer and the customer have excellent assurance of prompt system recovery.