BSP. pdf

The Burroughs Scientific Processor (BSP)

DAVID J. KUCK, MEMBER, IEEE, AND RICHARD A. STOKES

Abstract—**The Burroughs Scientific Processor (BSP), a high-performance computer system, performed the Department of Energy LLL loops at roughly the speed of the CRAY-1. The BSP combined parallelism and pipelining, performing memory-to-memory operations. Seventeen memory units and two crossbar switch data alignment networks provided conflict-free access to most indexed arrays. Fast linear recurrence algorithms provided good performance on constructs that some machines execute serially. A system manager computer ran the operating system and a vectorizing Fortran compiler. An MOS file memory system served as a high bandwidth secondary memory.**

Index Terms—**Conflict-free array access, high-speed computer, parallel computer, pipeline computer, scientific computing, vectorizing compiler.**

I. Introduction

FROM the beginning of the Burroughs Scientific Processor (BSP) design activity, we attempted to develop a system with high performance and reliability that is practical to manufacture, easy to use, and produces high quality numerical results. It is not possible to substantiate the success or failure in achieving these goals because the product was cancelled before user installations were realized. However, a full prototype system was operational for several months on customer benchmarks and demonstrated the practicality of the architecture. A number of points of technical merit have surfaced and are presented herein.

Because of the market place for which the BSP was intended, we chose Fortran as the main programming language for the machine. This choice leads to a need for array-oriented memory and processor schemes. It also leads to various control mechanisms that are required for Fortran program execution. To design a cost-effective and user-oriented system, more than programming languages must be considered; characteristics of the types of programs to be run on the machine must be carefully considered. We have throughout the design effort paid attention to the syntax of Fortran and also the details of "typical" scientific Fortran programs.

In the past 10 years several high performance systems have been built, including the pipelined CDC STAR (CYBER-205) [12], CRAY-1 [22], MU5 [24], and TI ASC [29], as well as Burroughs Illiac IV (BBKK68) and PEPE [9], the Goodyear Aerospace STARAN [21], and the ICL DAP [10], all parallel machines. While parallelism and pipelining are effective ways of improving system speed for a given technology (circuit and memory family, etc.), they both have shortcomings. Some pipelines perform better, the longer the vectors are that they have to process. The performance of other pipeline systems depends on vector lengths matching high-speed register set sizes. Parallel systems perform best on vectors whose length is a multiple of the number of processors available. Either type of system performs adequately if vector sizes are very large relative to the machine, but as these systems are used in wider application areas, short vector performance becomes more important. Other limitations of most pipeline processors have been that the arithmetic operations to be pipelined can reasonably be broken into only a limited number of segments and that overlapping of several instructions in one pipeline leads to unreasonable control problems.

Another important characteristic of high performance machines is the level of the language they execute. The CDC STAR and TI ASC, for example, have in their machine languages scalar and very high-level vector instructions, while ILLIAC IV, on the other hand, has a traditionally very low level machine language. A high-level machine language that is well matched to source programs can make compilation and control unit design easier and also helps ensure high system performance. A difficulty of array instructions can be that the setup time for instructions effectively stretches the pipeline length out intolerably.

System Overview

In the BSP we have combined parallelism and pipelining, and have provided array instructions that seem well matched to user programs. The machine has a five segment, memory-to-memory data pipeline, plus earlier instruction-setup pipeline segments. The data pipeline executes instructions that represent whole array assignment statements, recurrence system evaluations, etc., in contrast to most machines in which one or two arithmetic operations may be pipelined together. The BSP has 17 parallel memories, 16 parallel processors, and two data alignment networks. Since most instructions can be set up in the control pipeline preceding the data pipeline, instruction setup overhead should be insignificant in most cases. Thus, we have attempted to balance those architectural features that can provide good speedups with various overheads that can degrade or ruin system performance.

Another important factor in most supercomputers is I/O speed. If a very high speed processor is connected to standard disks, system performance may collapse because of I/O bound computations. The BSP has a high performance semiconductor

Manuscript received June 5, 1978; revised December 3, 1981. This work was supported in part by the National Science Foundation under Grant US NSF MCS77-27910 and the Burroughs Corporation, Paoli, PA.

D. J. Kuck is with the Department of Computer Science, University of Illinois, Urbana, IL 61801.

R. A. Stokes is with the Burroughs Corporation, Paoli, PA 19301.

file memory. This is used as a backup memory and to provide a smooth flow of jobs to and from the BSP.

Certain technological decisions were dictated by a Burroughs parallel development of a standard circuit and packaging design called Burroughs Current Mode Logic (BCML). A relatively long clock period was established in order to reduce the number of pipeline segments (for both data flow and instruction setup), avoid the complexities of high frequency clock distribution, and to facilitate manufacturing and testing the machine. The memory cycle time, the time to align 16 words between memories and processors (in either direction), and the time for many processor operations are all one 160 ns clock period; two clocks are required for floating-point addition and multiplication. In terms of these major events per clock, we attempted to lay out an array instruction set whose performance in the final system could be easily estimated during the design period.

A very important point in predicting system performance, and hence rationally choosing between design alternatives, is the determinacy of the system's behavior. We attempted to remove as much uncertainty as possible by several design choices. First, a parallel memory system was designed that provides conflict-free access to multidimensional arrays for most of the standard access patterns observed in programs. For cost reasons a parallel memory is required to achieve adequate bandwidth, and our design guarantees that for most instructions the effective bandwidth will be exactly at its maximum capacity. Since array elements are accessed in a different order from that in which they are processed, data alignment networks are needed along the path from memory to the processors and from the processors back to memory. These alignment networks also operate in a conflict-free way for most common operations. Finally, to guarantee that the memory-to-memory data pipeline is seldom broken, an array instruction set was designed. For example, a single BSP instruction can handle a whole assignment statement (with up to five right-hand side arguments) nested in one or two loops. The instruction can represent a number of 16 element slices of the operands, as long vector operations are automatically sliced and the slices overlapped in the memory-to-memory pipeline. Furthermore, as one vector assignment statement instruction ends, the next one can be overlapped with it in the pipeline. So for short vector operations there is usually no problem in keeping the pipeline full, since several different Fortran level instructions may be in operation at once.

Thus, for a wide class of instructions it was possible to predict the system performance (up to the clock speed) very early in the design process (1973). Furthermore, it has been possible during the later stages of design to make tradeoffs in these terms. Array instructions have also been very beneficial in allowing logic designers and compiler writers to communicate with each other about their own design efforts and to make tradeoffs in concrete performance terms.

A Fortran compiler has been implemented that vectorizes ANS Fortran programs, thus allowing old programs to be run without expensive reprogramming efforts. Vector extensions to Fortran are also provided to allow users to "improve" certain parts of old programs, if desired, or to write new programs in efficient ways. The vectorizer not only handles array operations, it also substantially speeds up linear recurrences—as found, for example, in processes that reduce vectors to scalars (e.g., inner product, polynomial evaluation, etc.)—and effectively handles many conditional branches within loops.

For some applications, numerical stability is a serious problem. The BSP does high quality (approximate R*) rounding of its 36-bit mantissas and also provides double precision hardware operations. Furthermore, interrupts are generated for standard floating-point faults. Error detection and correction are provided throughout the system and automatic instruction retry is provided to ease the burden on the user in some cases.

The BSP and its file memory form a high-speed computing system that may be viewed as standing inside a computational envelope. This envelope is serviced by a *system manager* that can be a Burroughs B6700, B6800, B7700, or B7800. This front-end general-purpose system provides the following:

- compilation of BSP programs,
- archival storage for the BSP,
- data communication and time-sharing services to a user community,
- other languages and computation facilities.

Thus, a typical user will interactively generate compiled program and data files on the system manager, pass them to the BSP for execution, and have results returned via the system manager with permanent files maintained on the system manager's disks. Most job scheduling and operating system activities for the BSP are carried out on the system manager, so the BSP is dedicated to high-speed execution of user application programs.

II. System Overview

Fig. 1 shows a block diagram of the BSP and the system manager. The BSP itself consists of three major parts: the control processor, the parallel processor, and the file memory. In this section some characteristics of these parts of the BSP will be presented. In subsequent sections we will give more details about how they operate.

A. Control Processor (CP)

The control processor is a high-speed element of the BSP that provides the supervisory interface to the system manager in addition to controlling the parallel processor and the file memory. The CP consists of a scalar processor unit, a parallel processor control unit, a control memory, and a control and maintenance unit.

The CP executes some serial or scalar portions of user programs utilizing an arithmetic element (similar to one of the 16 arithmetic elements in the parallel processor) that contains additional capabilities to perform integer arithmetic and indexing operations. The CP also performs task scheduling, file memory allocation, and I/O management under control of the BSP operating system.

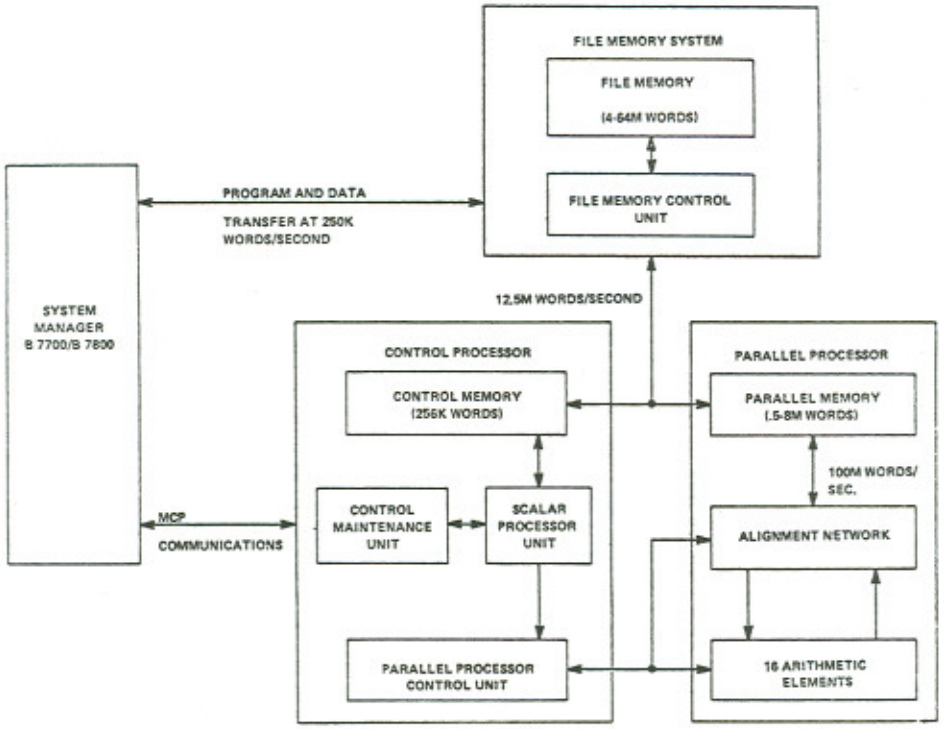

Fig. 1. BSP system diagram.

Scalar Processor Unit (SPU): The scalar processor unit processes all operating system and user program instructions that are stored in control memory. It has a clock frequency of 12.5 MHz and is able to perform up to 1.5 million floating-point operations/s. All array instructions and certain scalar operations are passed to the parallel processor control unit, which queues them for execution on the parallel processor.

Parallel Processor Control Unit (PPCU): The PPCU receives array instructions from the scalar processor unit. The instructions are validated and transformed into microsequences that control the operation of all 16 arithmetic elements in the parallel processor. Vectors of any length are handled automatically by the PPCU hardware, relieving the programmer and compiler of this burden.

Control Memory (CM): The control memory is used to store portions of the operating system and all user programs as they are being executed. It is also used to store data values that are operands for those instructions executed by the scalar processor unit. The control memory is a 4K bit/chip NMOS memory with a 160 ns cycle time. Capacity of the memory is 256K words; each word consists of 48 data bits and 8 bits for error detection and correction. Four words are accessed simultaneously, giving a minimum effective 40 ns access time per 48 bit word.

Control and Maintenance Unit (CMU): The control and maintenance unit serves as the direct interface between the system manager and the rest of the control processor for initialization, communication of supervisory commands, and maintenance. It communicates with the input/output processor of the system manager. The CMU has access to critical data paths and registers of the BSP, so that it can perform state analysis and circuit diagnostics under control of maintenance software running on the system manager.

B. Parallel Processor (PP)

The parallel processor performs array-oriented computations at high speeds by executing 16 identical operations simultaneously in its 16 arithmetic elements. Data for the array operations are stored in a parallel memory (PM) consisting of 17 memory modules. Parallel memory is accessed by the arithmetic elements through input and output alignment networks. A memory-to-memory data pipeline is formed by the five steps (fetch, align, process, align, store) and overlap in the pipeline provides significant performance benefits.

Parallel Memory (PM): The parallel memory is used only to hold data arrays for the parallel processor and consists of 17 memory units, each of which may contain from 32K to 512K words, making a total of from 0.5 to 8 million words. It is a 4K bit/chip NMOS memory with a 160 ns cycle time as in the control processor memory. Each word contains 48 data bits and 8 bits for error detection and correction. The maximum rate of data transfer between the PM and the arithmetic elements is 10^8 words/s. The organization of the PM permits simultaneous access to most commonly referenced components of an indexed array, such as rows, columns, or diagonals. For some operands the compiler must choose between allocating storage in PM or CM, and performance can suffer if this is not done properly.

The Alignment Networks (AN): The BSP has two alignment networks: the input alignment network for data fetching and the output alignment network for data stores. Both units contain full crossbar switching networks as well as hardware

for broadcasting data to several destinations and for resolving conflicts if several sources seek the same destination. This permits general-purpose interconnectivity between the arithmetic array and the memory storage modules. It is the combined function of the memory storage scheme and the alignment networks that supports the conflict-free capabilities of the parallel memory. The output alignment network is also used for interarithmetic element switching to support special functions such as the data compress and expand operations and the fast Fourier transform algorithm.

Arithmetic Elements (AE): At any time all of the arithmetic elements are executing the same instruction on different data values. The arithmetic elements operate at a clock frequency of 6.25 MHz and are able to complete the most common arithmetic operations in two clock periods. Each arithmetic element can perform a floating-point add, subtract, or multiply in 320 ns, so the BSP is capable of executing up to 50 million floating-point operations/s. Each arithmetic element can perform a floating-point divide in 1280 ns and extract a square root in 2080 ns.

C. File Memory (FM)

The file memory is a high-speed secondary storage device that is loaded by the system manager with BSP tasks and task files. These tasks are then queued for execution by the control processor. The FM is also used to store scratch files and output files produced during execution of a BSP program. It is the only peripheral device under the direct control of the BSP; all other peripheral devices are controlled by the system manager.

The FM utilizes high-speed semiconductor memory as its storage medium; it combines a 1 ms access time with a 12.5M word/s transfer rate. Since it is entirely semiconductor, the reliability of the file memory is much greater than that of conventional rotating storage devices.

III. Languages and Their Translation

The BSP can be regarded as a high performance Fortran machine, although many of the ideas in the design are useful for various languages. In this section we present some details of the vector form language seen by the PPCU and discuss how vector forms can be obtained from Fortran programs by a vectorizing compiler. The essential elements of many numerical algorithms are represented by these vector forms and numerical programs in most languages could be reduced to the same set of vector forms. However, the large collection of Fortran programs existing in the numerical computation community has dictated that the primary language of the BSP be Fortran. Nevertheless, vector extensions to Fortran are provided so that users may write new programs in a more convenient language than Fortran, and also to allow faster translation and possibly faster executable BSP code. We conclude this section with a sketch of the BSP vector Fortran extensions.

A. Vector Forms

The parallel processor control unit sequences the five stages of the data pipeline: fetch, align, process, align, store (FAPAS). In this pipeline several instructions corresponding to Fortran statements may be in execution at one time. To clarify this process, several definitions are required; these will lead to an understanding of the parallel processor control unit and compiler.

The BSP has a total of 64 *vector forms* that may be grouped in the following four types:

1) array expression statements,
2) recurrence and reduction statements,
3) expand, compress, random store, and fetch, and
4) parallel memory transmissions to and from control memory and file memory.

Array expression statements include indexing and evaluating right-hand side array expressions ranging from monad to pentad (five right-hand side operands), plus the assignment of the resulting values to parallel memory. A separate vector form exists for each possible parse of each right-hand side expression. The array operations are performed in an element by element fashion and allow scalars and array variables of one or two dimensions to be mixed on the right-hand side. For example,

```
      DO 5 I = 1, 30
        DO 5 J = 7, 25
5         X(I, J) = (A(I, J + 1) * 0.5 + B(I + 1, J))
                  * X(I, J + 1) + C(J)
```

would be compiled as a single vector form. This vector form can be regarded as a six-address instruction that contains the four array arithmetic operation specifications and the assignment operation.

Recurrence vector forms correspond to assignment statements with data dependence loops. For example,

```
      DO 3 I = 1, 25
3       Y(I) = F(I) * Y(I - 1) + G(I)
```

has a right-hand side that uses a result computed on the previous iteration. This recurrence produces an array of results, while others lead to a scalar result and are called *reductions*. For example, a polynomial evaluation by Horner's rule leads to the reduction

```
        P = C(O)
      DO 5 I = 1, 25
5       P = C(I) + Y * P.
```

Both of these are recurrences that can be represented by a linear system of the form $x = Ax + b$, where A is a lower triangular matrix with a single band, one diagonal below the main diagonal, and x is an unknown vector. We will refer to a linear recurrence of dimension n and order m as an $R\langle n, m\rangle$ recurrence, where n is the dimension of the matrix A and $m + 1$ is the bandwidth of matrix A. Thus, the above program leads to an $R\langle 25, 1\rangle$ system. Fast efficient algorithms exist for solving such systems and the $R\langle n, 1\rangle$ solver of [7] and [23] for small n and the $R\langle n, 1\rangle$ solver of [6] for large n have been built into the BSP. For wider bandwidth recurrences the column sweep

algorithm [15] is more efficient and it is used in the BSP. However, the user is not concerned with any of these considerations, since the vector forms described have array control unit hardware for their direct execution, as we shall see shortly.

Note that all recurrences would have to be executed serially without these algorithms. With them, $R\langle n, m\rangle$ systems ($1 < m \leq 16$ will obtain speedups proportional to m. For $m = 1$ and n of moderate size (say, 50 to 100), a speedup of 5 to 6 can be obtained, with greater speedups for large n. By speedup we mean time reduction compared to a hypothetical BSP with just one AE.

The third type of vector forms involves various sparse array operations. For example, in the case of a Fortran variable with subscripted subscripts, e.g., $A(B(I))$, no guarantee can be made concerning conflict-free access to the array A. In this case the indexing hardware generates a sequence of addresses that allows access to one operand per clock and these are then processed in parallel in the arithmetic elements. These are called *random store* and *random fetch* vector forms. Sparse arrays may be stored in memory in a compressed form and then expanded to their natural array positions using the input alignment network. After processing, the results may be compressed for storage by the output alignment network. These are called *compressed vector operand* and *compressed vector result* vector forms and they use control bit vectors that are packed, such that one 48 bit word is used for accesses to three 16 element vector slices.

A list illustrating the above three classes of vector forms is found in Table I. The mnemonics and comments should give an idea of what these vector forms do.

The fourth class of vector forms is used for I/O. Scalar and array assignments are made to control memory and parallel memory depending on whether they are to be processed in the scalar processor unit or the parallel processor, respectively; however, it is occasionally necessary to transmit data back and fourth between these memories. Transmissions to file memory are standard I/O types of operations.

These four types of vector forms comprise the entire set of array functions performed by the BSP. At the vector form level, the array processor may be regarded as arbitrarily large; thus, vector code generation is simplified in the compiler because it can transform Fortran programs into objects that map easily into vector forms, as we shall see shortly. However, in most Fortran programs some parameters are not defined at compile time (e.g., loop limits), so some run-time source language processing remains. This is carried out by the scalar processing unit of the control processor, and when it is finished the parallel processor is controlled by a template sequencing mechanism (see Section IV-D).

Vector forms are very high-level instructions with many parameters. For example, an array expression statement vector form corresponds to an assignment statement parse tree and leads to the execution of up to four operations on operands that may be combinations of scalars and one- or two-dimensional arrays. The following is a sketch of how the scalar processing unit (SPU) initiates the execution of a vector form in the parallel processor control unit (PPCU).

Consider a triad vector form

$RBV, Z = (A\ op_1\ B) op_2\ C, OBV$

where Z, A, B, and C are vector descriptors, op_1 and op_2 are operators, and RBV and OBV are optional result and bit-vector descriptors, respectively. Bit vectors may be used to specify the elements of an array to be operated on or stored; they are optional (but not shown) for a number of the entries in Table I. In executing the triad, the SPU issues the following sequence of instructions that describe the vector form to the PPCU:

*V*FORM TRIAD, op_1, op_2
OBV
RBV
*V*OPERAND *A*
*V*OPERAND *B*
*V*OPERAND *C*
*V*RESULT *Z*.

The *V*FORM instruction contains bits that name the first template (see Section IV-D) to be executed, specify actual operator names, indicate the presence of bit-vectors, and specify the program countercontents; it also contains other synchronization and condition bits. The *OBV* and *RBV* descriptors give the bit-vector starting addresses and lengths. The *V*OPERAND and *V*RESULT instructions give the start of the vector relative to an array location, the location of the array, the volume of the array, the skip distance between vector elements to be accessed, and optionally the skip distance between the start of subsequent vectors in a nested pair of loops. The *V*FORM instruction is preceded by a *V*LEN instruction that specifies the level of loop nesting and array dimensions. At this point, all source language parameters have been bound and run-time source language processing ends. The remaining processing done by the PPCU, e.g., array bounds checking, is the same for all operations. We shall return to a discussion of template sequencing in Section IV.

B. Fortran Vectorizer

In ordinary Fortran programs it is possible to detect many array operations that easily can be mapped into BSP vector forms. This is accomplished in the BSP compiler by a program called the Fortran vectorizer. We will not attempt a complete description of the vectorizer here, but we will sketch its organization, emphasizing a few key steps. For more discussion of these ideas, see [15], [17], and [4].

First, consider the generation of a program graph based on data dependences. Each assignment statement is represented by a graph node, and directed arcs are drawn between nodes to indicate that one node is to be executed before another. Algorithms for data dependence graph construction are well known [1], [4] and will not be discussed here. It should be observed that some compilers have used naive algorithms, for example, checking only variable names. The BSP algorithm does a detailed subscript analysis and thus builds a high quality graph with few redundant arcs, thereby leading to more array operations and fewer recurrences.

TABLE I

Vector Forms

MONAD	It accepts one vector set operand, does one monadic operation on it and produces one vector set result.	Z ← op A
DYAD	It accepts two vector set operands, does one operation on them and produces one vector set result.	Z ← A op B
VSDYAD	It is similar to the DYAD except that operand B is a scalar.	Z ← A op B
EXTENDED DYAD	It accepts two vector set operands, does one operation and produces two vector set results.	(Z1,Z2) ← A op B
DOUBLE PRECISION DYAD	It accepts four vector set operands (i.e., two double precision operands), performs one operation and produces two vector set results.	(Z1,Z2) ← (A1,A2) op (B1,B2)
DUAL-DYAD	It accepts four vector set operands, does two operations and produces two vector set results.	Z ← A op_1 B Y ← C op_2 D
TRIAD	It accepts three vector set operands, does two operations and produces one vector set result.	Z ← (A op_1 B) op_2 C
TETRAD1	It accepts four set operands, does three operations and produces one vector set result.	Z ← ((A op_1 B)op_2 C) op_3 D
TETRAD2	It is similar to the TETRAD1 except for the order of operations.	Z ← (A op_1 B)op_2 (C op_3 D)
PENTAD1	It accepts five vector set operands, does four operations and produces one vector set result.	Z ← (((A op_1 B)op_2 C) op_3 D)op_4 E
PENTAD2	It is similar to the PENTAD1 except for the order of operations.	Z ← ((A op_1 B)op_2 (C op_3 D))op_4 E
PENTAD3	It is similar to the PENTAD1 except for the order of operations.	Z ← ((A op_1 B)op_2 C) op_3 (D op_4 E)
AMTM	It is similar to the MONAD and is used to transmit from parallel memory to control memory.	Z ← op A
TMAM	It accepts 6 vector set operands from control memory to transmit to parallel memory.	Z ← A1(0,0), A2(0,0) A3(0,0), A4(0,0), A5(0,0), A6(0,0)
COMPRESS	It accepts a vector set operand, compresses it under a bit vector operand control and produces a vector set result.	X ← A, BVO
EXPAND	It accepts a vector operand, expands it under a bit vector control and produces a vector set result.	X ← V, BVO
MERGE	It is the same as the EXPAND except that the vector set result elements corresponding to a zero bit in BV are not changed in the parallel memory.	X ← V, BVO
RANDOM FETCH	It performs the following operation Z(j,k) ← U(I(j,k)), where U is a vector and I is an index vector set.	
RANDOM STORE	It performs the following operation X(I(j,k)) ← A(j,k), where X is a vector and I is an index vector set.	
REDUCTION	It accepts one vector set operand and produces one vector result given by X(i) ← A(i,0) op A(i,1) op A(i,2) op A(i,3) ... A(i,L), where op must be a commutative and associative operator.	

TABLE I (CONTINUED)

DOUBLE-PRECISION REDUCTION	It accepts two vector set operands (one double-precision vector set) and produces two vector results (one d.p. vector) given by $(X_1(i),X_2(i)) \leftarrow (A_1(i,0),A_2(i,0))$ op $(A_1(i,1), A_2(i,1))$ op ... $(A_1(i,L), A_2(i,L))$, where op must be a commutative and associative operator.
GENERALIZED DOT PRODUCT	It accepts two vector set operands and produces one vector result given by $X(i) \leftarrow \{A(i,0)\ op_2\ B(i,0)\}\ op_1\ \{A(i,1)\ op_2\ B(i,1)\}\ op_1 \ldots \{A(i,L)\ op_2\ B(i,L)\}$, where op_1 must be a commutative and associative operator.
RECURRENCE-1L	It accepts two vector set operands and produces one vector result given by $X(i) \leftarrow (\{\ldots\{(B(i,0)\ op_1\ A(i,1))\ op_2\ B(i,1)\}\ op_1 \ldots\}\ op_1\ A(1,L))\ op_2\ B(i,L)$ where op_2 can be ADD or IOR and op_1 can be MULT or AND.
PARTIAL REDUCTION	It accepts one vector set operand and produces one vector set result given by $Z(i,j) \leftarrow Z(i,j-1)$ op $A(i,j)$, where op must be a commutative and associative operator.
RECURRENCE-1A	It accepts two vector set operands and produces one vector set result given by $Z(i,j) \leftarrow \{Z(i,j-1)\ op_1\ A(i,j)\}\ op_2\ B(i,j)$, where op_1 can be MULT or AND and op_2 can be ADD or IOR.

As an example, consider the following program:

```
  DO 5 I = 1, 25
1     A(I) = 3 * B(I)
          DO 3 J = 1, 35
3             X(I, J) = A(I) * X(I, J - 1) + C(J)
5     B(I) = 2 * B(I + 1).
```

A dependence graph for this program is shown in Fig. 2, where nodes are numbered according to the statement label numbers of the program. Node 1 has an arc to node 3 because of the $A(I)$ dependence and node 3 has a self-loop because $X(I, J-1)$ is used one J iteration after it is generated. The crossed arc from node 1 to node 5 is an antidependence arc [16] indicating that statement 1 must be executed before statement 5 to ensure that $B(I)$ on the right-hand side of statement 1 is an initial value and not one computed by statement 5. Arcs from above denote initial values being supplied to each of the three statements: array B to statements 1 and 5, and array C to statement 3. The square brackets denote the scope of loop control for each of the DO statements.

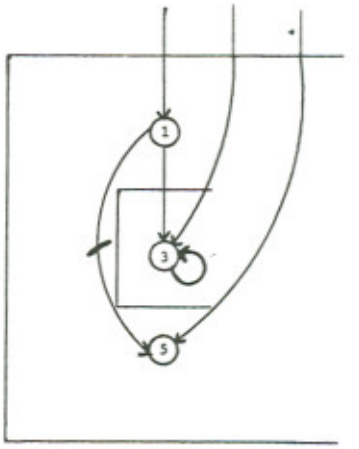

Fig. 2. Data dependence graph.

Given a data dependence graph, loop control can be distributed down to individual assignment statements or collections of statements with internal loops of data dependences. In our example there is one loop (containing just one statement) and two individual assignment statements. After the distribution of loop control, the graph of Fig. 2 may be redrawn as shown in Fig. 3.

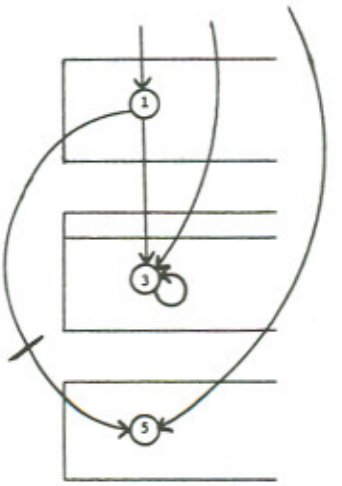

Fig. 3. Graph with distributed loop control.

The graph of Fig. 3 can easily be mapped into BSP vector forms. Statements 1 and 5 go into array expression statement vector forms directly since they are both dyads. Had they had more than five right-hand side variables, their parse trees would have been broken into two or more array expression statement vector forms and joined by a temporary array assignment. Statement 3 can be split into 25 independent re-

currence systems, each with a bandwidth of 1 caused by the $J-1$ to J dependence, resulting in 25 independent $R\langle 35, 1\rangle$ systems, each of which maps into a recurrence vector form. Alternatively, it can be computed as a series of 35 array expression statement vector forms (triads). The maximum speed choice is made based on the loop limits.

The above cases are rather simple constructs, but they are typical of those found in existing Fortran programs. Several kinds of more complex cases are possible. If arrays with subscripted subscripts occur, they are compiled using the random fetch or store vector forms mentioned earlier. If recurrences with nonlinear right-hand sides occur, e.g., $X(I) = A(I) * X(I-1) * X(I-2) + B(I)$, then serial code is compiled. To this point we have ignored conditional statements inside loops, another problem that in the past has caused serial code to be compiled.

A number of IF statements can in fact be handled in parallel in the BSP. For a theoretical discussion of various types of IF's, see [14], where examples and measurements of the frequency of various types of IF's are presented. It turns out that many of the commonly found IF's can be handled in the BSP by using standard vector forms which include bit vectors. In this way, certain IF statements can be combined with assignment statements in a single vector form. For example, consider the following program:

```
      DO 1 I = 1, 92, 2
         DO J = 1, 46
1           IF(A(I, J).LT.0) B(I, J) = A(I, J) * 3.5.
```

This loop can be mapped into a single array expression statement vector form with bit-vector control that performs the parallel tests and makes the appropriate assignments to $B(I, J)$. By using loop distribution, many of the IF's found in ordinary Fortran programs can be transformed into such vector operations that allow substantial speedups on the BSP. Of course, there is also a residual set of IF's that must be compiled as serial code.

A traditional objection to array computers was that too many of the statements found in ordinary programs could not be vectorized and would have to be executed in a traditional sequential manner. In the BSP a combination of software and hardware innovations has led to a system that avoids most of these traditional objections. Of primary software importance are the distribution of loop control, fast algorithms to solve linear recurrences, and the vectorization of IF statements. Also of key importance is a good test for data dependence between subscripted variables, the appropriate introduction of additional subscripts to variables of lower dimension than their depth of loop nesting, and the transformation of scalar expressions inside loops as well as their substitution into subscripts. After a source program has been mapped into vector forms it is ready for execution.

C. Fortran Language Extensions

To provide users with language conveniences for writing new programs and to allow rewriting old programs that are difficult to vectorize automatically, several language extensions are being provided in the BSP software. Some of these extensions are also part of the proposed new ANS Fortran. The extensions may be categorized in four cases as array description, array operation, control, and I/O statements; we shall deal with them in that order, providing only a quick sketch of the ideas.

Arrays may be declared with the colon notation, using positive and negative subscripts, e.g., REAL $A(0{:}3, 0{:}3, -3{:}3)$ declares a $4 \times 4 \times 7$ array. Portions of declared arrays may be renamed for easy reference using an ARRAY statement as the following example shows:

```
REAL A(100, 100)
ARRAY ROW 2(J = 1:100) = A(2, J),
      DIAG(I = 1:100) = A(I, I)
```

identifies ROW 2 as a vector consisting of the second row of A and DIAG as the main diagonal of A. No storage is allocated or data are moved by an ARRAY statement, only additional array descriptors are created.

Array operations can be specified in various ways. If A and B are declared arrays, then $A = 0$ sets all elements of A to zero and $B = B + 1$ adds 1 to all elements of B. If A is two-dimensional, then $A(*, 0) = 2$ sets column zero of A to 2. Arithmetic, relational, and logical operators may be applied to pairs of arrays that are congruent, in which case element by element operations are performed. Furthermore, bit-vectors may be used to control array operations by use of the WHERE statement as follows. Assume that A and B are 500 element vectors, then

```
WHERE(A.GE.0) B = B + A
```

is equivalent to

```
      DO 10 I = 1, 500
10    IF (A(I).GE.0) B(I) = B(I) + A(I).
```

This is generalized to a block-structured WHERE DO that contains a sequence of OTHERWISE statements and ends with an END WHERE. PACK and UNPACK statements are provided to allow sparse arrays to be compressed and expanded, respectively; multidimensional arrays may be packed into vectors based on logical tests. There is also an IF-THEN-ELSE construct and an END DO that does not require a label in its DO statement. For scalars or congruent arrays, the exchange statement

```
A = = B
```

exchanges A and B in memory.

A collection of intrinsic functions is also provided, one set generalizes the standard scalar intrinsics to arrays (e.g., transcendental functions of the elements of an array) and the other set provides some standard array operations (e.g., dot product, matrix product, max of an array, etc.).

User programs can control I/O without supervisor intervention and without buffering by using the DIRECT statement which names an array that appears in a following READ statement. Execution of statements after the READ continues simultaneously with the input until the DIRECT named variable is encountered on the right-hand side of an assignment statement, at which point execution is suspended until the READ is completed.

IV. System Operation

In this section we give more details of the overall system operation. First, the parallel memory and its conflict-free structure are discussed (Section IV-A), and this is followed (Section IV-B) by the alignment networks that stand between the parallel memory and the arithmetic elements, which are discussed in Section IV-C. Section IV-D ties these components of the parallel processor together by detailing the template sequencing as carried out by the parallel processor control unit.

This also relates back to Section III-A, where it was pointed out that source programs are first mapped by software into vector forms which in turn are mapped by hardware into a sequence of templates. The template sequencing mechanism is one of the key points in achieving high system utilization through pipelining and overlap of vector forms in the BSP. This section concludes with a discussion of the high performance file memory used for secondary storage in the BSP.

A. Parallel Memory

The BSP parallel memory consists of 17 memory modules, each with a 160 ns cycle time; and since we access 16 words per cycle, this provides a maximum effective 10 ns memory cycle time. This is well balanced with the arithmetic elements which perform floating point addition and multiplication at the rate of (320 ns/16 operations) = 20(ns/operation), since each operation requires 2 arguments and temporary registers are provided in the arithmetic elements. Note that only array accessing (including I/O) uses parallel memory, since programs and scalars are held in control memory. Thus, perfect balance between parallel memory and floating-point arithmetic may be achieved for triad vector forms since three arguments and one result (four memory accesses) are required for two arithmetic operations. For longer vector forms, since temporaries reside in registers, only one operand is required per operation, so there is substantial parallel memory bandwidth remaining for I/O.

The total memory size ranges from 0.5 to 8 million 48 bit words. Eight parity bits provide single error correction and double error detection.

The main innovation in the parallel memory of the BSP is its 17 modules. In past supercomputers it has been common to use a number of parallel memory modules, but such memory systems are vulnerable to serious bandwidth degradation due to conflicts. For example, if 16 memories were used and a 16 × 16 array were stored with rows across the units and one column in each memory unit, then column access would be sequential.

Various storage schemes have been invented to avoid such memory conflicts. For example, arrays may be skewed [16] so that rows and columns can be accessed without conflict, and other related skewing schemes may be found with other useful properties. However, it is easy to show [5] that in general it is impossible to access rows, columns, and diagonals of square arrays without conflict if any power of two number of memory modules is used. Of course, various ad hoc procedures may be contrived, e.g., different skewing schemes for different arrays, but in the long run these are a compiler writer's (or user's) nightmare. A uniform, conflict-free procedure carried out by the hardware would be far superior for users of the system.

Early in the design of the BSP we decided to build the best possible parallel memory in this respect, and thereby avoid as many software implementation and performance problems as possible. For this reason we settled on a 17 memory module system that would provide conflict-free array access to most common array partitions, and yet have little redundant memory bandwidth since only one memory unit is unused per cycle.

With 17 memory modules it is clear that conflict-free access to one-dimensional arrays is possible for any arithmetic sequence index pattern except every 17th element. For two-dimensional arrays with a skewing distance of 4, conflict-free access is possible for rows, columns, diagonals, back-diagonals, and other common partitions, including arithmetic sequence indexing of these partitions [5]. The method extends to higher numbers of dimensions in a straightforward way.

One mundane characteristic of some Fortran programs can cause a problem here, namely, the use of COMMON in subroutine parameter passing. If used in the most general ways, this forces the storage of arrays in a contiguous way across parallel memory. In this case conflict-free access can still be guaranteed to any arithmetic sequence of physical memory addresses, as long as the difference between addresses is not a multiple of 17. This may force some array dimensions to be adjusted slightly for conflict-free access to all of the desired patterns.

To access parallel memory a set of 16 memory addresses must be generated. Assume that addresses are to a linear address space, i.e., multidimensional arrays have been mapped into a one-dimensional array, and that the array is stored across the memory modules beginning with module 0, through module 16, continuing in module 0, and so on. Then to access address α we must generate a *module number* μ and an *index* i in that module. These are defined by[1]

$$\mu = \alpha(\text{mod } 17)$$

$$i = \left\lfloor \frac{\alpha}{16} \right\rfloor$$

since there are 17 memory modules and we access 16 numbers (one for each AE) per memory cycle. Notice that this wastes $\frac{1}{17}$ of the address space, a minor penalty for the conflict-free access it provides. Address generation hardware for the memory system is somewhat complex, but can be done in parallel for a sequence of addresses in one clock using the scheme described in [18]. This hardware also generates indices to set the aligment networks appropriately for each memory access.

B. Alignment Networks

The separation of data alignment functions from processing and memory activities is another departure of the BSP from most previous computers. As discussed earlier, the BSP has an input alignment network (IAN) connecting parallel memory to the arithmetic elements, and an output alignment network (OAN) connecting the arithmetic elements either to themselves or to parallel memory. Alignment of the elements of two arrays is sometimes required by the parallel memory and sometimes required by program or algorithm constraints, as the following examples illustrate.

Suppose we want to add together two rows of a matrix, element by element. The origins of the rows will in general be stored in different memory modules, so one row must be shifted relative to the other to align them for addition. Now if we want to add the odd elements of one row to the elements of another row (half as long), the first row will have to be "squeezed" as well as shifted to align proper pairs of operands. Similar alignment problems arise in row-column, column-diagonal, etc., pairings. Since we must store arrays consistently in parallel memory, the output alignment network is used to satisfy

[1] We use $\lfloor x \rfloor$ to denote the integer part of x.

the storage requirements and indexing patterns of the variable on the left-hand side of each assignment statement.

The above uses of the alignment networks hold for recurrence and reduction vector forms. In these cases, data may be aligned after fetching via the IAN, but now the OAN is useful between processing steps. As a simple example, consider the summation of 32 numbers. This may be done in five steps, each step consisting of an addition followed by an output alignment network mapping of the AE's into the AE's in the form of a tree, which reduces the 32 numbers to one in $\log_2 32$ steps. Similarly, other reduction operations may be carried out using the OAN; these include such operations as finding the maximum or minimum of a set of numbers. Notice that for any such operations the reduction to a set of 16 numbers is carried out using all 16 processors, and after that the number of processors used is halved on each step.

In solving more general linear recurrences, a vector of results is produced, e.g., an $R\langle n, 1\rangle$ system leads to n results. Again, the OAN is used for an AE to AE mapping of intermediate results. Other important algorithms also require data alignment between operations; the FFT [20] and the Batcher merge and sort algorithms [2] are examples. These and other algorithms can be implemented directly in the BSP using microprogrammed AN sequencing patterns. Each alignment operation takes one clock in a FAPAS pipeline sequence. In array expression statement vector forms most are overlapped, while in recurrence vector forms the later alignments must be alternated with arithmetic.

In the course of some of the above algorithms, certain vector positions are vacated during the course of the computation, e.g., reduction operations reduce a vector to a scalar. An effective way of handling this is to have the IAN introduce *null elements* into the computation at appropriate points. In AE operations null elements are handled as follows:

$$\begin{aligned} \text{operand} &\leftarrow \text{null } \theta \text{ operand} \\ \text{operand} &\leftarrow \text{operand } \theta \text{ null} \\ \text{null} &\leftarrow \text{null } \theta \text{ null} \end{aligned}$$

for any operator θ. Memory modules block the storing of a null. Nulls are also used when a vector length is not equal to a multiple of 16, so the last slice of the vector is padded out with nulls by the IAN.

The alignment networks are constructed from multiplexers and are generalizations of crossbar switches. In addition to the permutation functions of crossbars, the alignment networks can also broadcast an input element to any selected set of destinations. Furthermore, the random store and fetch vector forms, as well as the compress and expand operations, use the alignment networks to carry out their mappings. Control of the AN's is closely related to PM control. As was pointed out in Section IV-A, memory addressing information and AN control indices are generated by the same hardware control unit.

The two alignment networks have similar control in that they are both source initiated; conceptually, for the IAN the memories specify which AE to transmit to and for the OAN, the AE's specify which memory they want to store to. In certain cases, e.g., random store and fetch, the AE's actually generate memory addresses as suggested here, whereas in the standard array expression statement or recurrence vector forms all of the addressing is carried out by a special control unit hardware. Conflict resolution hardware is provided to sequence certain alignments in several steps. For more AN details, see [18].

C. Arithmetic Element

The 16 arithmetic elements are microprogrammed, being sequenced by the parallel processor control unit using a wide (128 bit) microcode word. Besides the arithmetic operations expected in a scientific processor, the BSP has a rich set of nonnumeric operations that include field manipulation, editing, and Fortran format conversion operators. Floating-point addition, subtraction, and multiplication require two 160 ns clocks each, floating-point division requires 1280 ns, and the square root operation requires 2080 ns; the latter two use Newton–Raphson iterations that start with ROM values selected in each AE [11].

Single precision floating-point arithmetic is carried out using normalized signed magnitude numbers with 36 bits of mantissa and 10 bits of exponent, providing about 11 decimal digits of precision with a range between approximately 5.56×10^{-309} and 8.99×10^{307}. Four guard digits are retained within the AE and R* rounding is carried out using these four bits. Given that most alignment shifts are small [25], this should provide a good approximation of full R* rounding. Double-precision operations are carried out in the hardware (a double length product is always generated) about four times slower than single precision. The range of double precision numbers is the same as for single precision, and the precision is twice as great. Characters are stored as 8 bit EBCDIC bytes, packed six to a word.

Although they are not seen by users, each AE has a file of 10 registers, in addition to the standard registers used in the course of various operations. These are very convenient for holding intermediate results in the course of evaluating vector forms. The register assignment is done within the vector forms and the number of registers necessary to ensure high system performance was thus decided when the machine was designed. This is in contrast to building a machine first and then studying register allocation as a later compiler design question.

To add to the system reliability each AE contains residue checking hardware to check the arithmetic operations. Two-bit, modulo 3, residue calculations are carried out for each exponent and mantissa. To enhance the diagnosability of the entire system, the AE's (actually the AE-alignment network interfaces) contain Hamming code generators, detectors, and correctors for the data path loop from the AE's through the alignment networks and memory, and back to the AE's. This allows control information to be included in the Hamming code in order to check for failures in the control hardware of the parallel memory and alignment networks as well as the data paths mentioned above. The eight parity bits allow single error correction and double error detection. Most double-bit parity failures and all residue check failures lead to an instruction retry (see Section IV-D).

D. Template Sequencing

The execution of a vector form must be broken into a sequence of elemental array sequences called templates. This is done by the parallel processor control unit which issues a sequence of (one or more) *templates* to execute each vector form. The template provides the control framework in which vector forms are executed. Because it is desirable to overlap the execution of these template in the FAPAS pipeline, appropriate matching must be found in one template to accommodate the next template. For example, the fetching of operands for the next template may occur before the storing of the results from the previous template. In terms of such matching, *template families* have been defined with respect to the interface characteristics of their front and back templates. Each is said to have a front and back family number.

During execution the PPCU chooses templates on the basis of the vector form and the family numbers. For example, in executing a dyad followed by a triad, the PPCU will match the front family of the first triad template with the back family of the last dyad template. Then (assuming that there are more than 16 triad operations to perform) a second triad template will be chosen by matching its front family with the back family of the first triad template. Within three templates this reaches a cycle of one or two templates that repeats until the end of this particular vector form execution.

It is important to realize that the FAPAS overlap between templates as well as the arithmetic element register allocation for each one is done at machine design time. This leads to an *a priori* understanding of the machine's performance over a wide range of computations and also allows more vector operation overlap than might be possible otherwise, since all possible template combinations have been considered at machine design time.

This is in contrast to most previous high performance machines, wherein such overlap attempts are made at compile time or execution time. For example, in the CDC 6600 [27] and its successors, the control unit SCOREBOARD attempts run-time overlap and this is aided by compile-time transformations [26]. In the IBM 360/91 an overlap mechanism was built into the processor [28] in an effort to sequence several functional units at once. Most previous high performance machines seem to have overlap mechanisms that combine compile-time and run-time (control unit and processor) overlap decisions in ways similar to these. In some cases, however, certain common functions can be chained together (e.g., multiply and add for inner product in the CRAY-1), although in the CRAY-1, for example, intricate run-time considerations dictate whether or not chaining is possible [8].

The five-stage FAPAS pipeline may have four templates in process at once, and five more templates may be in various stages of setup in the PPCU pipeline that precedes execution. The amount of such overlap that exists at any moment depends, of course, on the width of individual templates. However, it is clear from an analysis of the templates that for most of the common Fortran constructs a very high percentage of processor and memory utilization can be expected in the BSP. Notice that there is overlap between templates of one vector form as well as overlap between different vector forms.

Due to the fact that the BSP executes a very high-level array language, complex hazard checking can be performed. As was mentioned in the discussion of template families, one template's fetches may begin before a previous template's store has been executed. In some cases, two previous stores may be pending while a third template's fetch is executed. This leads to *data dependence hazards* that must be checked before executing such memory accesses, i.e., if a fetch is for data that are to be stored by a previous template, the fetch must be delayed until after the store. Such hazard checking is carried out for up to two stores before a fetch by a combination of software and hardware in the BSP.

To clarify the above ideas, consider the following example program:

```
      DO 1 I = 1, 40
1     Y(I) = A(I) + B(I)
      DO 2 I = 1, 90
2     Z(I) = (C(I) + D(I)) * E(I).
```

This would be compiled as a dyad vector form followed by a triad; the dyad would be executed using $3\left(=\left\lceil\frac{40}{16}\right\rceil\right)$ templates and the triad using $6\left(=\left\lceil\frac{90}{16}\right\rceil\right)$ templates. Fig. 4 shows a FAPAS pipeline timing diagram for the execution of these templates.

In Fig. 4 notice that three distinct dyad templates are used:[2] dyad (2, 3), dyad (3, 6), and dyad (6, 7). The last memory cycle on clock 3 is provided for a previous template—some waste can be expected when beginning or ending a computation—but otherwise all memory cycles are occupied until clock 8. Notice also that after processing begins, the first three templates use $\frac{2}{3}$ of the processor clocks until clock 12.

The triad processing begins with three distinct templates: triad (2, 4), triad (4, 8), and triad (8, 8). It then continues in a steady state with triad (8, 8) templates. Except for one last clock at the dyad interface, the triad templates operate with total utilization of both the parallel memory and the parallel processor. Generally speaking, longer templates achieve very high utilization of the system hardware.

The total number of clocks required to execute this sequence of templates is 39, and since the original program contains 220 floating-point operations, the effective speed of this computation is greater than 35 million floating-point operations/s (Mflops). Assuming processing overlap at the beginning and end of this sequence, only 36 clocks are required for processing and on this basis a rate of more than 38 Mflops is achieved.

The use of vector forms and templates allows the easy implementation of a number of desirable features. The basic idea is that templates can be regarded as global microinstructions (the PPCU is microprogrammed) or "macroinstructions" that sequence the entire parallel processor. Consequently, even with

[2] The parenthesized numbers are the front family and back family identification numbers. The back family number of one must match the front family number of the next.

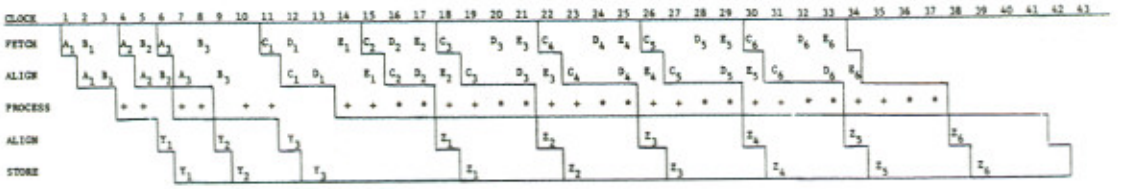

Fig. 4. Triad template sequencing.

a set of overlapped templates in progress, it is easy to delay the entire parallel processor. This is easiest to think of in terms of Fig. 4. For example, if an error occurs, a vertical slice is made through Fig. 4 at the end of some clock period and all further steps may be deferred until the interrupt is handled. Also, longer operations than addition and multiplication (i.e., division or square root) can simply be handled by extending the process step for sufficiently many clocks and deferring all other tracks in the FAPAS diagram, resuming them when the longer operation is complete; double precision is implemented in a similar manner. Furthermore, instruction retry is facilitated by this ability to break the FAPAS sequence at any step and back up to the beginning of a vector form, since memory stores are prevented in case of an error.

The distinction between data pipelining and overlap here and in other machines should be clear at this point. The BSP pipelines each template through the five FAPAS segments, in contrast to traditional pipelining of arithmetic operations; the individual operations being decomposed are array assignment statements, recurrences, etc., in the BSP. A sequence of templates can be likened to a sequence of additions in an arithmetic pipeline. On the other hand, most modern machines are overlapped in the sense that memory and arithmetic can be performed simultaneously, based on run-time lookahead. In the BSP the analogous process is overlap between vector forms, as was illustrated in Fig. 4.

Run-time lookahead schemes are limited by data hazards, jumps, and various other resource conflicts. By complex analysis of source programs, much more simultaneity is possible. For the BSP, many IF and GOTO statements are replaced by bit-vector modifications of vector forms, so even these difficulties of standard run-time lookahead schemes are often avoided.

Globally, the PPCU may be viewed as accepting a single stream of array instructions, namely, vector forms, and producing five streams of array instructions to sequence the FAPAS segments. Thus, in the terminology of [16], the BSP array processor is a SIAMEA (single instruction array/multiple execution array) machine. Of course, the scalar processor unit also may execute instructions from a given Fortran program simultaneously with the parallel processor.

E. File Memory

The BSP file memory consists of two sections, the file storage units and the file memory controller. This is the secondary memory of the BSP, with longer term storage being provided on conventional disks and other I/O devices attached to the system manager (see Fig. 1).

During the BSP design phase several semiconductor technologies showed promise as high-speed secondary memory devices, in competition with conventional head-per-track disks. Charge-coupled device (CCD) and random access memory (RAM) chips were considered because they provide very attractive characteristics, they were well along in development, and their price outlook seemed likely to be competitive with head-per-track disks. As a result, an MOS RAM was chosen to provide the BSP file memory with a very low latency and a high transfer rate, which is expected to perform with good reliability and maintainability at a reasonable cost.

The file storage unit is built with semiconductor high density memory devices that provide a transfer rate of two words every 160 ns with a maximum latency of less than 1 ms. Nonaddressed modules are operated at $\frac{1}{4}$ that clock rate to conserve power, yet provide the refresh needed for the volatile devices. Each file storage unit is organized in 4 M ($M = 2^{20}$) word sections and may contain 4 sections for a total of 16 M words of 48 data bits plus 8 parity bits. Two words are accessed in parallel for a transfer rate of 112 bits/160 ns $= \dfrac{2 \text{ words}}{160 \text{ ns}} = 12.5 \times 10^6$ word/s. A file memory system may contain up to 4 file memory units for maximum file memory capacity of 64 M (64×10^6) words. Thus, the file memory provides a backup storage of one to two orders of magnitude the size of parallel memory.

Consider the file memory in relation to a typical BSP computation. For example, a block of 32K words may be read at an overall effective rate of about 10^7 words/s. Since the BSP operates at a maximum of 50 Mflops, this means that only 5 floating-point operations need be performed per I/O word to balance the I/O and processing rates. To sustain this, output must be considered as well so the ratio becomes 10 floating-point operations per I/O word, for balance. These ratios seem well within the range of most large scientific calculations (whose ratios often range up to 100 or more).

The file memory stands between the BSP and the system manager. The file memory controller can transmit data to and from the BSP parallel memory or control memory as well as the system manager I/O processor (see Fig. 1), the latter at the 0.25×10^6 words/s maximum channel speed of the system manager. I/O instructions are passed between the BSP control processor and the file memory controller.

The file memory controller (FMC) contains buffer areas to match the file memory speed with that of the system manager. I/O requests from the BSP and system manager are queued in the FMC in a 32 entry queue. Normally, the slower

system manager receives highest priority and the BSP requests are handled in a first-in, first-out manner.

The file memory can be addressed to the word level and a block to be transmitted can be any number of words. The FMC converts logical addresses into physical addresses. A logical address contains a file name, a starting address in the file, a block length, and a destination (or source) memory address. File protection is provided by FMC hardware that allows any combination of four access modes: system manager READ or WRITE and BSP user program READ or WRITE. When operating in problem state, BSP I/O instructions can be executed with no supervisor program intervention at all for error-free transmissions. Synchronization bit registers are provided that allow BSP user programs to test for I/O completion without supervisor program intervention. The FMC also contains hardware for I/O instruction retry for all errors not automatically corrected. Thus, a number of situations that might traditionally have required slow, operation system intervention are handled by FMC hardware in the BSP.

Task switching within the BSP takes less than one-half second, but the speed of flow of jobs from the system manager, through file memory, and into the parallel memory and control memory, depends on many details of individual jobs.

V. Conclusion

We have outlined the organization of the BSP, a high performance scientific computer. Its key features include high quality, fast arithmetic, conflict-free access to arrays in parallel memory, separate data alignment networks, a pipelined control unit that sequences a high-level data pipeline and can overlap the execution of its vector form language, and a semiconductor file memory for secondary storage. Many error-checking and correcting features are included throughout the system to enhance its reliability and maintainability. Software was provided on a system manager computer that handles most operating system functions and has a Fortran vectorizing compiler that can also handle vector extensions.

The BSP system design effort began in early 1973, and led to an operational prototype machine in 1978. The maximum system speed is 50 million floating-point operations/s. The system performed as a "class 6" computer by running the Department of Energy LLL loops at speeds in excess of 20 Mflops. In fact, its average speed is almost identical to the average speed of the CRAY-1 on these benchmarks. A major design goal was a system that could achieve a high sustained performance over a wide variety of scientific and engineering calculations using standard Fortran programs. It is estimated that 20–40 Mflops could be achieved for a broad range of Fortran computations.

Acknowledgment

The authors are grateful to the entire BSP design team for years of work in the development of this system.

References

[1] U. Banerjee, "Data dependence in ordinary programs," M.S. thesis, Dep. Comput. Sci., Univ. of Illinois, Urbana-Champaign, Rep. 76-837, Nov. 1976.

[2] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Comput. Conf.*, vol. 32, 1968, pp. 307–315.

[3] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV computer," *IEEE Trans. Comput.*, vol. C-17, pp. 746–757, Aug. 1968.

[4] U. Banerjee, S. C. Chen, D. J. Kuck, and R. A. Towle, "Time and parallel processor bounds for Fortran-like loops," *IEEE Trans. Comput.*, vol. C-28, pp. 660–670, Sept. 1979.

[5] P. Budnik and D. J. Kuck, "The organization and use of parallel memories," *IEEE Trans. Comput.*, vol. C-20, pp. 1566–1569, Dec. 1971.

[6] S. C. Chen, D. J. Kuck, and A. H. Sameh, "Practical parallel band triangular system solvers," *ACM Trans. Math. Software*, vol. 4, pp. 270–277, Sept. 1978.

[7] S. C. Chen and D. J. Kuck, "Time and parallel processor bounds for linear recurrence systems," *IEEE Trans. Comput.*, vol. C-24, pp. 701–717, July 1975.

[8] "The CRAY-1 computer," *Preliminary Reference Manual*, Cray Res. Inc., Chippewa Falls, WI, 1975.

[9] P. H. Enslow, Ed., *Multiprocessors and Parallel Processing.* New York: Wiley-Interscience, 1974.

[10] P. M. Flanders, D. J. Hunt, S. F. Reddaway, and D. Parkinson, "Efficient high speed computing with the distributed array processor," in *High Speed Computer and Algorithm Organization.* New York: Academic, 1977, pp. 113–128.

[11] D. D. Gajski and L. P. Rubinfield, "Design of arithmetic elements for Burroughs scientific processor," in *Proc. 4th Symp. Comput. Arithmetic*, Santa Monica, CA, 1978, pp. 245–256; also in *Proc. 1978 LASL Workshop Vector and Parallel Processors*, Los Alamos, NM, 1978.

[12] R. G. Hintz and D. P. Tate, "Control data STAR-100 processor design," in *Proc. IEEE COMPCON 1972*, Sept. 1972, pp. 1–4.

[13] D. J. Kuck, "ILLIAC IV software and application programming," *IEEE Trans. Comput.*, vol. C-17, pp. 758–770, Aug. 1968.

[14] ——, "Parallel processing of ordinary programs," in *Advances in Computers*, vol. 15, M. Rubinoff and M. C. Yovits, Eds. New York: Academic, 1976, pp. 119–179.

[15] ——, "A survey of parallel machine organization and programming," *ACM Comput. Surveys*, vol. 9, pp. 29–59, Mar. 1977.

[16] ——, *The Structure of Computers and Computations*, Vol. I. New York: Wiley, 1978.

[17] D. J. Kuck, Y. Muraoka, and S. C. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speed-up," *IEEE Trans. Comput.*, vol. C-21, pp. 1293–1310, Dec. 1972.

[18] D. H. Lawrie and C. R. Vora, "Multidimensional parallel access computer memory system," U.S. Patent No. 4,051,551, Sept. 27, 1977.

[19] ——, "The prime memory system for array access," *IEEE Trans. Comput.*, vol. C-31, this issue, pp. 435–442.

[20] M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *J. Ass. Comput. Mach.*, vol. 15, pp. 252–264, Apr. 1968.

[21] J. A. Rudolph, "A production implementation of an associative array processor—STARAN," in *Proc. 1972 AFIPS Fall Joint Comput. Conf.*, vol. 41, 1972, pp. 229–241.

[22] R. M. Russell, "The CRAY-1 computer system," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 63–72, Jan. 1978.

[23] A. H. Sameh and R. P. Brent, "Solving triangular systems on a parallel computer," *SIAM J. Numer. Anal.*, vol. 14, pp. 1101–1113, Dec. 1977.

[24] F. H. Sumner, "MU5—An assessment of the design," in *Proc. IFIP Congress, Information Processing 1974.* Amsterdam, The Netherlands: North-Holland, 1974, pp. 133–136.

[25] D. W. Sweeney, "An analysis of floating-point addition," *IBM Syst. J.*, vol. 4, no. 1, pp. 31–42, 1965.

[26] J. F. Thorlin, "Code generation for PIE (Parallel Instruction Execution) computers," in *Proc. AFIPS Spring Joint Comput. Conf.*, 1967, pp. 641–643.

[27] J. E. Thornton, *Design of a Computer, the Control Data 6600*. Glenview, IL: Scott, Foresman, and Co., 1970.

[28] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Develop.*, vol. 11, pp. 25–33, Jan. 1967.

[29] W. J. Watson, "The TI ASC-A highly modular and flexible super computer architecture," in *Proc. 1972 AFIPS Fall Joint Comput. Conf.*, 1972, pp. 221–228.

David J. Kuck (S'59–M'69) was born in Muskegon, MI, on October 3, 1937. He received the B.S.E.E. degree from the University of Michigan, Ann Arbor, in 1959, and the M.S. and Ph.D. degrees from Northwestern University, Evanston, IL, in 1960 and 1963, respectively.

From 1963 to 1965 he was a Ford Postdoctoral Fellow and Assistant Professor of Electrical Engineering at the Massachusetts Institute of Technology, Cambridge. In 1965 he joined the Department of Computer Science, University of Illinois, Urbana, where he is now a Professor. Currently, his research interests are in the coherent design of hardware and software systems. This includes the development of the PARAFRASE system, a program transformation facility for array and multiprocessor machines. His recent computer systems research has included theoretical studies of upper bounds on computation time, empirical analysis of real programs, and the design of high-performance processing, switching, and memory systems for classes of computation ranging from numerical to nonnumerical. The latter work includes the study of interactive text processing and database systems, from the point of view of both the computer and the user.

Dr. Kuck has served as an Editor for a number of professional journals, including the IEEE TRANSACTIONS ON COMPUTERS, and is presently an area editor of the *Journal of the Association for Computing Machinery*. Among his publications are *The Structure of Computers and Computations*, vol. I. He has consulted with many computer manufacturers and users and is the founder and president of Kuck and Associates, Inc., an architecture and optimizing compiler company.

Richard A. Stokes was born in the Bronx, NY, on June 30, 1931. He received the A.B. degree in mathematics from St. Michael's College, Winooski, VT, and the B.S.E. degree in computer science from George Washington University, Washington, DC.

Following military service he joined the National Security Agency where he was involved in the early application of magnetic cores and transistor to computer design. In 1960 he joined the Martin Company as project leader in the development of CRT graphic displays. In 1964 he joined the Burroughs Corporation, Paoli, PA, as a Staff Engineer working on the design of the B 8501 computer. Later he became Lead Engineer in the development of the Illiac IV computer. In 1976 he was made General Manager of the plant responsible for the design and manufacture of the BSP. He holds several patents in computer design including the B 8501, the Illiac IV, and BSP.