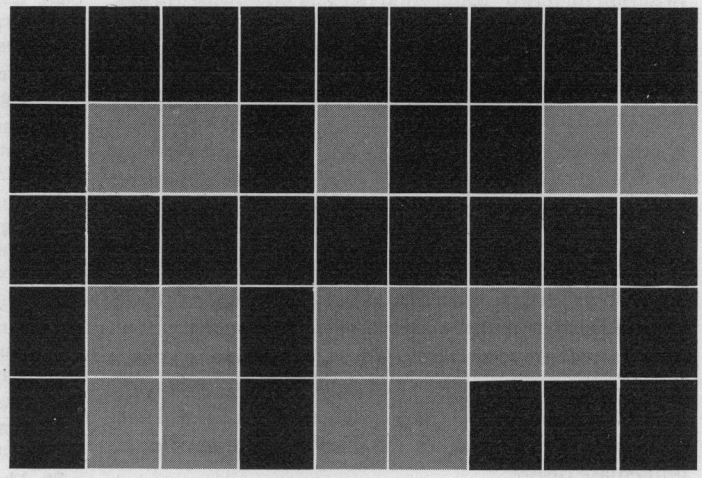
Architecture, not hardware, gives the FPS array processor family its computational speed. Here is an inside look at the trade-offs and ideas that went into its design.

An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family



Alan E. Charlesworth
Floating Point Systems, Inc.

In 1975, Floating Point Systems introduced the AP-120B, the initial member of its array processor family. George O'Leary, vice president of engineering, and I codesigned the architecture. A major inspiration was an integer array processor designed and implemented by Glen Culler of Culler Harrison, Inc. The AP-120B and two later models, the AP-190L and FPS-100, have been used primarily in signal processing. In 1980, the FPS-164 was introduced to extend our AP architecture into large-scale scientific computing.

This article centers on design trade-offs and the use of internal architecture in our APs, features that determine how fast they can perform useful arithmetic. External architecture issues, such as the best way to interface an AP to a host computer, are outside the present scope of this discussion.

Table 1.
Arithmetic/memory activity ratios.

	ADDITIONS	MULTIPLICATIONS	MEMORY ACCESSES	RATIO
CONVOLUTION	1	1	2	1:1:2
FFT (RADIX 4)	1.8	1	2.3	1.8:1:2.3
VECTOR INNER PRODUCT	1	1	2	1:1:2
MATRIX ROW ELIMINATION	1	1	3	1:1:3
	ADDITIONS	**MULTIPLICATIONS**	**DIVISIONS**	**RATIO**
KNUTH MEASUREMENT	2.3	1	0.38	2.3:1:0.38
GIBSON MIX	1.8	1	0.39	1.8:1:0.39

FPS-AP architectural principles

The design objective of the FPS-AP family is to efficiently perform the computationally intensive portions of scientific and signal processing. The architectural challenge was to devise arithmetic elements, memories, and registers that could be programmed effectively and manufactured at a reasonable cost. The first design step was to turn the general goal into the following seven architectural guidelines.

Optimize both scientific and signal processing. Array processors evolved for signal processing applications, which are usually dominated by two algorithms: the fast Fourier transform and convolution.[20] Some array processor architectures have been fine-tuned through the omission of some capabilities not required for these algorithms.

We also examined benchmarks from more general types of scientific computing and decided to go beyond signal processing. The challenge became one of adding capabilities sufficient to perform scientific computing well while not compromising cost effectiveness.

At the root of many scientific problems lies the solution of simultaneous linear equations such as $Ax=b$ and related matrix operations.[38] Two basic kernel computations dominate: the inner (dot) product of two vectors and the elimination of one row by another.[61] These can be expressed as follows:

$$\text{Inner product: } A = \text{SUM}(X(i) \times Y(i)) \quad \text{for } i \text{ from 1 to } N$$

$$\text{Elimination: } Y(i) = Y(i) - A \times X(i) \quad \text{for } i \text{ from 1 to } N$$

0018-9162/81/0900-0018$00.75 © 1981 IEEE

where X and Y are rows or columns out of an $N \times N$ two-dimensional matrix.

In addition to these benchmark kernels, Knuth[54] and Gibson[44] have measured the behavior of scientific Fortran programs (Table 1).

To perform these algorithms at maximum speed, a processor must be able to initiate a floating-point multiplication, two floating-point additions, and three memory accesses simultaneously in every machine cycle. The processor would then be balanced for these benchmark problems.

Given the complexity of high-speed floating-point arithmetic, we chose to include the minimum number of floating-point arithmetic elements: a multiplier and an adder. One large memory was included, along with a second, smaller memory for use in computations needing more memory bandwidth. This choice balanced the FPS-APs for the vector inner product and convolution and brought it within a factor of 1.8× for the FFT and 1.5× for matrix row elimination.

Since divisions do not appear at all in the kernel benchmarks and only occasionally in the Fortran measurements, no separate divider unit was included. Instead, division is programmed in software by a small table look-up and a Taylor series expansion involving multiplications and additions. Using this approach, the vector divide algorithm is 43 percent as fast as the vector multiply or add.

The 38-bit word chosen for the AP-120B gave it a decimal digit of accuracy over the 32-bit word used by other array processors. This additional accuracy has allowed use of the AP-120B in scientific applications requiring more than 32-bit precision but less than 64-bit precision. An example is the simulation of electrical transmission grid power flow and transient stability. The word length on the FPS-164 is a full 64 bits, meeting the accuracy needs of most scientific computing.

Balance vector and scalar capabilities. Most of an array processor's calculations consist of operations on ordered sets of data, usually vectors and matrices. Because vector computations are repetitive, their performance can often be improved by the use of special hardware; however, the scalar parts of a computation are usually more random and heterogenous, and therefore must be done sequentially.

If vector and scalar runtimes are roughly equal, neither takes a disproportionate amount of execution time—the two capabilities are balanced. Figure 1 formulates balanced vector/scalar performance.

Amdahl[11] estimates that it is reasonable to obtain an overall vector fraction F of about 75 percent for typical scientific problems. Such a vector fraction yields a balanced vector/scalar performance ratio of 3:1. Given a moderate vector/scalar performance ratio, a vector processor also performs well on short vectors and sequential operations. Therefore, we adjusted the cycle time of our APs to shorten the functional unit pipelines to only two and three stages. Some vector processors, such as the Star-100, have much longer pipelines and perform relatively poorly on short vectors.

Use only a single processor. Parallelism can be exploited to a first approximation either by giving more power to a conventional single-processor computer such as the CDC 7600, or by coordinating a collection of multiple processors such as the Illiac or Computer Signal Processing's Map series of array processors. In the single-processor approach, a program can exploit parallelism by overlapping several operations among the multiple subunits within the single processor. In the multiprocessor approach, several different programs can operate concurrently in separate processors and communicate via flags or interrupts.

The multiprocessors provide more power for pure vector operations, but difficulties arise in coordinating the several processors for scalar operations. We chose the single-processor approach because it is easier to implement, is more familiar to users of conventional computers, and performs better for a moderate mix of vector and scalar operations.

Perform vector operations with scalar hardware. Most vector supercomputers, such as the Cray-1, provide one set of hardware to perform scalar operations and another to perform vector computations. Significant savings were made in the FPS-APs by making the scalar hardware powerful enough to process vectors. Software loops constructed from scalar operations are used to program vector operations. The scalar hardware can also be used to overlap unrelated scalar operations, something that could not be done with vector hardware.

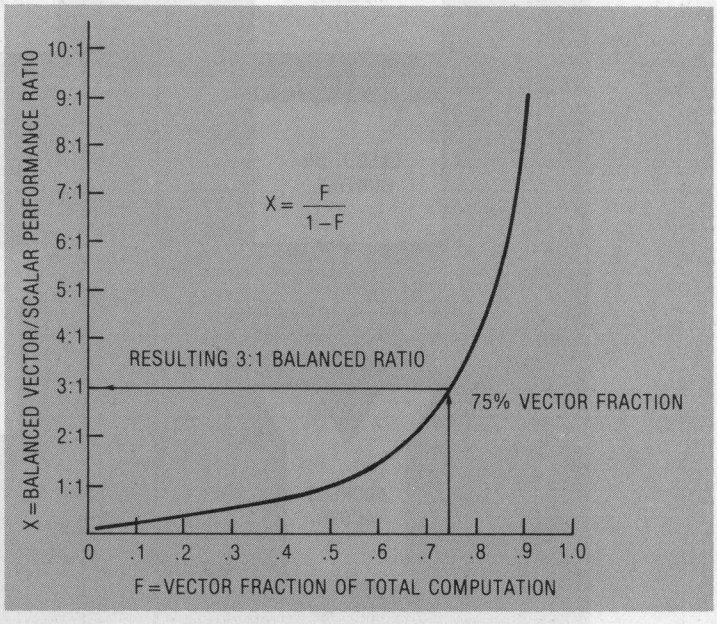


Figure 1. Balanced vector/scalar performance ratio.

Table 2.
AP vector/scalar balance.

		VECTOR	SCALAR	RATIO
MULTIPLY/ADD	$(A+B\times C)$	3 CYCLES	10 CYCLES	3.3:1
DIVIDE	(A/B)	7 CYCLES	22 CYCLES	3.1:1

Detect parallelism with software rather than hardware. Much more than conventional computers, parallel processors are burdened by complicated control logic, which is needed to synchronize and coordinate concurrent activities. We shifted most of this burden to the programmer or compiler. The programmer or compiler explicitly codes all parallelism into the program only once, so that the hardware need not detect it millions of times per second. In our APs, extra hardware is devoted to parallel data paths and multiple functional units rather than to large amounts of extra control and synchronization logic. Thus, a conscious trade-off balanced hardware and software complexity and kept the price of FPS-APs substantially below that of supercomputers.

Program directly at the hardware level. Modern conventional computers involve several layers of execution. A compiler translates higher-level languages such as Fortran into assembly language, then a microprogram executed by the hardware interprets the assembly language. Flynn shows that the assembly-language level of most computers is optimal neither for the microcode to interpret nor for the compiler to use.[37] This is a result of the assembly-language level for the common computer families (IBM-370, PDP-11, etc.) being frozen a decade or more ago under hardware constraints vastly different from today's. One alternative, identified by Flynn, achieves maximum execution by compiling "direct to microcode."[37] This eliminates the intermediate levels, allowing the compiler to generate microcode with direct access to the internal parallelism of the hardware.

This tactic, chosen for our APs, has some drawbacks that weigh against the gain in execution speed. An assembly-level instruction can correspond to four or five microinstructions, causing compiled code sizes to be proportionately larger than those for a conventional computer. In addition, FPS-AP compilation is more complicated and time-consuming than compilation for a conventional computer. Thus, it must be assumed that faster code execution times are more important than slower compile times. Execution speed and moderate hardware cost came at the expense of increased software complexity.

Implement with standard hardware. The combined effect of the above strategies increased execution speed over conventional computers by exploiting software-detected

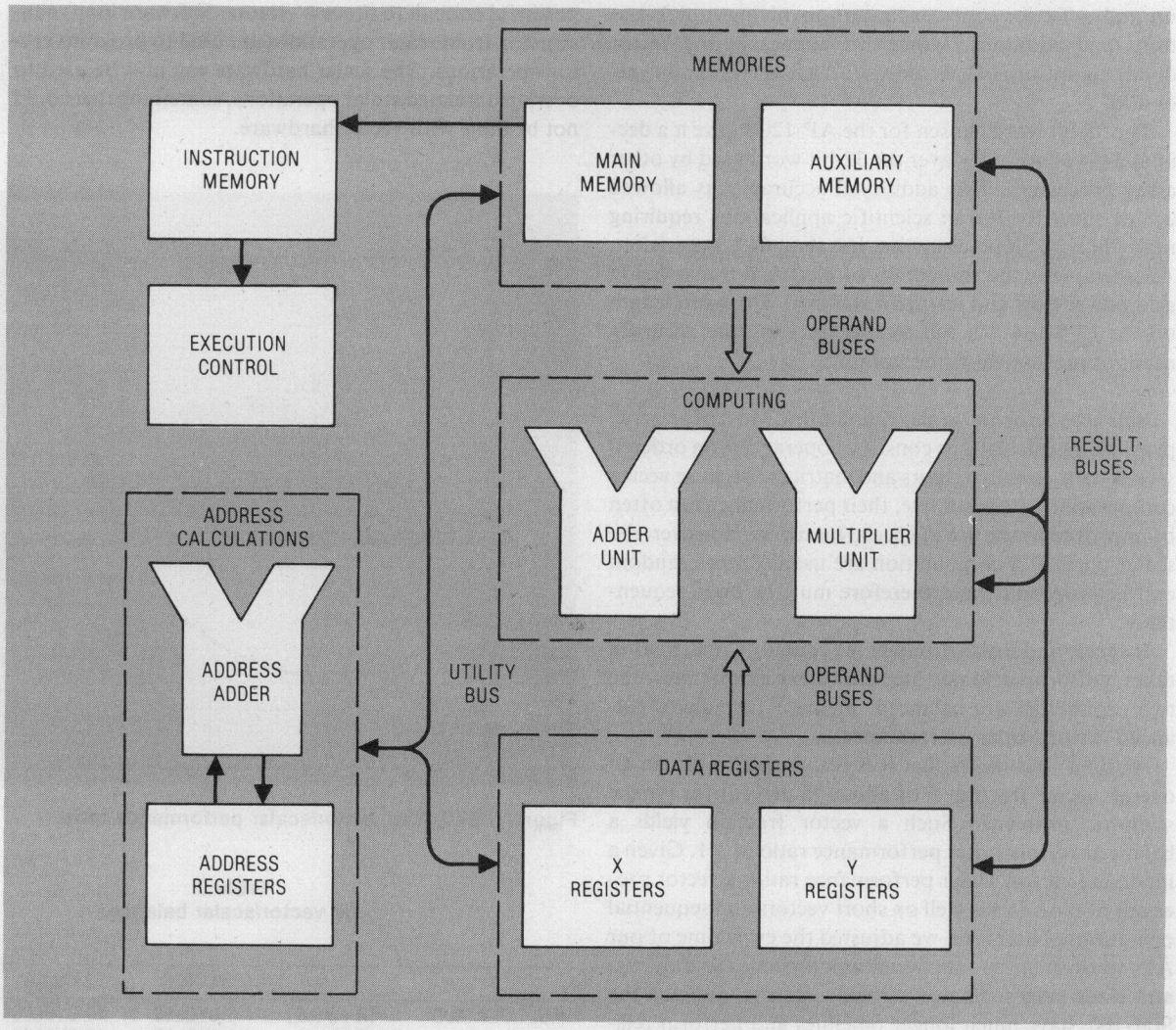


Figure 2. AP multiple functional units.

parallelism, not through raw hardware speed. The hardware is no faster than the internal microprocessors of conventional computers. For example, an instruction executes every 167 nanoseconds.

Primary FPS-AP architectural features

Using these seven principles, we devised a processor architecture with three primary features: multiple functional units, multiple interconnections, and multi-operation instructions.

Multiple functional units. The multiple functional units (Figure 2) provide the basic parallelism of FPS-AP architecture by allowing several independent operations (one in each unit) to occur simultaneously. Each functional unit has a given task and can be more effective on its narrow specialty than can general-purpose hardware. Multiple functional units have a long history on scientifically oriented mainframe computers such as the CDC-7600.

The functional units allow a maximum of two data computations, two memory accesses, an address computation, four data register accesses, and a conditional branch to be initiated in a given CPU cycle, for a total of 10 distinct operations.

The operations that the adder, multiplier, and memory perform are inherently more time-consuming than those of the registers and addressing units. Since the processor must be capable of starting a new operation in every functional unit on each cycle, we could have lengthened cycle time to match the slowest unit. Instead, the more complicated units were divided into several equal time steps called *stages*. The stages are organized like a bucket brigade, each passing its partial results to the next stage in the pipeline.

The pipelined functional units (Figure 3) have two or three internal stages, enough to enable all the units to initiate a new operation in every 167-nanosecond cycle. However, a given result is not completed until two or three cycles later. Thus, while parallel multiplications can be done every cycle, a purely sequential multiplication takes three cycles. These relatively short pipeline lengths balance long-vector performance against short vectors and scalars.

Arithmetic Functional Units. The *adder unit* performs floating-point additions and subtractions and data format conversion. In the FPS-164, it also performs integer arithmetic, logical operations, shifts, and divide/square root approximations. This makes the adder capabilities more symmetrical for compiler code generation.

The *multiplier unit* performs floating-point multiplications. In the FPS-164, it also performs integer multiplications.

Memory functional units. The primary store for data words is *main memory*. It is the secondary store for instruction words. On the AP-120B, it has a maximum size of 384,000 words; this is extended to 1.5 million words on the FPS-164 to accommodate large scientific problems. Memory mapping registers are also added on the FPS-164 to provide the separately protected instruction and data spaces needed for multiple users.

Auxiliary memory is smaller and optional. Used by algorithms with high memory bandwidth requirements (like the FFT and matrix multiplication), this second memory can double the solution speed of memory-limited problems. Its size ranges from 8000 to 32,000 words. The auxiliary memory also has a read-only section for frequently used tables of constants, such as a cosine table for FFTs.

Data registers. The *X and Y data register units* each contain 32 registers. One register location can be read and another written from both of the register units on every CPU cycle, a total of four simultaneous accesses. These registers provide temporary fast locations for intermediate results and help to reduce traffic in and out of the main memory.

Address calculation units. The *address adder unit* adds, subtracts, and performs logical operations on memory addresses, loop counters, and integer data contained in the address registers. The width of the address arithmetic

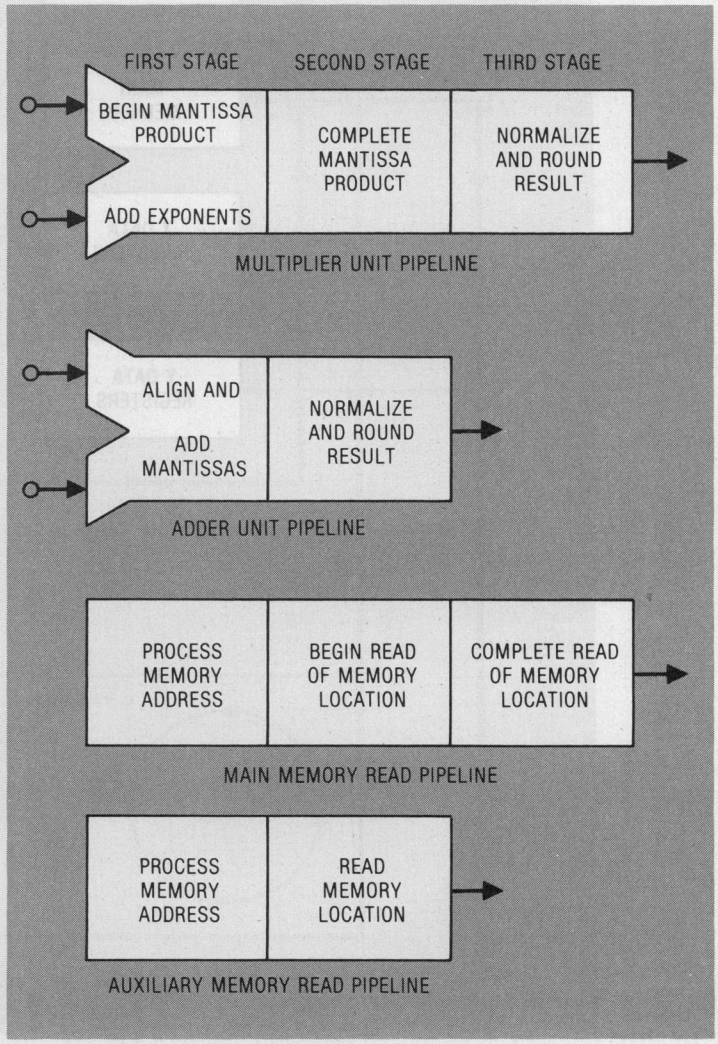


Figure 3. The AP functional unit pipelines.

is increased from 16 bits on the AP-120B to 32 bits on the FPS-164 to allow direct addressing of larger memory spaces.

The *address registers* provide space for the calculations needed to address array elements. They are analogous to index registers on conventional computers. The number of registers is increased from 16 on the AP-120B to 64 on the FPS-164 to accommodate the compiler's need for addressing temporaries.

Execution control. The *instruction memory* is dedicated to instruction words. Keeping code fetches separate from data fetches allows both to proceed simultaneously without mutual interference. They range from 1000 to 4000 instruction words. The instruction memory of the AP-120B must be manually loaded from main memory or from the host computer. Extra hardware is added on the FPS-164 to automatically load code words from main memory to accommodate large programs.

The *branch control* allows the processor to decide which instruction to execute next in parallel with completion of the current instruction. Branches test the state of the machine from the previous instruction, as they usually do on conventional computers. The FPS-APs can perform decision operations nearly as well as conventional computers. Many other vector processors significantly penalize branching.

Multiple interconnections. The multiple functional units require a network of interconnections dense enough to allow the concurrent flow of sufficient amounts of data between units to keep them all fully occupied (Figure 4).

Computing produces six individual results: from the adder, the multiplier, the two register units, and the two

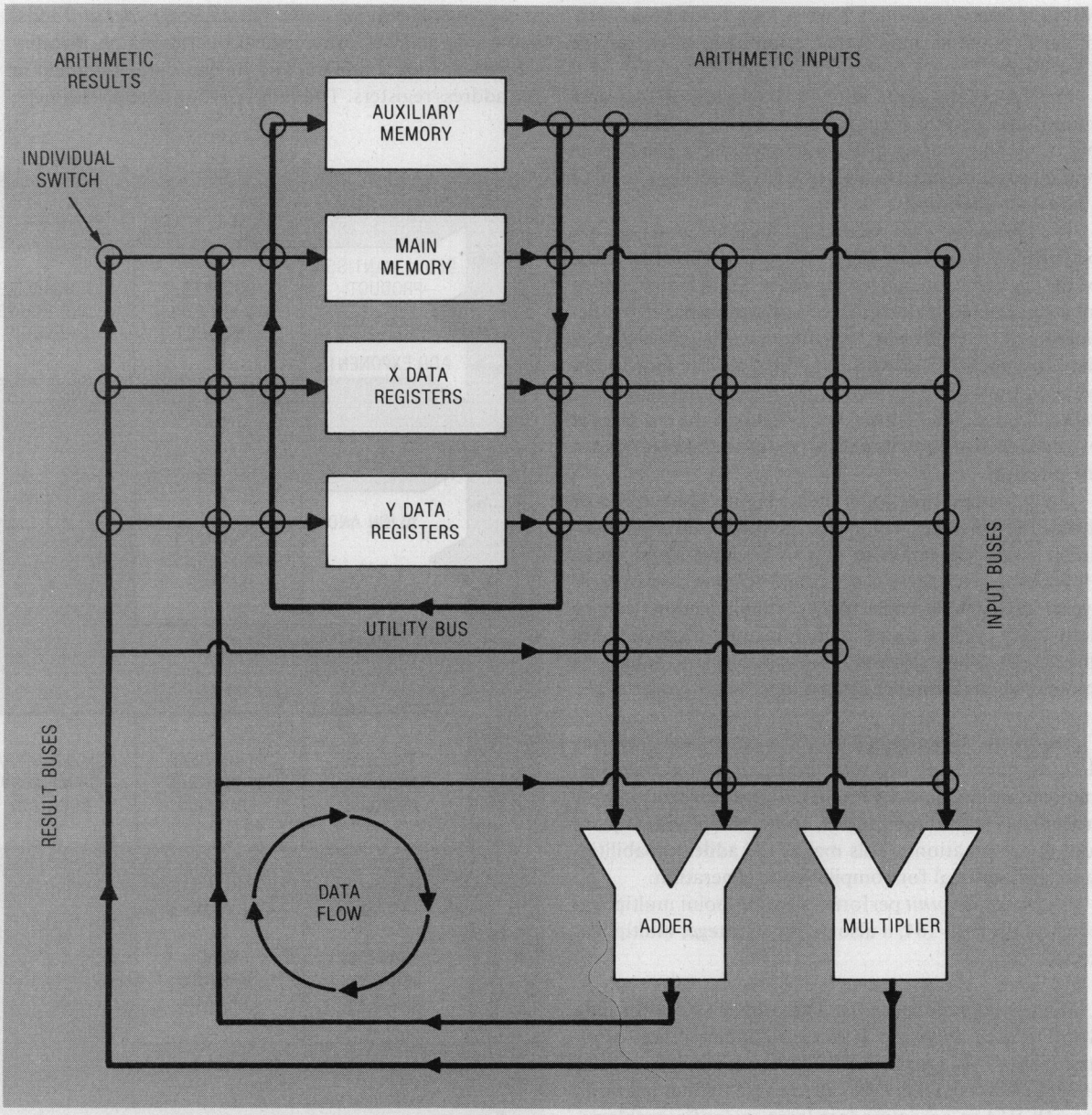


Figure 4. The AP interconnection network.

memories. These results must be connected to the eight individual inputs: the four arithmetic operands, the two register units, and the two memories.

Interconnection schemes[57] range from a single bus, where only one item can be transferred at a time, to a full crossbar switch, where all possible connections can be made simultaneously.

A full crossbar for our APs would have had 6×8 connections to be switched simultaneously. For the FPS-164 with 64-bit wide words, this would have meant switching 3072 individual data lines simultaneously. This is beyond the practical limits of printed circuit connectors. Instead, the designers chose a partial crossbar that makes 30 of the possible 48 connections. The crossbar is implemented with six dedicated buses, four to supply operands to the arithmetic units and two to carry results away from the arithmetic. A seventh nondedicated utility bus links the register and memory units.

The densest portion of the interconnect is centered on the two register units, which are connected to all seven buses. They act as temporary buffers for arithmetic results, especially when another unit cannot immediately use a result. The register bandwidth is limited, however, to two reads and two writes per cycle. If both memories and both arithmetic units are active, a total of six reads and four writes are required.

The programmer overcomes the gap between the register bandwidth of four accesses per cycle and the maximum requirement of ten accesses per cycle by bypassing the registers and using the direct connections between the memories and the arithmetic units. Each of the four memory and arithmetic results connect directly to one of the two inputs of the adder and multiplier. Data can flow directly from memory into the arithmetic and back into memory without using the registers. Each time a direct connection is used, a cycle of delay is saved. Without the direct connections, the pipelines would effectively be one stage longer.

The partial crossbar represents a trade-off between implementation constraints and maximum execution speed. Five years of FPS-AP coding experience have shown the compromise to be a favorable one.

Multi-operation instructions. To run at full speed, each of the ten functional units needs its own subinstruction on every cycle. The 64-bit instruction word (Figure 5) contains a separate subinstruction parcel for each of the 10 functional units. This solution is similar to the wide control words used by the microprocessors that implement most conventional computers.[15] Using the 10 parcels, a program can specify a new operation for each of the 10 functional units on every CPU cycle.

The parcels are analogous to the individual instructions of conventional computers. The adder parcel, for example, specifies an operation (e.g., subtract) and selects two operands—register contents, memory outputs, or previous results—to be subtracted. The 10 parcels allow a single instruction to add, multiply, fetch numbers from each memory, decrement a loop counter, and test for loop completion. A complete vector loop is possible in a single instruction.

The 167-nanosecond CPU instruction cycle rate allows a maximum of 60 million total operations, or 12 million floating-point operations per second. The 10 possible concurrent operations approximately correspond to the five instructions (multiply, add, two memory accesses, and a conditional branch) of a conventional register-oriented computer. They yield a maximum rate of 30 million instructions per second.

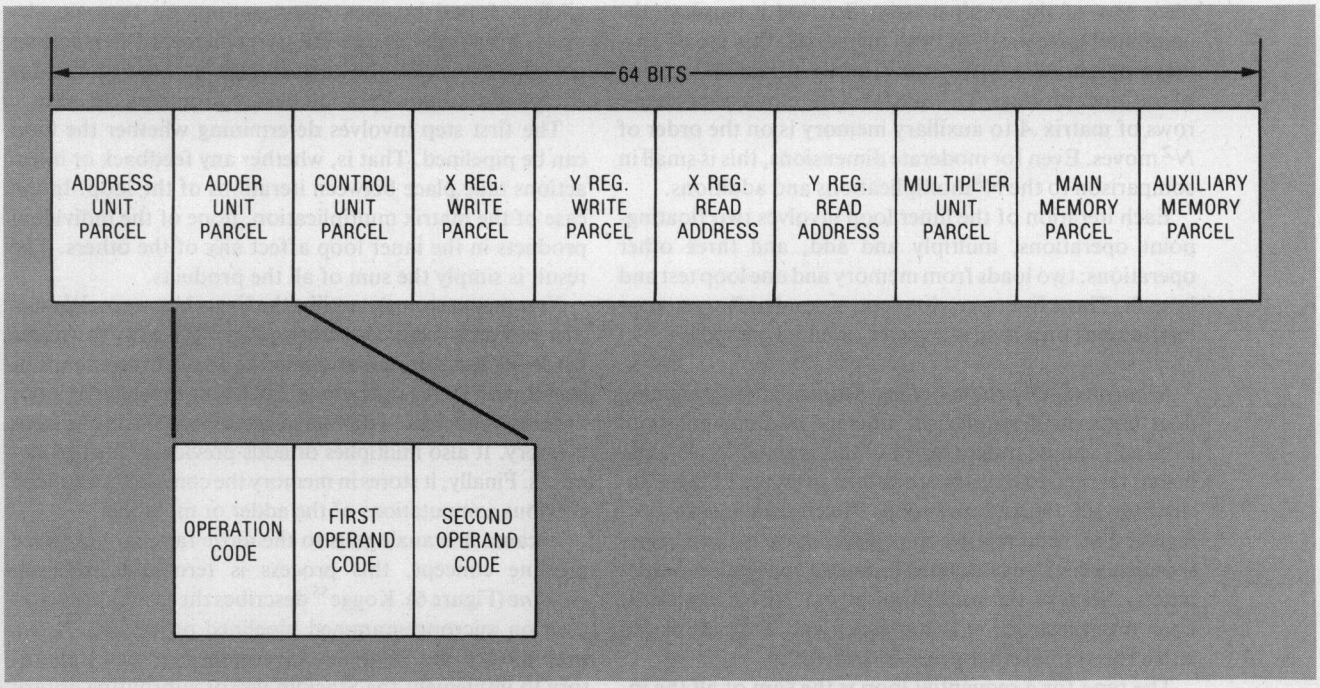


Figure 5. The AP multiparcel instruction word.

Computing techniques using FPS-AP architecture

FPS-AP architecture can be programmed at three levels of parallelism,[57] depending upon the characteristics of the problem to be solved.

- *Sequential programming:* Operations follow one after another without overlap.
- *Overlapped programming:* Local parallelism is exploited by paralleling individual arithmetic and memory operations.
- *Pipelined programming:* Global parallelism is exploited by paralleling the entire computation on one element of a vector with those on preceding and succeeding elements.

On our APs, outer loops and the less frequently used portions of an algorithm are programmed according to the overlapped method. Inner loops are pipelined when possible, since they usually constitute the vast majority of total execution time. Programs are sequentially coded only for debugging purposes.

An example problem. The inner loop of the matrix multiplication $C = A \times B$ illustrates the use of these three programming methods on FPS-APs. A given point of the result matrix is computed by summing the products of the appropriate row of matrix A by the appropriate column of matrix B. For an $N \times N$ matrix, this process is an N element vector inner product of each row of A with each column of B: a total of N^3 products to be summed.

The two input matrices and the result matrix are stored in main memory. Successive rows of matrix A are copied into auxiliary memory, which is used as a temporary row buffer. The inner product of a particular row is computed with each successive column of matrix B to form a complete row of the result matrix. Because it employs the combined bandwidth of both memories, this use of auxiliary memory as a buffer significantly increases the speed of the inner product. The overhead for moving successive rows of matrix A to auxiliary memory is on the order of N^2 moves. Even for moderate dimensions, this is small in comparison to the N^3 multiplications and additions.

Each iteration of the inner loop involves two floating-point operations, multiply and add, and three other operations: two loads from memory and one loop test and branch. These five operations correspond to five separate instructions on a typical register-oriented computer.

Sequential AP programming. Sequential programming does not exploit parallelism inherent in a computation. FPS-APs can be programmed in this manner by defining macro register-to-register operations from the FPS-AP instruction set. Operations such as "fetch from memory-to-register" or "add register-to-register" have a direct correspondence to the instructions found on conventional computers. None of the parallelism of our APs is exploited. Each macroinstruction is completed and its result placed into a register before the next is started.

The time for a sequential loop is the sum of all the individual operation times in a computation. For the matrix multiplication this is 16 cycles. The resulting rate is 0.75 MFLOPs, or 1.9 MIPs.

Overlapped FPS-AP programming. Overlapped programming exploits the local parallelism often found within a computation. In our AP family, an unrelated calculation can begin before the previous operation finishes. In an expression like $(A \times B) + (C \times D)$, the two multiplications can overlap. CDC-7600/Cyber-170 computers are programmed using this method.

Overlapping is usually applied to a section of code called a *basic block*, which has only one entry at the top and one exit at the bottom. The absence of multiple entries or exits from a block allows the operations inside to be rearranged for speed, with no external effect. The various sequences of dependent computations within a basic block are examined to find the longest, or *critical*, path. The best case is to overlap all the shorter paths with the critical path.

The longest path for the matrix multiplication inner loop is nine cycles, the sum of the pipeline lengths in the critical path. Overlapping the inner loop compresses the entire computation into nine instructions. Any unavoidable resource conflicts, such as the need to perform two additions at once, would increase the actual length beyond this lower bound.

The lower bound for an overlapped loop is the sum of the times of operations in the critical path. For the matrix multiplication this is nine cycles. The resulting rate is 1.3 MFLOPs, or 3.3 MIPs.

The overlapped method has wide applicability to general computing, since most scalar problems have significant local overlap. The FPS-164 Fortran compiler generates overlapped code that is very competitive with that done by hand.

Pipelined method. Pipelining exploits the global parallelism found between computations on separate elements in a vector. If operations on one set of elements do not affect those on the next, there is no need to wait for completion of one iteration before beginning the next.

The first step involves determining whether the loop can be pipelined. That is, whether any feedback or interactions take place between iterations of the loop. In the case of the matrix multiplication, none of the individual products in the inner loop affect any of the others. The result is simply the sum of all the products.

Vector supercomputers like the Cray-1 have special units that perform vector operations. On FPS-APs, the scalar hardware units are programmed to use software loops in performing vector operations. Each time through the loop, the program fetches a new set of vector input elements from memory. It also multiplies or adds previously fetched elements. Finally, it stores in memory the completed results of previous computations of the adder or multiplier.

Because it is analogous to the more familiar hardware pipeline concept, this process is termed a *software pipeline* (Figure 6). Kogge[55] describes the technique as it is used on microprogrammed pipelined processors. It has been used at the Lawrence Livermore National Laboratory to implement the Stacklib vector subroutine library for the CDC-7600.

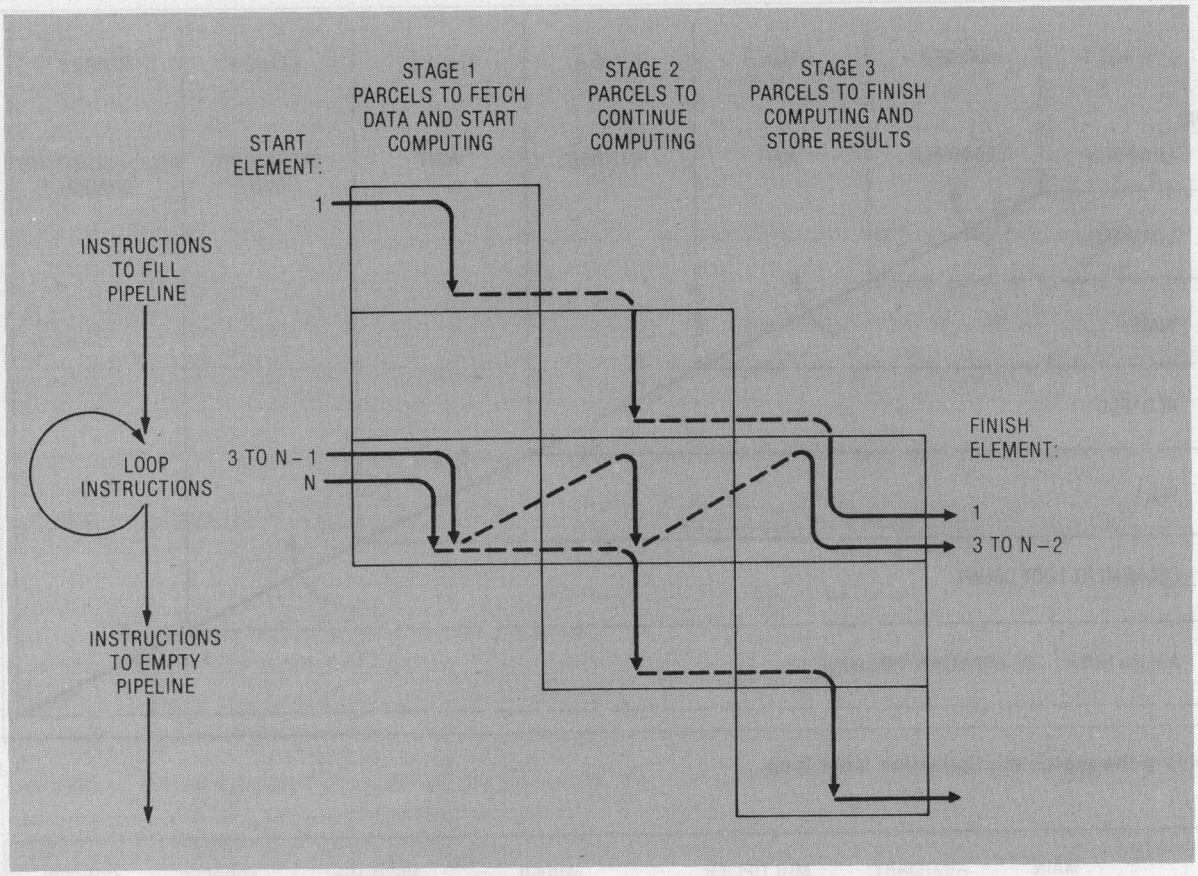


Figure 6. Example of a three-stage software pipeline.

In general, however, it is an unfamiliar concept, since a CPU must have a high degree of programmable internal parallelism to make software pipelines useful. I/O operations, on the other hand, can be performed in parallel with CPU activity on almost all computers. It is thus more common to pipeline the input-compute-output process, fetching blocks from the I/O device before they are needed by a program. Software pipelining is much like read-ahead I/O buffers, but on a more microscopic level.

The number of instructions I in a software-pipelined loop determines the number of CPU cycles spent per iteration, and hence the computation rate of the pipeline. Since each functional unit can be used once per instruction, the lower bound on the minimum number of instructions is limited by the most frequently used functional unit, or *critical resource*. For moderately simple loops such as the radix-four FFT (or vector dyads and triads), the lower bound can almost always be achieved. In more complicated situations, it might not be possible to route data directly between functional units because of the partial crossbar. This overhead is part of the trade-off against the cost of a full crossbar.

The number of times around the loop required to finish a task is the number of stages S in the pipeline; this number determines the start-up overhead for the pipeline. If L is the number of instructions required to overlap the computation, the number of stages is on the order of L/I. The total time for an N-element pipelined computation is $I \times (S+N)$.

The following four-step process is used to pipeline a problem.

(1) Program the computation using the overlapped method to estimate the critical path length L. L is nine instructions for the matrix multiplication. Test the algorithm for correctness before proceeding.

(2) Determine the critical resource of the computation, which in turn determines the minimum possible number of instructions I and the number of stages ($S=L/I$). The matrix multiplication has the functional unit usage given in Table 3.

Table 3.
Matrix multiplication functional unit usage.

ONE	LOAD FROM MAIN MEMORY
ONE	LOAD FROM AUXILIARY MEMORY
ONE	MULTIPLIER UNIT OPERATION
ONE	ADDER UNIT OPERATION
ONE	ADDRESS UNIT OPERATION (DECREMENT LOOP COUNT)
ONE	CONTROL UNIT OPERATION (BRANCH TO LOOP HEAD)

All the functional units are used equally, once per loop. Therefore, the minimum possible number of instructions in the loop is one. The estimated number of stages in the pipeline is nine.

(3) Generate the pipelined loop code from the overlapped loop by attempting to fold the overlapped

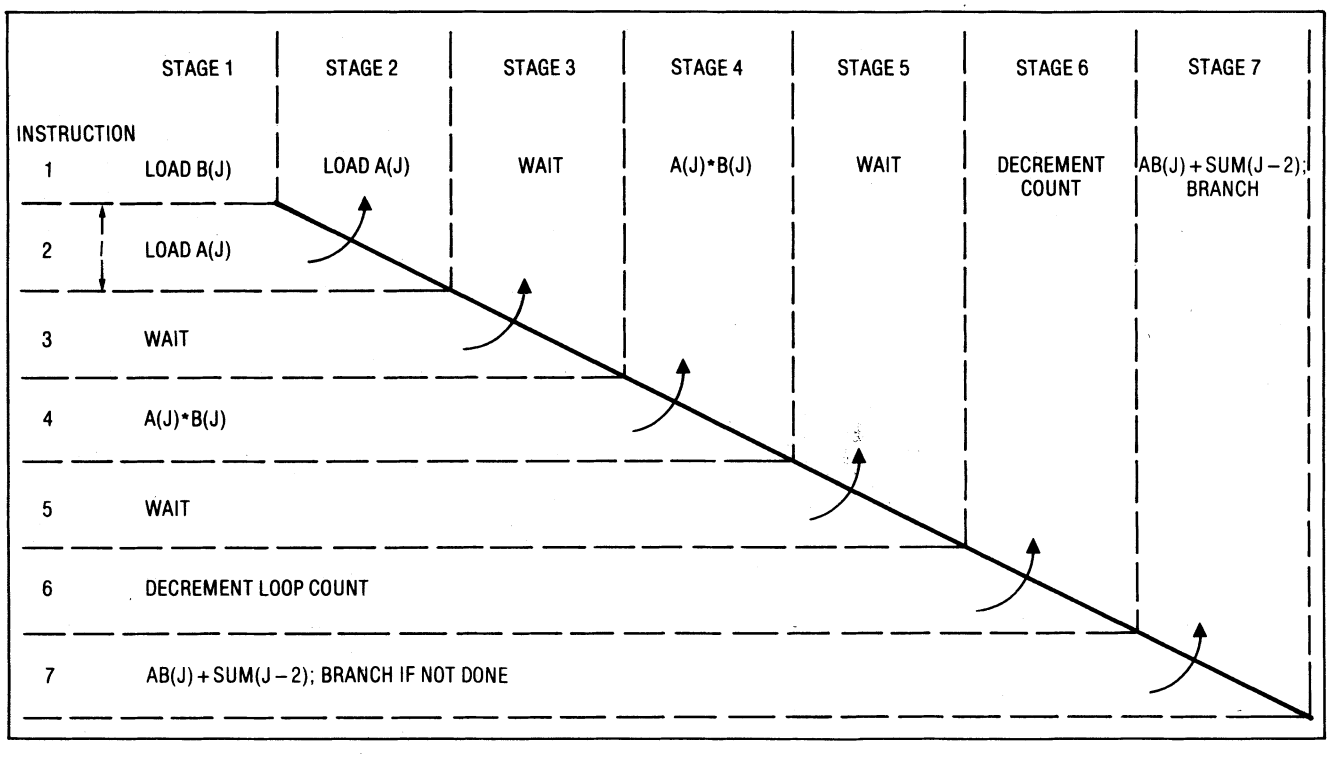


Figure 7. Folding the matrix multiplication inner loop.

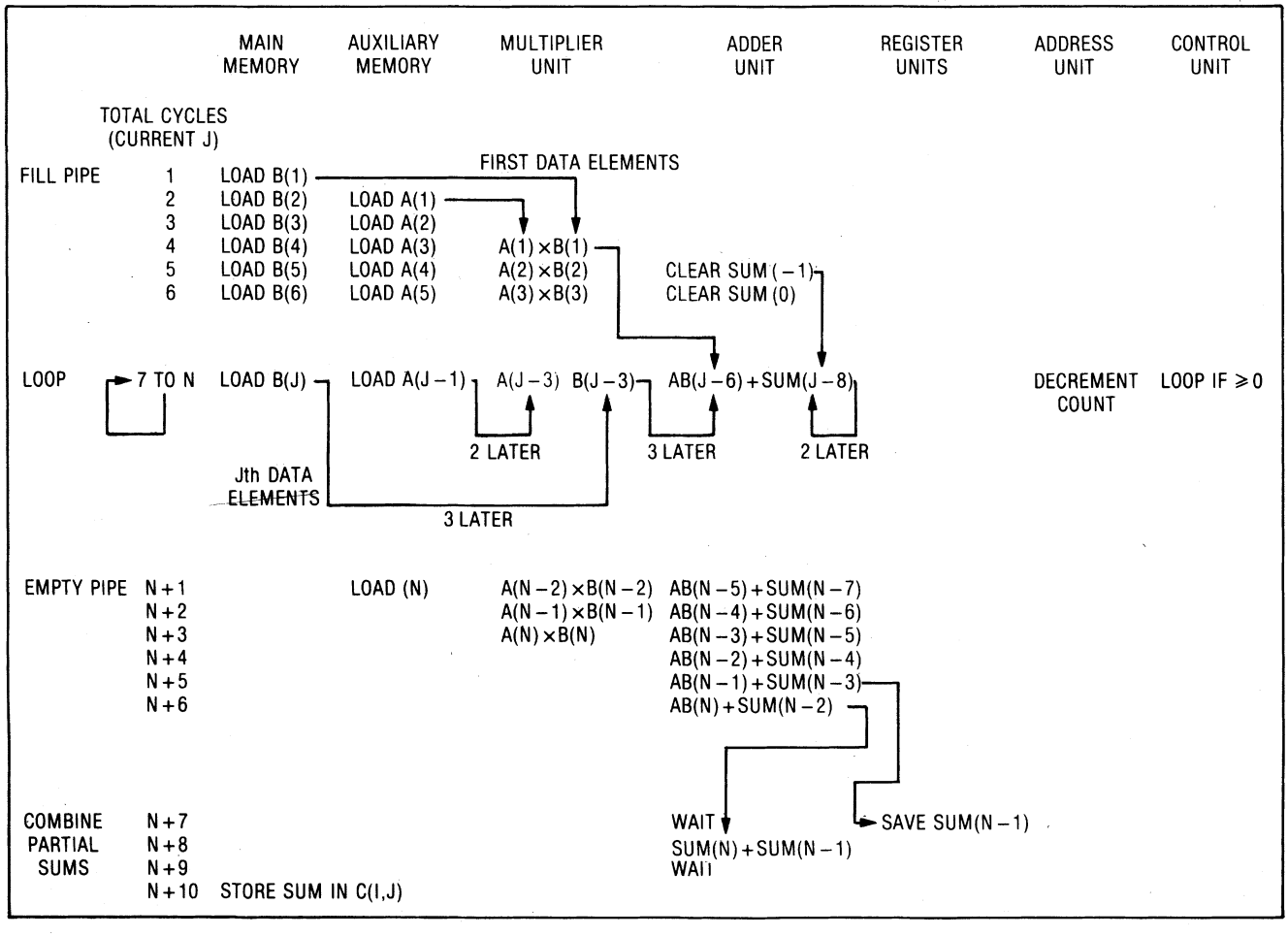


Figure 8. Pipelined matrix multiplication inner loop instructions.

code back onto itself every Ith instruction (Figure 7). If unavoidable conflicts arise due to missing connections in the partial crossbar, lengthen the loop by one instruction and attempt the folding process again. Repeat this step until the entire computation is folded into a pipelined loop. Success is usually attained within a few instructions of the lower bound. It is possible to fold the five operations of the matrix multiplication into a single FPS-AP instruction.

Only seven stages are actually needed for the inner loop (Figure 8), not the estimated nine. This is because each new product is added to the running sum from two elements earlier, without waiting for the sum to complete. Each stage of the adder separately computes half of the total sum. The two partial sums are added at the end of the loop to give the final result.

(4) Generate S-1 "prelude" and S-1 "postlude" sections to fill and empty the pipeline. Each prelude section is programmed to successively include the parcels from an additional pipeline stage. The first section includes only the parcels from stage one, the second from stages one and two, and so forth. The postlude sections are programmed inversely, progressing from the inclusion of all but the first stage to only the last stage.

The matrix multiplication has seven one-instruction stages. Thus, six instructions are needed to fill the pipeline and six more to empty it. At the end, four instructions are needed to add the two partial running sums.

The lower bound for a pipelined loop is one cycle for each critical unit operation in a loop. For the matrix multiplication this is one cycle. The resulting rate is 12 MFLOPs or 30 MIPs. Research is currently underway at FPS to extend the FPS-164 compiler to generate pipelined code.

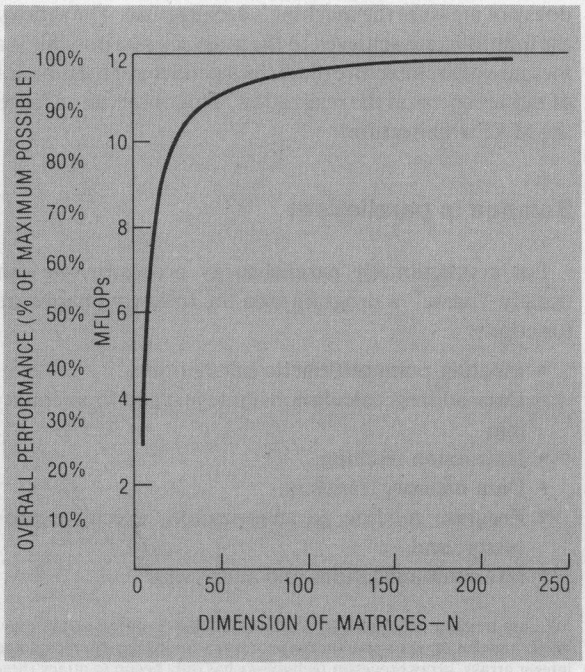


Figure 9. Overall matrix multiplication performance.

Overall matrix multiplication performance. While the inner loop performance is 12 MFLOPs, the need to copy each row of matrix A once from main memory into the auxiliary memory slightly reduces the overall performance. Figure 9 graphs the actual performance relative to the matrix dimension. A size of 40×40 is within 90 percent of the asymptotic rate of 12 MFLOPs.

Pipelining of more complicated algorithms. Software pipelines on FPS-APs are not limited to simple vector additions and multiplications. For example, the inner loop of the radix-4 FFT has 22 floating-point additions, 12 multiplications, 16 main memory read/writes, and nine auxiliary memory reads. It has been programmed in the limiting number of 22 instructions. Quite complicated inner loops have been pipelined on our APs. The limit is usually reached when register space for intermediate results has been exhausted.

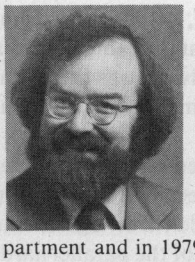
In its first six years, the AP-120B has been successful in a broad range of scientific and signal processing problems. The following factors are central to its wide applicability:

- It is based upon the generality of parallel multiplications and additions on large arrays of data.
- Its vector and scalar capabilities are balanced; neither is a disproportionate bottleneck.
- Parallelism is explicitly controlled by software. The hardware can be kept relatively simple by using software to detect parallelism off-line.

The FPS-164 has expanded the applicability of the FPS-AP architecture to large-scale seientific computing by providing increases in both computational precision and maximum program size. ■

References

The references cited in this article are listed in a separate bibliography included in this issue, pp. 53-57.



Alan E. Charlesworth is technical assistant to the vice president of engineering at Floating Point Systems, Inc. He joined Floating Point Systems in 1974 and in 1975 codesigned the hardware and software architecture of the AP-120B array processor. He established the FPS software department and managed it from 1975 through 1977. In 1978, he joined the new products group of the Advanced Engineering Department and in 1979 led the hardware and software architectural design of the FPS-164 64-bit scientific array processor.

Charlesworth attended Stanford University from 1963 through 1967.