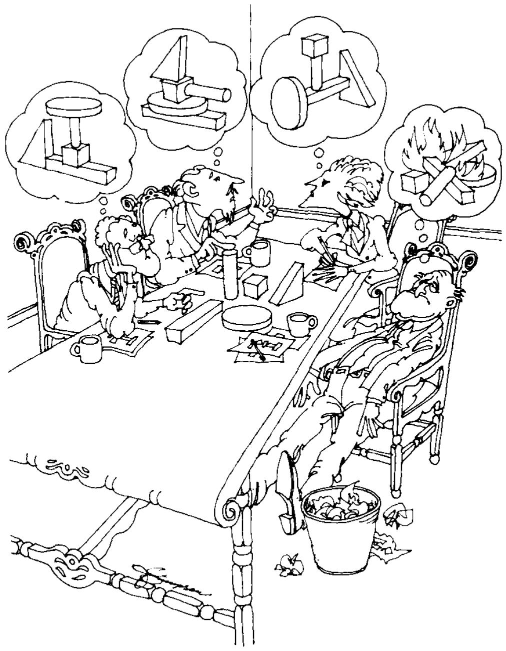


The Cydra 5 Departmental Supercomputer

Design Philosophies, Decisions, and Trade-offs

B. Ramakrishna Rau, David W.L. Yen,
Wei Yen, and Ross A. Towle
Cydrome, Inc.

The Cydra 5 Departmental Supercomputer targets small work groups or departments of scientists and engineers.[1] It costs about the same as a high-end superminicomputer ($500,000 to $1 million), but it can achieve about one-third to one-half the performance of a supercomputer costing $10 to $20 million. This results from using high-speed, air-cooled, emitter-coupled logic technology in a product that includes many architectural innovations.

The Cydra 5 is a heterogeneous multiprocessor system. The two types of processors are functionally specialized for the different components of the work load found in a departmental setting. The Cydra 5 numeric processor, based on the company's directed-dataflow architecture,[2] provides consistently high performance on a broader class of numerical computations than do processors based on other architectures. It is aided by the high-bandwidth main memory system with its stride-insensitive performance. The interactive processors offload all nonnumeric work from the numeric processor, leaving it free to spend all its time on the numerical application. Lastly, the I/O processors permit high-bandwidth I/O transactions with minimal involvement from the interactive or numeric processors.

To meet price-performance targets for a new minisupercomputer, a team of computer scientists conducted an exhaustive—and enlightening—investigation into the relative merits of available architectures.

The Cydra 5 grew from eight years of research and development dating back to work done at TRW Array Processors and at ESL (a subsidiary of TRW). The polycyclic architecture[3] developed at TRW/ESL is a precursor to the directed-dataflow architecture developed at Cydrome starting in 1984. The common theme linking both efforts is the desire to support the powerful and elegant dataflow model of computation with as simple a hardware platform as possible.

The driving force behind the development of the Cydra 5 was the desire for increased performance over superminis on numerically intensive computations, but with the following constraint: The user should not have to discard the software, the set of algorithms, the training, or the techniques acquired over the years. As a result, the user would be able to move up in performance from the supermini to the minisuper in a transparent fashion. This transparency is important for a product such as the Cydra 5, which is aimed at the growth phase of the minisupercomputer market. Such a product must cater to a broader and less forgiving user group than the pioneers and early adopters who purchased first-generation minisupercom-

A version of this article appeared in *Proc. 22nd Hawaii Int'l Conf. on Systems Sciences*, Jan. 3-6, 1989, Kailua-Kona, Hawaii.

0018-9162/89/0100-0012$01.00 © 1989 IEEE

puters. Ideally, a departmental supercomputer will display none of the idiosyncrasies of typical supercomputers and minisupercomputers and, in fact, will project the "feel" of a conventional minicomputer, except for its much higher performance on numerically intensive tasks.

Cydra 5 system architecture

From the outset we were determined not to build an attached processor. The clumsiness of the host processor/attached processor approach leads to difficulty of use and loss of performance. The programmer must manage two computer systems, each with its own operating system and its own separate memory. The programmer must explicitly manage the movement of programs and data back and forth between the two systems. Since the data transfer path is slow relative to the processing speed of both computers, it becomes a performance bottleneck. Program development tools for the attached processor, such as compilers and linkers, run on the host. To avoid an unhealthy dependence on a single brand of host computer, this software must be maintained on multiple brands of host computer.

A self-sufficient, stand-alone computer has none of these problems. On the other hand, it assumes the burden of performing all of the general-purpose, nonnumeric work—such as networking, developing programs, and running the operating system—in addition to the numerically intensive jobs for which it was originally intended. As we will show later, the tradeoffs made in designing a supercomputer and a general-purpose processor are often diametrically opposed. As a result, each ends up being the most cost effective for a different class of jobs. Whereas a supercomputer may have 20 to 30 times the performance of a general-purpose processor on numerically intensive tasks, it may have only three or four times the performance on general-purpose tasks. When price is considered as well, the supercomputer ends up having poorer cost-performance than the general-purpose processor on nonnumeric tasks, since its expensive floating-point hardware is irrelevant.

We wanted the Cydra 5 to be not only a stand-alone computer but also a departmental supercomputer. By this we mean a number of things. It should be affordable to a small group or department of engineers or scientists, which means an entry-level price under $500,000. It should be easy to use by such a group and not require a "high priesthood" of skilled systems analysts catering to the idiosyncrasies of the machine. Lastly, it should be designed to handle *all* of the work load created by a department, not just the numerical tasks. As noted, this includes tasks such as compiling, text editing, executing the operating system kernel, and networking—tasks for which a supercomputer architecture is not cost effective.

These goals led to one of the key decisions regarding the Cydra 5: to have a numeric processor, highly optimized for numerical computing, and a tightly integrated general-purpose subsystem that would handle the nonnumeric work load. In other words the Cydra 5 was to be a heterogeneous multiprocessor, with each processor functionally specialized for a different, but complementary, class of jobs.

Initially, we planned to acquire a general-purpose processor on an original-equipment-manufacturer basis and integrate into it a numeric processor of our own design. We had determined that we needed about 10 million instructions per second of computation capability from the general-purpose processor to handle the I/O load imposed by the application running on the numeric processor, as well as the rest of the general-purpose work load. We soon discovered that a superminicomputer in this performance range would itself have a list price of about $500,000—the targeted price for the entire Cydra 5! We found consistently that lower priced general-purpose computation engines whose performance and price were closer to what we wanted had underdeveloped I/O capability by departmental supercomputer standards. This situation remains unchanged, if you examine the current crop of workstations and superworkstations. The only workable scheme that met both our cost and performance constraints was to design our own general-purpose subsystem consisting of multiple microprocessor-based processors.

Following a careful evaluation, we chose the as yet unannounced Motorola 68020 microprocessor. The various RISC (reduced instruction set computer) microprocessors were only on the drawing boards at the time. Around the 16-MHz 68020 we designed a fast interactive processor incorporating a 16-Kbyte, zero-wait-state cache. A scheme developed by two of the authors[4] maintained cache coherency in this multiprocessor environment.

We could not afford to develop an operating system from scratch, so we selected Unix, the only nonproprietary operating system available. Every workstation and minisupercomputer vendor has had to make the same choice for the same reason. As a result, Unix has become the de facto standard operating system for the engineering and scientific community. The more difficult choice was between the two competing flavors of Unix: Berkeley 4.2 and AT&T System V. Although Berkeley 4.2 was clearly dominant in 1984-85, we believed that with the addition of virtual memory and networking, and with AT&T's more aggressive support, System V would pull ahead by the time Cydra 5 was introduced. Accordingly, we took a deep breath and jumped on the System V bandwagon.

Although Cydrix, Cydrome's implementation of Unix, complies with the System V interface definition, it does contain a number of extensions, primarily for performance reasons. For use with a supercomputer, Unix is a rather low-performance, uniprocessor operating system. We rewrote the kernel significantly to symmetrically (not in a master-slave fashion) distribute it over multiple interactive processors so that one or more processors could be simultaneously executing in kernel mode. This allowed us to bring the aggregate computing capability of multiple processors to bear on the task of supporting the I/O for the numeric processor application. The file and I/O systems also received numerous enhancements.

As a consequence of this series of design decisions, the Cydra 5 Departmental Supercomputer is two computers in one: a numeric processor that is the functional equivalent of other minisupercomputers, and a general-purpose multiprocessor that plays the role of a front-end system (Figure 1). However, these two subsystems are very tightly integrated. They share the same memory system and peripherals and are managed by the same operating system. Because of this, the Cydra 5 with Cydrix presents the illusion of a simple uniprocessor system to the user who does not wish to be bothered with what is inside the black box.

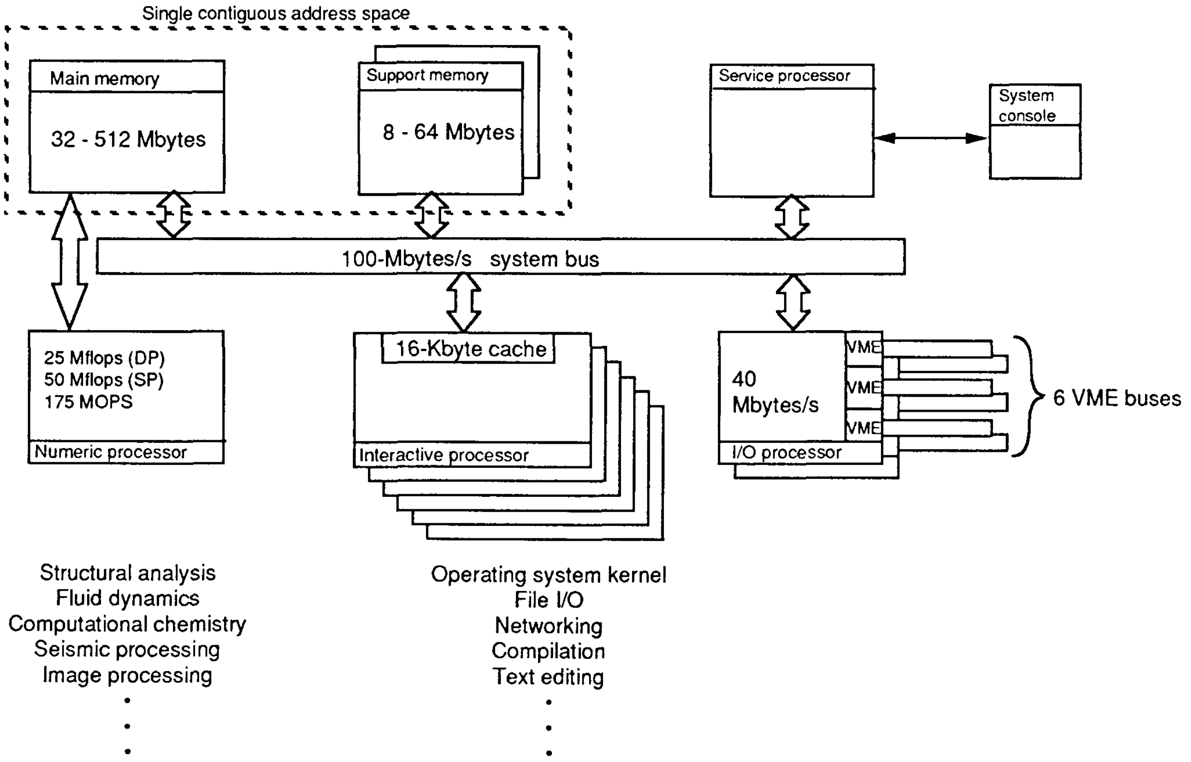


Figure 1. The Cydra 5 heterogeneous multiprocessor. The general-purpose subsystem consists of up to six interactive processors, up to 64 Mbytes of support memory, one or two I/O processors, and the service processor/system console connected over a 100-Mbyte/s system bus. Each I/O processor handles up to three VME buses, to which the peripheral controllers are attached. Also connected to the system bus, via a 100-Mbyte/s port, is the pseudorandomly interleaved main memory. The numeric processor has three dedicated ports into the main memory, each providing 100-Mbyte/s bandwidth. One of these is for instructions; the other two are for data. The main memory and support memory share a common address space and are both accessible from any processor.

The directed-dataflow architecture

Assuming the use of the fastest reasonable technology, any further increase in performance requires the effective exploitation of parallelism in one form or another.

Fine-grained versus coarse-grained parallelism. There are two major forms of parallelism: coarse-grained and fine-grained. Coarse-grained parallelism, popularly referred to as parallel processing, means multiple processes running on multiple processors in a cooperative fashion to perform the job of a single program. In contrast, fine-grained parallelism exists within a process at the level of the individual operations (such as adds and multiplies) that constitute the program. Vector, SIMD (single-instruction, multiple-data), and the attached-processor, or VLIW (very long instruction word), architectures are examples of architectures that use fine-grained parallelism.

Coarse-grained parallelism is complementary to fine-grained parallelism in that they can be used in conjunction. However, coarse-grained parallelism is not user transparent, since state-of-the-art compilers cannot, except in limited situations, take a sequential program written in a language such as Fortran and automatically partition it into multiple parallel processes. The user must explicitly restructure the program to capitalize on this type of parallelism. Since this did not satisfy our criteria for ease of use, we focused on the exploitation of fine-grained parallelism.

A bottom-up perspective. The final objective is to minimize the execution time of any given program. We can express this execution time T as

$$T = N \times C \times S$$

where N is the total number of instructions that must be executed, C is the average number of processor cycles per instruction, and S is the number of seconds per processor cycle. To a first approximation, N, C, and S are affected primarily by the compiler's optimization capability, the instruction set architecture, and the implementation technology, respectively. However, the picture is more complicated, and decisions that decrease one factor may end up increasing another.

The techniques used to minimize T have been many and varied. For general-purpose processors, the traditional approach was to reduce N at the expense of a smaller increase in C. The general thrust was to better utilize the micro-parallelism present in horizontally microcoded machines by defining more complex instructions with more internal micro-parallelism in the hope that N would decrease more sharply than C would increase. This approach is now termed CISC (complex instruction set computer). By contrast, the RISC approach focuses on the use of very simple, hardwired, pipelined instructions exclusively to reduce C and S. The resulting increase in N is minimized by the use of code optimization techniques in the compiler for an overall reduction in T. Although both approaches have been successful at different times and under different circumstances, they are not sufficient to meet the performance objectives of supercomputers.

The emphasis in supercomputers is on execution of arithmetic (particularly floating-point) operations. The starting point for all supercomputer architectures is multiple, pipelined, floating-point functional units, in addition to any needed for integer operations and memory accesses. The fundamental objective is to keep them as busy as possible. Assuming this will be achieved, the hardware must be equipped to provide two input operands per functional unit and to accept one result per functional unit per cycle. Furthermore, since the results of one operation will be required as inputs to subsequent operations, some form of interconnection is needed between the result and input buses. Finally, since results are not always used immediately after generation, storage in the form of one or more register files is needed. Figure 2a shows the data paths of a generic supercomputer. The details, of course, vary from one machine to the next; the number and types of functional units, the number of register files, and the structure of the interconnect can all be different.

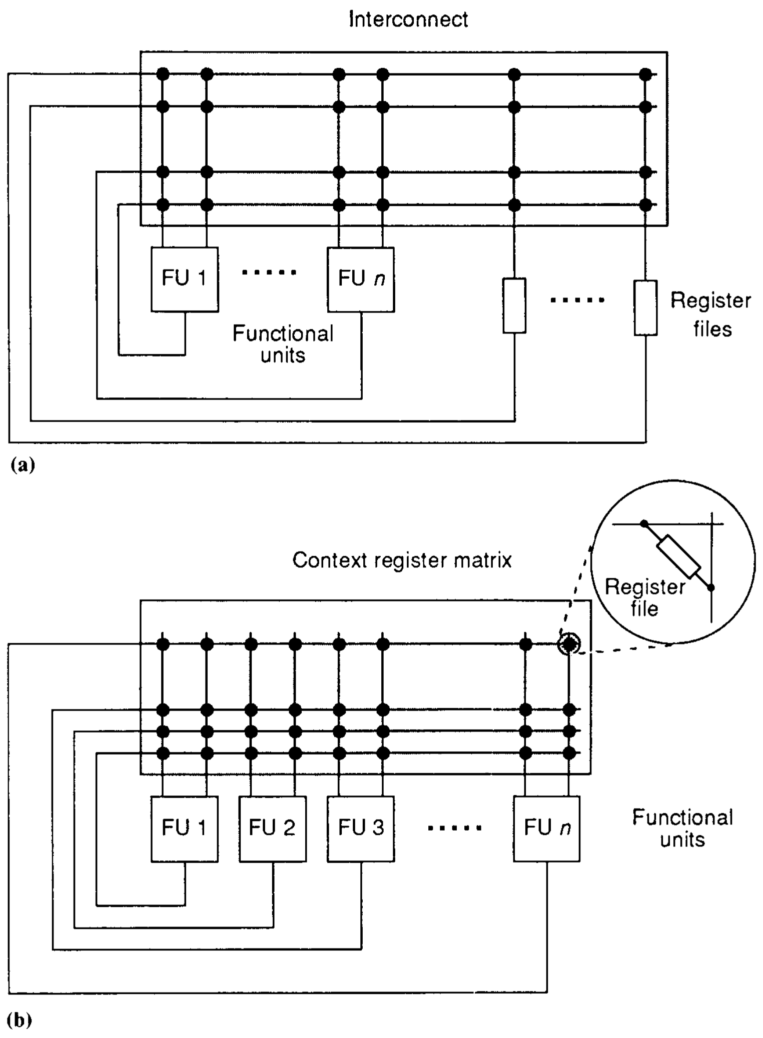


Figure 2. (a) Generic supercomputer data paths. (b) Generic directed-dataflow-processor data paths. Each row of cross-point register files in the context register matrix can be written into by only a single functional unit, and all register files in a single row have identical contents. Since each of the register files in a row can be read in parallel, each row is logically equivalent to a single multiported register file capable of one write and multiple reads per cycle. Each of the cross-point register files can be written to by only a single functional unit and can be read by only a single input of a single functional unit—the one associated with that column of cross-point register files. This, along with the property that each register file is capable of one read and one write each cycle, guarantees conflict-free access to the context registers for every functional unit for inputs as well as outputs.

Interesting and rather fundamental differences exist between the data paths of a scalar processor and those of a supercomputer. In the scalar processor the critical data paths consist of the circuit from the general-purpose registers (GPRs), through the integer arithmetic-logic unit or the cache, and back to the GPRs. This makes it relatively easy to keep the physical distances small, the pipelining moderate, and the cycle time short. In contrast, critical data paths in a supercomputer must include multiple floating-point pipelines, large numbers of register files, and a complex interconnect. The physical distances are necessarily larger, and there are more electrical loads to be driven on each bus. Both factors cause a larger fraction of the cycle to be consumed in merely transferring data from one point to another. This makes it necessary to increase the depth of pipelining to avoid compromising the cycle time. Thus the trade-off is short pipeline latencies and better sequential performance versus multiple, deep pipelines and better parallel performance.

All the hardware in a supercomputer can be justified only if it is kept well utilized. But keeping all these pipelines busy requires that multiple operations be issued (as opposed to being in execution) at every cycle. This is impossible in conventional scalar architectures, since a maximum of one instruction is issued per cycle.

Two styles of uniprocessor architecture have been developed to circumvent this bottleneck. One is the vector architecture, which attempts to reduce N by the use of very complex instructions—vector instructions—where a single vector instruction does the work of multiple, identical scalar operations. Once a few vector operations have been launched, multiple operations are issued per cycle, one each by every vector operation that is active. This is philosophically akin to the CISC approach and suffers from the standard problem of complex instructions: They work well when they exactly fit the job that must be done, but they are useless if the task differs even slightly.

The second, more flexible, approach is an architecture in which multiple operations can be issued in a single instruction.[2,3,5-7] We can view this as an extension of the SIMD architecture; instead of the same opcode's being applied to all the pairs of input data, as in SIMD, distinct opcodes are used. The formal name for such an architecture might be single instruction, multiple operation, multiple data (SIMOMD), or MultiOp for short. Other terms used for such an architecture include horizontal[3] or very long instruction word (VLIW).[5] We can also view this as a generalization of the RISC architecture, where C is reduced to less than 1 by issuing multiple operations (or multiple instructions, from a scalar processor point of view) per cycle.

When we started in 1984, MultiOp execution was used in a number of attached-processor products—for example, the FPS-164.[6] Expert programmers were able to coax much better sustained performance out of these products than could be achieved with vector processors having the same peak performance. But programming these products for high performance had all the complexities associated with microprogramming, and some more besides.[3,8] The problem resulted from the processors' having been designed entirely from a bottom-up, hardware perspective, with no thought to how they were to be programmed or the obstacles the compiler writer would face. While we felt it constituted a step in the right direction, it was clear to us that the attached-processor architecture was unacceptable. Although the general hardware structure of the directed-dataflow architecture was dictated by the considerations discussed above, the subtler aspects resulted from a top-down thought process driven by our model of computation.

A supercomputer architecture must do well on all types of numerical computation it might encounter.

The model of computation. Although there is a tendency to categorize architectures superficially by their hardware attributes (pipelined or VLIW, for example), we believe that the underlying models of computation and usage are what really matter. They determine the context within which all of the design decisions are made, and they lead to architectural features that, though subtle, have a major impact on the performance and breadth of a product's applicability. Just as pipelining has been applied to a number of quite different architectures, so too we find that the ability to issue multiple operations per cycle has been used in at least three types of architectures with quite distinct underlying models of computation: the microprogrammed attached-processor architectures, the VLIW architecture, and the directed-dataflow architecture.

The attached processors borrowed their model of computation from microprogramming, where the perspective is very bottom-up. In microprogramming, the hardware is viewed as a collection of functional units and buses, and the task of the microprogrammer is to determine on a cycle-by-cycle basis which functional unit inputs and outputs connect to which buses. The microprogram is generally viewed as a sequence of micro-operations, and any parallelism needed to exploit a horizontal microarchitecture is exposed on a localized, "peephole" basis. Microprograms for instruction set interpretation put very little emphasis on iterative constructs (loops) but a lot on branching. When an architecture with these underlying assumptions is adopted for numerical processing (where the emphasis is on loops), a great deal of programming complexity results if one wishes to fully exploit the opportunities for parallelism.[3]

Although the VLIW school of thought, too, has its roots in microprogramming, the VLIW architecture is properly viewed as the logical extension to the scalar RISC architecture. The underlying model is one of scalar code to be executed, but with more than one operation issued per cycle. The obstacle to good performance is the high frequency of branches in typical programs, a problem common to microprogramming. Consequently, trace scheduling,[9] originally developed for microprogramming, is used with VLIW processors. No special consideration—beyond the standard scalar processing compiler technique of loop unrolling—is given to loops in software, and none whatsoever is given in hardware. Using trace-scheduling techniques, VLIW can provide good speedup on scalar code, and when loop unrolling is used, some further speedup on iterative computations occurs as well. However, given the lack of architectural emphasis on loops, VLIW does not do as well on vectorizable computations as a vector processor does.

A supercomputer architecture must do well on all types of numerical computation it might encounter. This includes straight-line or sequential code as well as branching scalar code; but clearly of equal or greater importance are the iterative constructs that constitute the heart of numerical computations. Loops, whether they are vectorizable or of the type that contains recurrences, conditional branching, or irregular references to memory, must get explicit support in the hardware. The vector architecture provides hardware support, but only for a limited subset of loops (those without recurrences and conditional branches), reflecting a restrictive model of computation. If greater generality is required in the capabilities of the architecture, a more general model of computation is needed, one that does well not only on scalar and vector code like the VLIW and vector architectures, respectively, but also on the important class of loops that possess parallelism but are not vectorizable.

The dataflow model of computation. By making execution of an operation contingent only on its inputs' being available and a functional unit's being free to execute the operation, the dataflow model of computation exposes and exploits every bit of parallelism in the computation regardless of its form.[10] Thus it provides an excellent basis for a fine-grained parallel architecture of broad applicability.

However, most dataflow research assumes that the program is written in a dataflow or functional language, whereas we had to face the real world of Fortran programs. Consequently, we had to base our architecture on an extended dependency graph model that included the concept of memory and the corresponding memory read and write operations. This, in turn, required that the model include the concept of antidependency and output dependency,[11] in addition to that of data dependency. Also, to reflect the rather unstructured nature of Fortran control constructs, we had to include a richer control dependency model that went beyond nested IF-THEN-ELSEs. Nevertheless, the basic philosophy was unchanged; an operation is a candidate for execution as soon as all its incoming dependencies, of all types, have been satisfied. Notwithstanding the differences between the dataflow model of computation and our dependency graph model, we will, for the sake of convenience, refer to our model of computation as a dataflow model.

The dataflow model has the richness needed to yield parallelism on both scalar and iterative computation. In scalar code, parallelism is restricted by the presence of conditional branches and their associated control dependencies. Dataflow gets around this problem with a mode of operation known as *eager execution*, which causes operations to be executed before it is certain that their execution is needed (Figures 3a-3f). This is equivalent to selectively removing certain control dependence arcs to increase the parallelism.

In iterative computations, dataflow dynamically unrolls a loop the same number of times that the loop was supposed to be executed.[10] This generates the maximum parallelism possible, limited only by the inherent dependencies in the computation. (The amount of this parallelism actually used depends on the number of functional units present.) The dataflow architecture's ability to exploit whatever parallelism exists in all of these constructs makes it the architecture best able to exploit all of the fine-grained parallelism existing in programs. Furthermore, since parallelism is achieved without having to make any algorithmic changes to the program, the dataflow architecture delivers performance increases transparently. For these reasons the dataflow architecture serves as the basis for the Cydra 5.

The dataflow model has the richness needed to yield parallelism on both scalar and iterative computation.

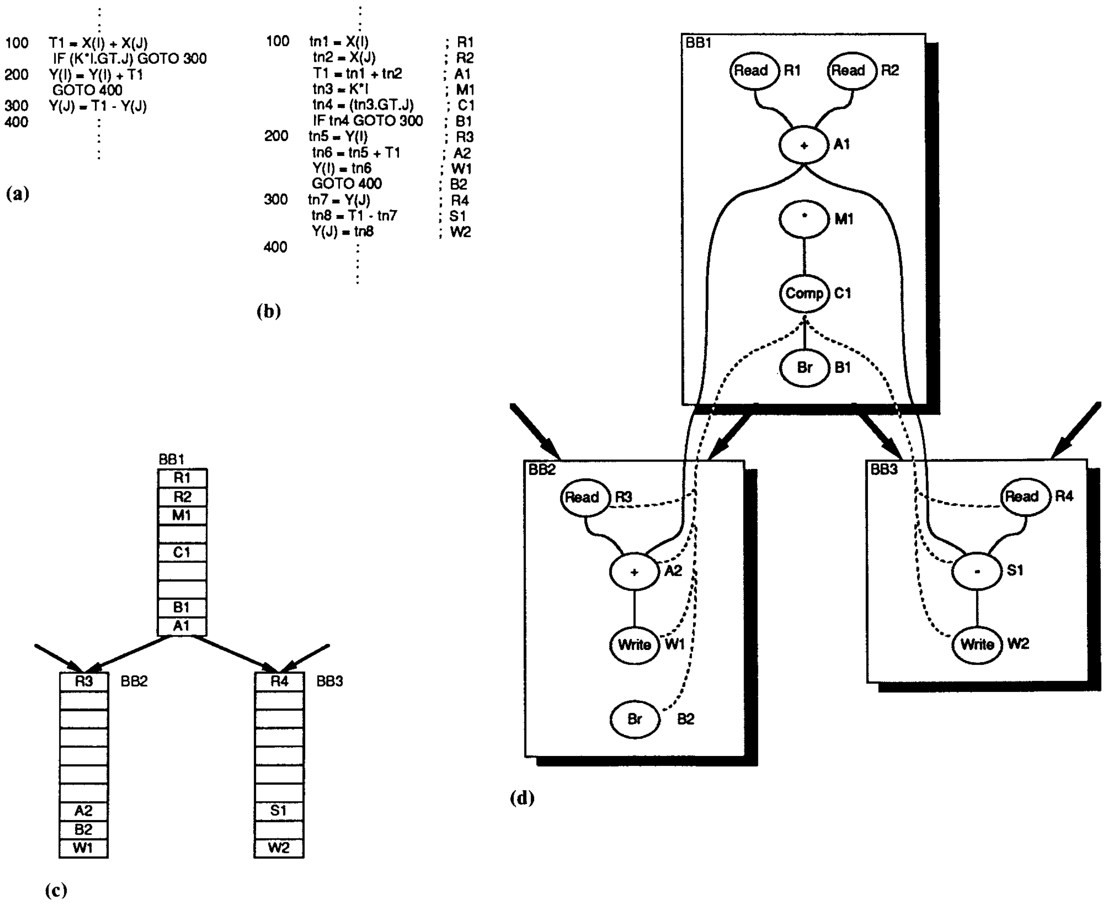
Directed dataflow. The directed-dataflow architecture is also significantly influenced by another philosophy, one of moving complexity and functionality out of hardware and into software whenever possible; this is the cornerstone of the RISC concept as well. The benefits of this philosophy are reduced hardware cost and often the ability to make better decisions at compile time than can be made at runtime. In the directed-dataflow architecture, the compiler makes most decisions regarding the scheduling of operations at compile time rather than at runtime—but with the objective of emulating faithfully the manner in which a hypothetical dataflow machine with the same number of functional units would execute a particular program.

The compiler takes a program and first creates the corresponding dataflow graph. It then enforces the rules of dataflow execution, with full knowledge of the execution latency of each operation, to produce a schedule that indicates exactly when and where each operation will be performed. While scheduling at compile time, the compiler can examine the whole program, in effect looking forward into the execution. It thus creates a better schedule than might have been possible with runtime scheduling. An instruction for the directed-dataflow machine consists of a time slice out of this schedule, that is, all operations that the schedule specifies for initiation at the same time. Such an instruction causes multiple operations to be issued in a single instruction.

So far, this is the same as any other VLIW processor. However, more than the ability to issue multiple operations per cycle is needed to efficiently support the dataflow model of computation in a compiler-directed fashion, especially when executing loops. Specifically, the directed-dataflow architecture provides two architectural features: the context register matrix and conditional scheduling control.

The context register matrix. Unlike the generic structure shown in Figure 2a, a directed-dataflow machine combines the register storage and the interconnect between functional unit outputs and inputs into a single entity known as the context register matrix, as shown in Figure 2b. In general the interconnect structure can be viewed as a sparse crossbar with certain cross-points absent but with a register file at each cross-point present. The context register matrix guarantees conflict-free access to the context registers for every functional unit. This in turn guarantees that once a schedule has been prepared by the compiler, it will not be rendered infeasible because of contention in getting data into or out of the context registers, which is one of the fundamental problems with the attached-processor architectures.[3]

When executing loops in a maximally parallel dataflow fashion, each iteration is viewed as a distinct computation executing in parallel with the other iterations. As with similar situations where the same code is being executed by distinct parallel computations, each iteration must have its own context so that when two iterations apparently refer to the same variable or, in this case, the same register (since they are both executing the same code), the physical locations actually accessed are distinct. When concurrent processes are forked, this is achieved by providing each process with a duplicate name space. In the case of recursive invocations of the same procedure, this is handled by providing separate stack frames. Each invocation's reference to a particular local variable is in the context of its own stack frame. Like-



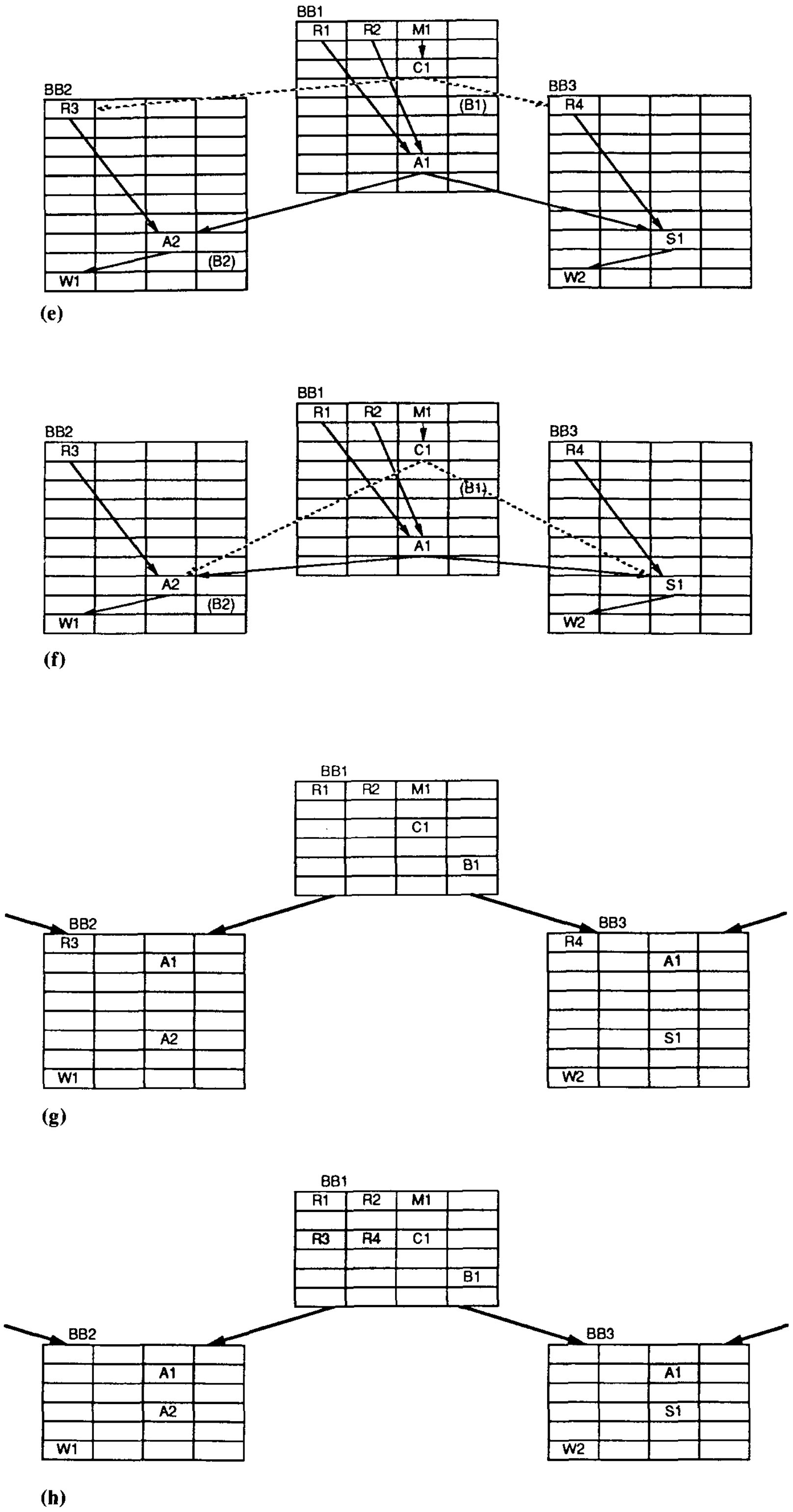
```
             :
100   T1 = X(I) + X(J)
      IF (K*I.GT.J) GOTO 300
200   Y(I) = Y(I) + T1
      GOTO 400
300   Y(J) = T1 - Y(J)
400          :
             :
```

(a)

```
             :
100   tn1 = X(I)             ; R1
      tn2 = X(J)             ; R2
      T1 = tn1 + tn2         ; A1
      tn3 = K*I              ; M1
      tn4 = (tn3.GT.J)       ; C1
      IF tn4 GOTO 300        ; B1
200   tn5 = Y(I)             ; R3
      tn6 = tn5 + T1         ; A2
      Y(I) = tn6             ; W1
      GOTO 400               ; B2
300   tn7 = Y(J)             ; R4
      tn8 = T1 - tn7         ; S1
      Y(J) = tn8             ; W2
400          :
             :
```

(b)

Figure 3. (a) A fragment of Fortran code. (b) Expansion of the code into individual operations with a label on the right-hand side for each operation. The scalar variables *K*, *I*, and *J* are assumed to be in registers already. (c) Sequential code schedule assuming a seven-cycle latency for memory reads and a two-cycle latency for all other operations. The code fragment consists of three basic blocks. BB2 and BB3 can be entered from BB1 as well as from elsewhere. A traversal of this code fragment takes 19 cycles. Note the scheduling of operations in the delay slots of the delayed branch. (d) The dataflow graph for this code fragment. Solid arcs are data dependencies. Dashed arcs are control dependencies that enable or disable operations, depending on the Boolean result of the comparison C1. Note that branch operations have no role in the dataflow model of computation. (e) Schedules resulting from the execution of the dataflow graph (assuming the ability to initiate two memory operations and one other operation per cycle). Operation A1 from BB1 is initiated in parallel with the execution of either BB2 or BB3. The extent of the overlap between the execution of BB1 with the execution of either BB2 or BB3 is determined by the control dependency from C1 to R3 or R4, which determines whether BB2 or BB3 should be executed. (f) Eager execution of R3 or R4 results from the removal of the control dependency from C1. Now both R3 and R4 are initiated before it has been determined whether BB2 or BB3 is to be executed. As a result the total execution time is reduced. (g) Directed-dataflow code that achieves the same effect as in (e). Operation A1 has been moved from BB1 into both BB2 and BB3. However, to preserve the original semantics, it should be executed only if BB2 or BB3 was entered from BB1. Therefore, both copies of A1 have the predicate corresponding to BB1 even though they are in BB2 and BB3. (h) Directed-dataflow code that performs the eager execution of R3 and R4 by moving them up into BB1. (They must also be copied into all basic blocks from which they can be entered.) The total execution time for this fragment of code is now 12 cycles.



wise, the maximally parallel execution of loop iterations requires that each iteration's register references be within the context of the corresponding *iteration frame*, that is, a set of registers allocated to a particular iteration.

The directed-dataflow architecture has the architectural facilities needed to dynamically allocate iteration frames at runtime and the requisite addressing capabilities to reference registers in both the current iteration frame and, in the case of recurrences, in previous iteration frames. Surprisingly, these architectural facilities incur only a modest hardware cost, namely the ability to reference the context registers with an instruction-specified displacement from a base register containing the *iteration frame pointer* (IFP). Since the IFP is decremented each time a new iteration is initiated, each iteration of the loop accesses a distinct set of physical registers. Any result computed during the same or a previous iteration can be accessed by using the appropriate displacement from the current value of the IFP. This displacement can be computed by the compiler, since it knows the difference between the current value of the IFP and the value at the time the result was generated, as well as the original displacement from that value of the IFP. Some interesting register allocation techniques in the compiler are central to the efficient use of the context registers, but they are beyond the scope of this article and will be reported elsewhere.

Conditional scheduling control. In a sequential model of computation, the program consists of a set of basic blocks, each containing a list of instructions. Only one basic block is active at any one time. The equivalent dataflow view is that a program consists of a set of basic blocks, each consisting of a dependency graph of operations executed in a parallel fashion. Conceptually, any given operation has two types of incoming dependencies: the data (input operands) dependencies, which determine *when* the operation can be issued, and a control dependency from an operation—in another basic block—that computes the predicate. The predicate determines whether the operations in a basic block are to be issued at all. The predicate is true and the operations are issued if and only if control would have flowed to that basic block in the corresponding sequential program.

Since an operation can be executed as soon as its data and control dependencies

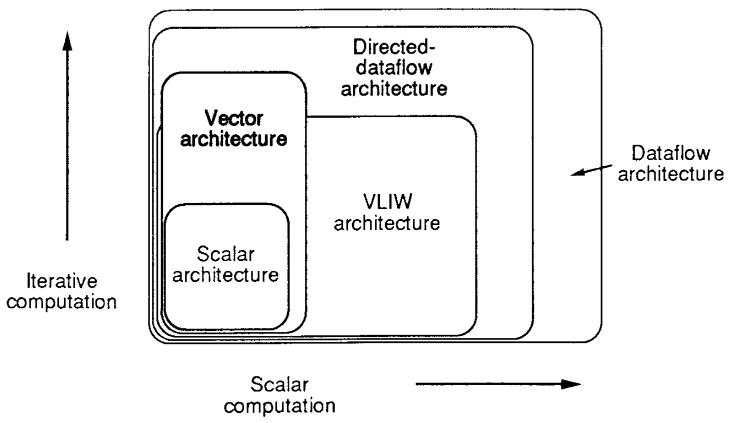


Figure 4. Relationship between various uniprocessor architectures. On vectorizable loops, directed-dataflow is slightly better than the vector architecture, which is better than VLIW, which in turn is better than the scalar architecture. On sequential code, directed dataflow is at least as good as VLIW, which is better than both the vector and scalar architectures. On nonvectorizable loops, directed dataflow is significantly better than all three architectures.

have been satisfied, it is possible to have multiple basic blocks active at the same time (Figures 3e and 3f), particularly in the case of loops, whether they have conditional branching within the body of the loop or not. This generates the desired parallelism but is possible only because dataflow can have multiple loci of control active simultaneously. The directed-dataflow architecture has the goal of achieving the same effect as dataflow, but with a single locus of control. This is achieved by including in each basic block operations from other basic blocks that should be executing in parallel (Figures 3g and 3h). This is a form of code motion, in this case for enhancing the parallelism in the program.

An explicit predicate is unnecessary in the sequential model of computation, since all operations in a single basic block, by definition, have the same predicate. This predicate is implied by the fact that the program branched to this basic block; that is, it decided that this basic block was to be executed. Thus the predicate's being true and the flow of control's arriving at the basic block are synonymous.

But these two concepts must be decoupled in the case of directed dataflow, since a basic block can contain operations that in the sequential program would have been in another basic block, and thus under a different predicate. Consequently, each operation is provided with a third input—the predicate—in addition to the two normal ones. An operation is issued only if control flows to its basic block and the predicate is true. The predicate input specifier specifies a register in a Boolean register file, which, for historical reasons, is termed the iteration control register (ICR) file. Boolean values, which result from compare operations, may be transferred into the ICR. In loops, each iteration generates predicates corresponding to the conditional branches (including the loop exit conditional) within the loop body. Therefore, the ICR too must support the capability for allocating iteration frames.

With hardware support in the form of the context register matrix and conditional scheduling control, the compiler can generate code for the directed-dataflow machine that retains the parallelism of the dataflow architecture. At the same time, it capitalizes on the efficiencies of moving scheduling from runtime to compile time. It is this architectural support for the dataflow model of computation that sets directed dataflow apart from other VLIW architectures.

Comparison with other fine-grained architectures. Figure 4 shows the relationship between various architectures that exploit fine-grained parallelism. It uses two criteria: performance on iterative computations and performance on scalar code. At opposite ends of the spectrum are the (dynamically scheduled) dataflow architecture, which can exploit all the parallelism, and the scalar architecture, which exploits none of it. The vector architecture is better than the scalar on the restricted class of vectorizable computations, but no better on scalar code. The VLIW architecture does better than the scalar architecture on scalar code and much better on iterative computations, but it does not do as well as the vector architecture on vectorizable loops. This is because the loop-unrolling techniques that it uses can yield only short vector performance levels. On scalar codes the directed-dataflow architecture is at least as good as the VLIW architecture and better than the vector architecture. It is slightly better than the vector architecture on vectorizable loops, since strip mining (Figure 5) is unnecessary, and it is far better on nonvectorizable loops. Directed dataflow, as a result of its architectural support for dataflow, is better than the VLIW architecture on iterative computations.

Numeric processor decisions and trade-offs

Technology selection. The choice of implementation technology was perhaps the most important decision we had to make, since it fundamentally affected all other decisions and trade-offs. Our back-of-the-envelope calculations indicated that the same architecture could be implemented in TTL/CMOS at two-fifths the performance of an emitter-coupled logic (ECL) implementation and two-thirds the cost (a 100-nanosecond rather than a 40-nanosecond cycle). If cost and performance had been the only considerations, it would have been a simple decision, since our corporate objective was to serve the high end of the departmental supercomputer market. However, we realized that opting for an ECL implementation would preclude the use of VLSI floating-point chips such as the Weitek chips and that it would require the design of more logic and more boards, as well as more development time—hence, more nonrecurring expenditures.

Also, some people at that time believed ECL was a doomed technology, that CMOS would catch up in performance—and at a fraction of the cost. Although this belief was, and still is, incorrect, there was tremendous pressure from the investment community to build a TTL/CMOS product and even to discard some of the functionality and build a subset of what we were planning. However, we were convinced that the low end of the market would be very crowded with minisupercomputers and array processor products. (This has, in fact, come to pass, except that superworkstations have replaced array processors as the threat at the low end.) We wanted to be above the general melee, so we decided to stick with our business plan and implement an ECL product. We now feel vindicated in that decision, since more and more computer vendors are currently moving to this technology. At a 40-nanosecond cycle time, this yielded a processor with a peak performance of 25 million floating-point operations per second (Mflops) with 64-bit operands, 50 Mflops with 32-bit operands, and 175 million operations per second overall.

In the interest of reducing the nonrecurring development costs and the development risk, we made another important decision: By and large, we would use off-the-shelf ECL components. Gate arrays would be used only in certain performance-critical parts of the design, and even so, not for control logic. Since this was the first implementation of a directed-dataflow processor, the decision, even in retrospect, was correct. But it had many unpleasant consequences. The 1985 vintage of standard ECL logic was at a very low level of integration. This meant that the numeric processor would occupy a lot of real estate. Since manufacturing considerations limited us to boards of roughly 18 inches on a side, the numeric processor would have to be spread out over a large number of boards. This led to a snowballing effect; large amounts of buffer logic were required to drive signals between the many boards. This in turn increased the total amount of logic even further. If we had had the option of implementing the numeric processor entirely with gate arrays, we could have reduced its size by a factor of three or four.

Number of functional units. Our performance goal was to be able to initiate one floating-point add and one floating-point multiply every 40-nanosecond cycle, using two separate pipelined functional units. Additional functional units were required to support these two floating-point pipelines. The first issue was the number of ports to memory that were needed. Our thinking here was influenced by the fact that our model of computation was dataflow and not vectors. We viewed the entire body of the loop as a single entity rather than as a number of separate vector operations. Thus we required memory reads and writes only for the array inputs and outputs, not for the scalar temporaries, which on a vector machine would be converted to vector temporaries. This reduced the relative number of memory operations needed on our machine (Figure 5).

Our program statistics indicated that within innermost loops the balance between memory bandwidth and floating-point computation capability was between one and three memory operations per pair of floating-point operations. Although certain important computations such as SAXPY (the addition of one vector to another one that has been multiplied by a scalar) require one and a half memory operations per floating-point operation, we felt that this was an expensive luxury in hardware and that the need was not statistically frequent enough to warrant the expense. Also, half a memory operation per floating-point operation seemed entirely inadequate. So, given the existence of two floating-point functional units, we decided to have two ports to memory. Since it is necessary to compute an address for each memory operation, typically to increment an index into an array, this implied the presence of two address (unsigned integer) adders. Also, to facilitate dope vector calculations for random references into multidimensional arrays, we provided an address multiplier.

Within loops there are also, typically, a certain number of integer operations. Since we did not wish to have these operations steal cycles from the floating-point units, we added an integer functional unit. This left us with a grand total of eight pipelined functional units. A subsequent crisis caused by the burgeoning amount of logic required a reduction in the size of the numeric processor. As a result of this exercise, we eliminated the integer unit and the address multiply unit as separate pipelines and merged them with the floating-point adder and address adder units, respectively.

Very early in the project, we constructed a prototype scheduler that would generate schedules for programs written in a low-

```
DO I = 1, N
  Q = U(I) * Y(I)
  Y(I) = X(I) - Q
  X(I) = Q + V(I) * X(I)
ENDDO
```

(a)

```
DO I = 1, N
  Q = U(I) * Y(I)
  Y(I) = X(I) - Q
  T = V(I) * X(I)
  X(I) = Q + T
ENDDO
```

(b)

```
DO I = 1, N, 64
  DO J = I, I+63
    Q = U(I) * Y(I)
    Y(I) = X(I) - Q
    T = V(I) * X(I)
    X(I) = Q + T
  ENDDO
ENDDO
```

(c)

Figure 5. (a) Fortran code for a loop. (b) Fortran loop rewritten with just one floating-point operation per statement. Each statement would become a vector floating-point operation. Using registers in a scalar machine would result in only six memory operations (four reads and two writes) per iteration. A memory-to-memory vector processor would require 12 memory operations per iteration (two reads and one write per statement). (c) Using vector registers can bring the memory operations back down to six per iteration, but because of the finite length of the vector register (assumed to be 64 in this example), strip mining must be used. The loop is transformed into a doubly nested loop. The inner loop handles 64 long chunks of the vector; the outer loop sequences through all chunks that make up the vector. (For simplicity, *N* is assumed to be a multiple of 64.) Strip mining limits the length of vector operations to no more than the vector register length, thus reducing performance. In the directed-dataflow architecture, since registers are continuously deallocated from iterations that have completed and allocated to new iterations, the number of memory operations per iteration is six, yet there is no restriction on the length of the vector operation.

level dependency graph language. This scheduler worked in a table-driven manner, using for this purpose a machine description file. By modifying this machine description file, we were able to estimate the relative performance of the numeric processor while varying the number of pipelines, the depth of the pipelines, and the assignment of opcodes to functional units. One of the alternatives we experimented with was the number of floating-point functional units, bearing in mind that computational balance required a memory port and an address unit for each floating-point unit. We found that the increase in performance with the number of floating-point units was quite sublinear, while the increase in cost and complexity was most definitely superlinear. We opted, therefore, to stay with two floating-point units and to design them to run as fast as possible instead of providing many slow units. Given a certain target performance level, the compiler needs to find less parallelism in the program in a processor with a few fast units than in one with many slow units.

At this point we had six pipelined functional units (including the two memory ports), each requiring two input operands and generating a result. Complete connectivity between all outputs and all inputs would have required a 6 × 12 crossbar with a register file at each cross-point. This was infeasible from an implementation viewpoint. And yet, with our understanding of the problems of programming Floating Point Systems' AP-120B, we were unwilling to eliminate cross-points in an ad hoc manner. Some underlying scheme having conceptual integrity and relevance from the viewpoint of the compiler writer was essential.

We partitioned the functional units on the basis of data versus addresses, placing the two floating-point units, the integer unit, and the two memory ports in the data cluster and the remaining functional units in the address cluster. This immediately reduced the number of cross-points by half. The final structure of the numeric processor's data paths is shown in Figure 6.

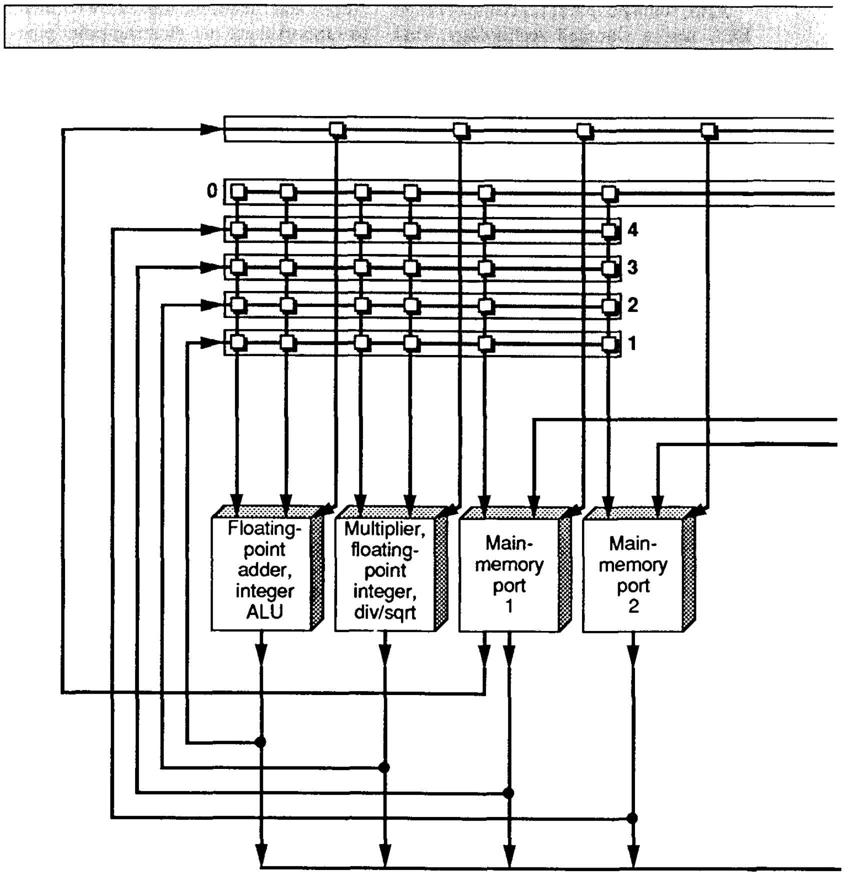
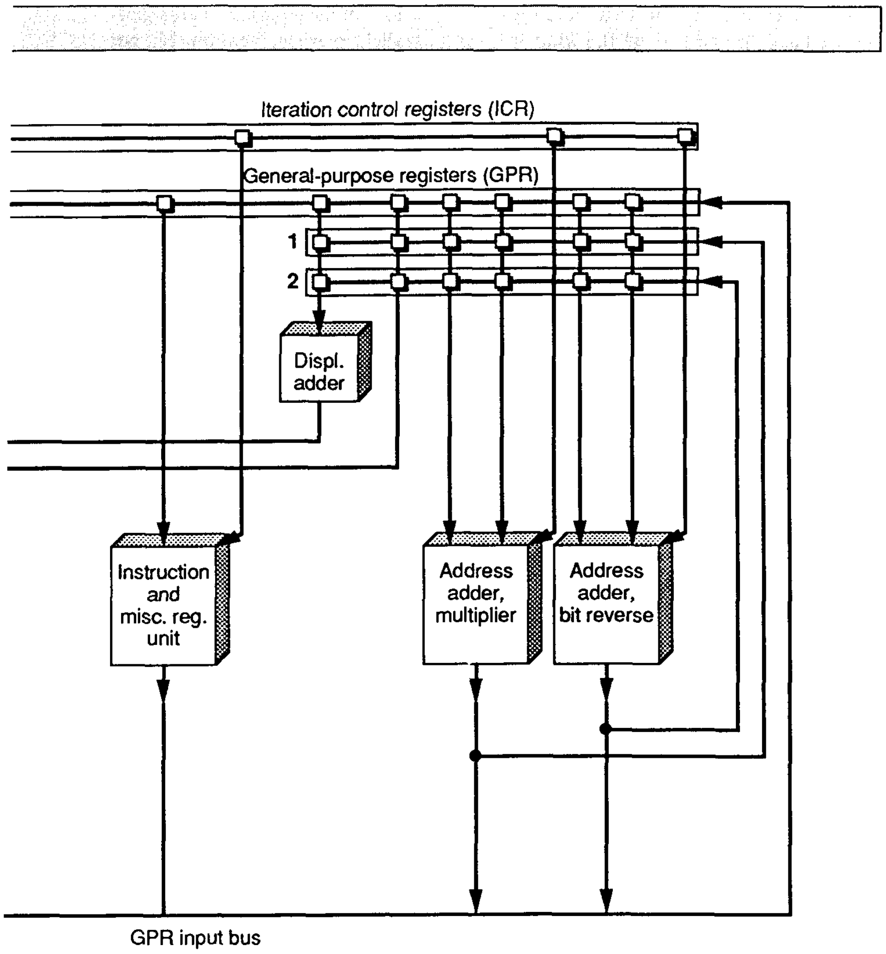

0
4
3
2
1
Floating-point adder, integer ALU
Multiplier, floating-point integer, div/sqrt
Main-memory port 1
Main-memory port 2

Figure 6. Major numeric processor data paths. The numeric processor E-unit contains two major parts: the data cluster and the address cluster. The data cluster consists of four functional-unit pipelines interconnected by the data context register matrix. These four pipelines are (1) the floating-point adder/integer ALU (four-cycle latency), (2) the floating-point/integer multiplier (five-cycle latency) and divider, as well as the square-root unit, (3) memory data port 1 (17-cycle latency), and (4) memory data port 2 (17-cycle latency). The address cluster consists of two address adder pipelines (three-cycle latency) interconnected by the address context register matrix. In addition, the first pipeline provides a bit-reverse capability, while the second provides an integer multiply capability. The context register

Another major constraint, especially in the context of an interconnection-rich architecture such as this, was the amount of board I/O. Obviously, the choice of data path width had a major impact on the number of signals crossing board boundaries. With 32-bit data paths we found ourselves up against a wall even with the use of fairly aggressive connector technology. A move to 64-bit data paths would have required the use of exotic and expensive connectors. The rule of thumb we had developed indicated that 64-bit data paths instead of 32-bit data paths would increase 64-bit performance by 50 percent and 32-bit performance not at all. Discretion being the better part of valor, we elected to use 32-bit data paths.

Register storage. The context register matrix provides the architectural facilities needed to dynamically allocate iteration frames at runtime. Since each iteration initiated is allocated an iteration frame, and since the total number of registers in the context register matrix is fixed, the iteration frames for past iterations must be deallocated at the same rate that new ones


Iteration control registers (ICR)
General-purpose registers (GPR)
1
2
Displ. adder
Instruction and misc. reg. unit
Address adder, multiplier
Address adder, bit reverse
GPR input bus

matrix (CRM) provides simultaneous, conflict-free access to all functional-unit inputs and outputs. Also, by virtue of the iteration frame pointer relative addressing into it, the CRM supports overlapped execution of loops. Each functional unit has three inputs. Two of these are the conventional operand inputs sourced by either the CRM or the general-purpose registers (GPR). The third input is a Boolean value from the iteration control register (ICR) used to conditionally control the issuance of operations. The output of each functional unit can go either to its row in the CRM or to the GPR. Like the CRM, the GPR and ICR consist of as many carbon-copy register files as there are inputs that can source them.

are allocated. While this is exactly what is desired for loop variants (values computed by each iteration), it poses a problem for loop-invariant values that are used, but never computed, within the loop. Unless they are continuously copied from one iteration frame to the next, they will be overwritten. To avoid the significant overhead of copying loop invariants, we provided a register file that is global to all iterations and does not possess the iteration frame capabilities. With considerable originality, we called this the *general-purpose register* file. In a single cycle it can be read by any number of functional unit input ports simultaneously and at distinct locations, just like a row in the context register matrix, and it can be written to by the output port of any functional unit, but only one at a time.

One of the more ad hoc decisions we made was choosing the number of registers per register file. The problem we faced was a lack of directly relevant statistics on the effect of this parameter on performance in the context of a directed-dataflow style of execution. So we plucked the decision out of thin air. Our collective intuition told us that 32 registers per register file was too few, and 128 registers looked difficult from an implementation viewpoint. Thus we settled on 64 registers per register file. Comforted by the lack of alternatives, we moved on.

The capacity of the iteration control register (ICR) was determined by two opposing considerations. Since each predicate would be used as input to operations scheduled to execute at times separated by rather long intervals, we expected the lifetime of these values to be long. This implied more capacity in the ICR than in the register files in the context register matrix. On the other hand, the number of bits available in the instruction format to address the ICR was at a premium. We finally decided to provide an ICR capacity of 128.

Opcode repertoire. The basic philosophy in the directed-dataflow architecture is to work with atomic operations that can be scheduled with maximum flexibility. So, we have no operations of the CISC type that read from memory (two operands), perform an operation, and write the result back to memory. To our way of thinking, this is actually four different operations packaged together, usurping the compiler's ability to achieve optimal scheduling. Except for the memory Read and Write opcode class, no other opcodes access memory. Their inputs and outputs are predominantly either the context register matrix or the GPR file. Also, with a few exceptions, the opcode repertoire is the normal set of integer, logical, floating-point, and memory operations. The few exceptions relate to supporting the directed-dataflow model of computation. The opcode repertoire reflects the numerical bias; while there is extensive support for floating-point operations, there is none for binary-coded decimal arithmetic or string operations. Except in the case of (bitwise) logical operations, where it would be redundant, opcodes are provided for both single-precision (32-bit) and double-precision (64-bit) operations. The data types supported by the execution unit hardware are 32-bit and 64-bit IEEE floating point, 32-bit and 64-bit 2's complement integers, 32-bit unsigned in the address cluster, and 32-bit logical.

The data paths, as well as the registers in the numeric processor, are 32 bits wide. We believed that register allocation in the compiler would be simplified if the two halves of a 64-bit datum could be independently assigned to unrelated registers. This meant we would need four source-register specifiers and two destination-register specifiers for a 64-bit operation. Since we have 32-bit data paths, it takes two cycles to provide the input operands for a 64-bit operation. Consequently, in the schedule as well as in the code, no operation can immediately follow a 64-bit operation. We decided to use this dead cycle to provide two source specifiers and one destination specifier. The rest of the specifiers are provided in the previous instruction (the one initiating the 64-bit operation).

The memory opcode repertoire includes opcodes to read and write 32-, 16-, and 8-bit data. The 16-bit reads and writes have signed as well as unsigned versions. With the 16-bit reads, this determines whether the 16-bit datum is interpreted as a signed or an unsigned integer. This in turn determines whether the sign is extended or zeros are inserted in the high-order 16 bits of the 32-bit destination register. All integer arithmetic is carried out thereafter on 32-bit data. When a 16-bit datum is written back to memory, use of the signed or unsigned opcode determines whether or not the 32-bit quantity in the register can be treated as a 16-bit quantity without overflow. If an overflow occurs, it is reported at the time of the 16-bit write. The 32-bit Exchange Read opcode exchanges the contents of the specified register and memory location as an indivisible operation. This opcode supports synchronization between asynchronous parallel processes.

Although the opcode repertoire is identical for both memory ports, an asymmetry results because of insufficient instruction word bits to go around. Memory port 1 can specify the memory address as an instruction-specified displacement off a base register. Memory port 2 cannot specify a displacement.

The numeric processor has two special opcodes to support loop execution: *brtop* and *nexti*. Two types of actions must be performed to control loops. One is to determine whether another iteration is to be executed and, if so, to allocate a new iteration frame. This is done by the nexti opcode. The other action is to actually branch back to the top of the loop if another iteration is to be executed. The brtop opcode does this in addition to everything the nexti opcode does. If it were not for the long, three-cycle branch latency, the nexti operation would be unnecessary. But in certain very small loops, the interval between the initiation of successive iterations can be less than the branch latency. If not for the nexti opcode, this would pose an unnecessary upper bound on performance of one iteration every three cycles. But with the nexti opcode, it is possible to initiate new iterations in the delay slots of the brtop operation. This allows up to three iterations per brtop executed and the initiation of up to one new iteration every cycle.

Instruction format. The data paths of the numeric processor can initiate six operations every cycle. Therefore, the MultiOp instruction format (Figure 7a) must be able to issue six operations on the six functional units, plus an additional one to control the instruction unit and other miscellaneous operations. A MultiOp instruction consists of seven partitions, one for each operation; each instruction looks like a conventional RISC instruction except for the existence of a predicate specifier. The typical format for each operation partition consists of an opcode, two source-register specifiers, one destination-register specifier, and one predicate-register specifier (Figure 7b).

The data cluster has four context register matrix rows and the GPR to select among for a source, and one row plus the GPR to select between for the destination. Similarly, the address cluster has two context register matrix rows and the GPR to select among for a source, and one row plus the GPR to select between for the destination. Assuming an average of five or six bits in the opcode field, 64 registers in each register file, and 128 locations in the ICR, this implies roughly 40 bits per operation partition or 240 bits per instruction. To avoid the need for complex instruction fetch logic, we were determined that the instruction word width would be a power of 2. Thus 256 bits appeared to be a reasonable target. Furthermore, since an instruction word width of 512 bits would have caused a considerable increase in the cost and complexity of the instruction unit, 256 bits seemed the only option available. This rigid constraint required a number of trade-offs, which are discussed below.

In portions of the program where significant parallelism exists, including but not limited to innermost loops, the MultiOp format is very effective. However, because it gobbles up 32 bytes every 40 nanoseconds, we were worried about the effect on instruction cache performance and capacity should the MultiOp format be used indiscriminately, even in portions of the code where little parallelism exists. With this in mind, we created the UniOp instruction format (Figure 7c), which allows only a single operation to be initiated per instruction, making it possible to fit multiple UniOp instructions in each 256-bit container. The opcode repertoire available in the UniOp format is identical to that in the MultiOp format. While executing UniOp instructions, the numeric processor is similar to other scalar architectures that have no pipeline interlocks in hardware.

The UniOp instruction must contain not only all the information contained in the corresponding MultiOp partition, but also a few additional bits to indicate which functional unit is being tasked. The longest partition in the MultiOp format is for memory port 1, which contains a literal field for specifying an address displacement or a data literal. This partition is 44 bits long. If each UniOp instruction is at least 44 bits, it is possible to fit at most five instructions per instruction container, for an average instruction width of 51 bits. We felt that this would dilate the size of the compiled code to an uncomfortable extent. As a compromise we decided to forgo the predicate capability in UniOp, reasoning that the UniOp sections of code were not supposed to be performance critical anyway. Now it was possible to fit six UniOp instructions per container. Any more than six per container would have required us to reduce the 16-bit address displacement field size. Since we felt that 16 bits was barely adequate, we were unwilling to reduce it further. The 40-bit UniOp format also permitted us to provide a 24-bit program address literal.

The MultiOp format contained 18 context register matrix or GPR specifiers and six ICR specifiers. Increasing the number of registers per register file from 64 to 128 would have required 18 additional instruction bits in MultiOp, which were not available. This helped us realize that adding more registers was not an option. Decreasing the number of ICR locations from 128 to 64 would have saved six MultiOp instruction bits—not enough to make an appreciable difference elsewhere.

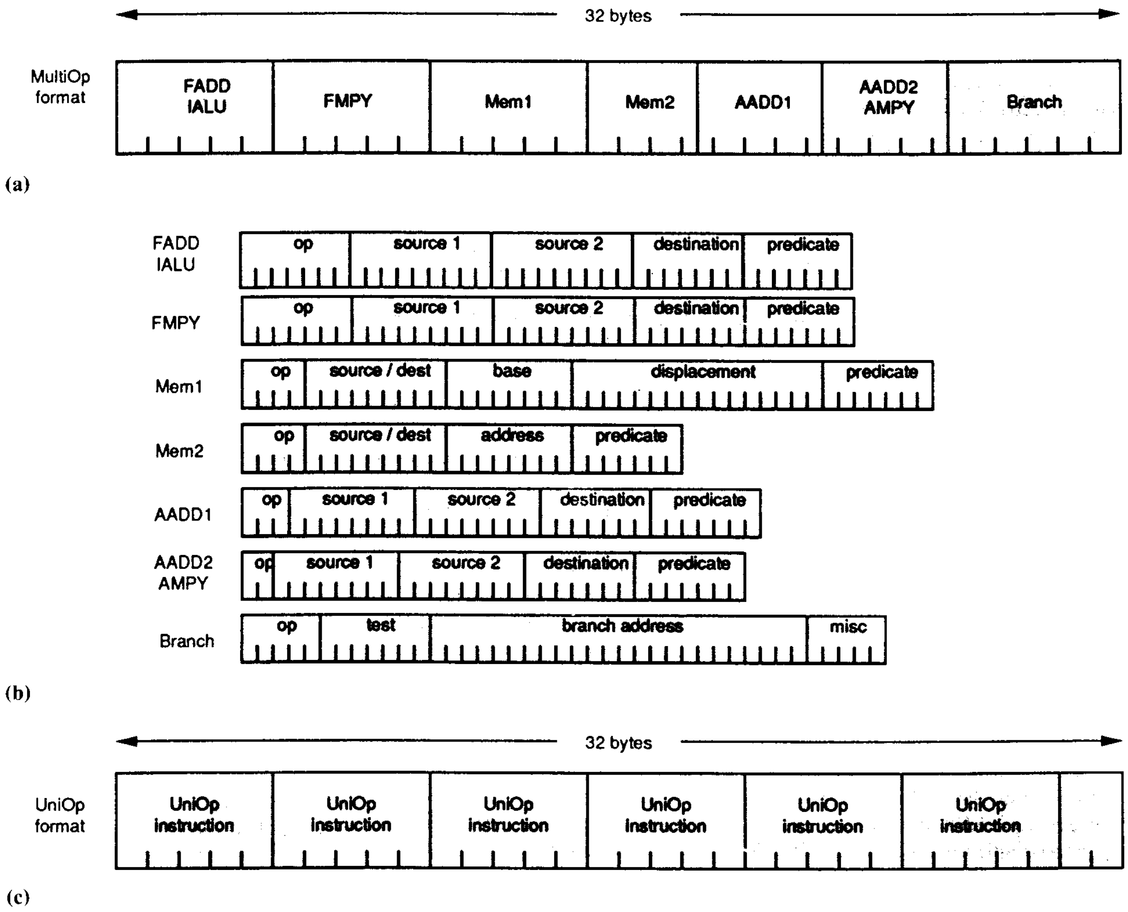


Figure 7. Numeric processor instruction formats. (a) The MultiOp format is 32 bytes long and permits seven operations to be issued during each 40-nanosecond cycle. (b) The structure of each partition in the MultiOp format. (c) The UniOp format allows six instructions to fit into a 32-byte container. Each instruction can issue only one operation per 40-nanosecond cycle.

Exception handling. Exceptions and interrupts pose a special challenge to architectures such as directed dataflow, where the execution sequence is so carefully and rigidly choreographed. The basic problem in handling an exception is *schedule tearing*, which means the carefully crafted computation schedule is being drastically altered by inserting the computation corresponding to the exception handler into the middle of the original computation. The compiler-generated schedule is literally torn apart to allow the exception handler to execute. This can cause two types of problems: first, the conflicting usage of scheduled resources, and second, the violation of implicit dependencies between operations.

Avoidance of resource conflicts is achieved by flushing all pipelines during transition between the user program and the exception handler. This is done by aborting operations in progress, by allowing them to execute to completion, or by saving and subsequently restoring their state of partial execution. Conceptually, the last alternative is the simplest. It is also the costliest in terms of hardware requirements, so it is the solution of last resort. Aborting operations is extremely complicated, since those operations will need to be reissued after exception handling, which means the program counter and the processor state must be backed up. Allowing operations that have been issued to go to completion is the best technique overall, except in certain cases. Clearly, if the exception is due to an operand page fault occurring in the course of executing a memory operation, the memory operation cannot complete until the page fault exception has been handled and the page has been brought into physical memory. In this case the best alternative is to save the state of that portion of the memory pipeline extending through the virtual address translation and to restore it after exception handling.

If special care is not taken, schedule tearing can result in the violation of required dependencies between operations. It can cause an operation scheduled to finish after a second one has started to actually finish *before* the second one starts. This will lead to incorrect results if the first operation writes to the same register the second one reads, and if the second operation expects to get the contents that existed prior to the register's having been overwritten. Such a situation is prevented by a compiler convention that decrees a register to be "in use" from the beginning of the operation that writes to it until the latest completion time of all operations using that value. With this convention in place, two operations that overlap each other's execution intervals will have to use different source and destination registers, thereby avoiding the problem.

Clearly, handling an exception requires that no further instructions (operations) be issued once the exception has occurred. But this can contradict the strategy of executing to completion an operation in progress if the information corresponding to that operation is distributed across multiple instructions. This is often the case in microprogramming-style architectures, where, instead of providing the opcode, the source specifiers, and the result specifiers at the same time, the architecture provides the result specifier many instructions later than the opcode, reflecting the time when each item of information is actually needed.

The problem is that when the exception occurs, not all of the information needed by the operation to execute to completion has been issued; that is, the operation is in a partially issued state. For the operation to execute to completion, further instructions must be issued, which in turn would cause further operations to be issued whose completion would require the issuance of still more instructions, and so on. The only solution would be to issue further instructions selectively in a way that would prevent the issuance of new operations until the operations already in progress had received all the information they need to execute to completion. After the exception has been handled, instruction issuance would have to begin with the first instruction initiating an operation that was not issued prior to exception handling, while taking care to selectively mask out operations issued previously. Because this is so messy, it is highly desirable that all information pertaining to a single operation be specified at the same time in the same instruction.

In the numeric processor, however, double-precision operations are distributed over two consecutive (in time) instructions. Thus, at least to a limited extent, the problems described above must be dealt with. Two measures accomplish this:

(1) The one instruction issued after the exception (since it may contain the second half of a double-precision operation) must be issued with all "new" (that is, single-precision or first half of double-precision) operations disabled.

(2) The opcode for the second half of a double-precision operation should be such that in isolation (when viewed as a single-precision operation or the first half of a double-precision operation) it will be interpreted as a no-op.

The first requirement allows the appropriate functional unit to get the information it needs to allow a double-precision operation issued in the previous cycle to execute to completion. The second requirement makes it simple to resume issuing instructions after handling the exception. Since they are interpreted as no-ops, second halves of double-precision operations will automatically "mask" themselves out.

However, the first requirement cannot be met when the second half of a double-precision operation lies in a different page from that in which the first half lies and if the instruction fetch for the second half generates a page fault. In this case the double-precision operation must be aborted and restarted by resuming instruction issuance with the last instruction actually issued. For this to be possible, the inputs to operations issued in the instruction to be restarted must not have been modified. This will be true if another compiler convention is observed: The registers that are sources to an operation in a particular instruction should be viewed as "busy" through the end of the issuance of all double-precision operations whose first halves are in the same instruction.

PC history queue. One consequence of very deep pipelines is that between the issuance of an operation and its completion, it is possible to have executed multiple branch operations. If an operation generates an exception, the deep pipelining makes it extremely difficult for a debugger to figure out the program counter value for the instruction that initiated the offending operation. To solve this problem, we included a circular 256-entry *PC history queue* (PCHQ). During normal operation the current PC value is written into the PCHQ on every cycle. The state of the PCHQ is frozen when an exception occurs. Knowing the latency of the offending operation, the debugger can index back into the PCHQ by that amount to locate the PC value for the corresponding instruction.

Although this was the original motivation for the PCHQ, it was soon pressed into service for an additional function. Again due to the depth of the pipelines, between the time an exception occurs and the time all pipelines have been flushed, many operations complete that can generate additional exceptions. These operations will not be reissued after exception handling, so the exceptions must all be recorded and handled en masse. Since this exception logging process occurs only after the first exception has occurred, and since the PCHQ has stopped recording PC values at this point, we decided to switch the PCHQ from the task of recording PC values to the task of recording exception records at the time the first exception occurs.

The main memory system

The ideal memory system for a supercomputer would provide large capacity at a low price and extremely high bandwidth with very low access time. Furthermore, the bandwidth and access time would be insensitive to the size of the data sets being operated on, the manner in which the data are placed in memory, and the order in which they are referenced. Needless to say, such a memory system has never been built. Each computer architect must decide which of these attributes are essential and which can be compromised.

Data cache anomalies. General-purpose computing almost invariably employs caches. Assuming locality of reference and

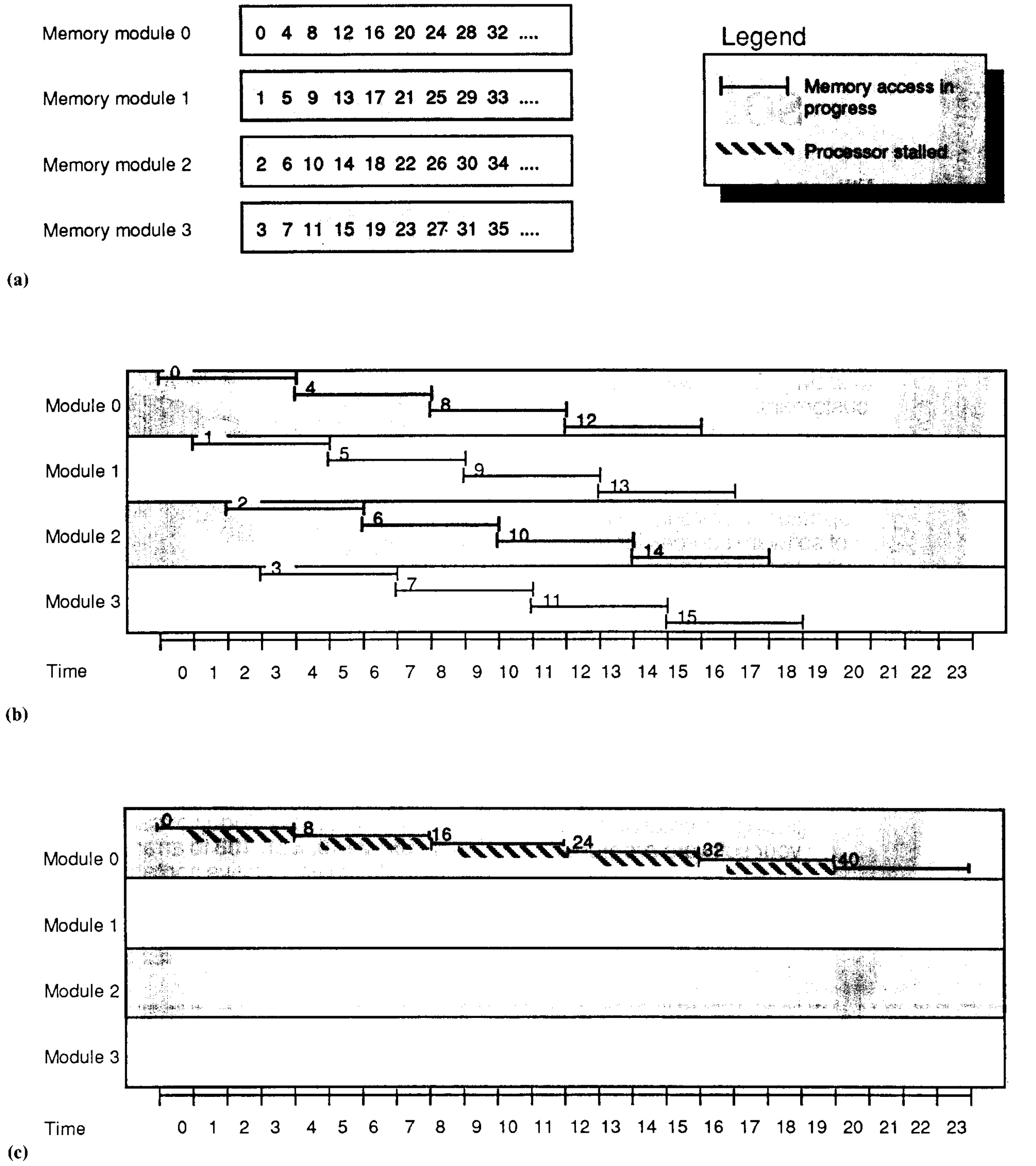


Figure 8. Sequentially interleaved memory. (a) The conventional assignment of memory locations to memory modules in a sequentially interleaved memory system with four modules. A memory module is busy for four cycles when handling a request. Thus, the peak bandwidth is one request per cycle. (b) With a sequential request stream, perfect operation takes place. No request ever encounters a busy module, and the peak bandwidth is achieved. (c) For a request stream with a stride of eight, every request is directed to the same module. Every request encounters a busy module, the processor must halt three cycles for each cycle it advances, and the achieved bandwidth is one request per four cycles.

hence a high hit rate, the cache provides the desired high bandwidth and, on the average, low access time, while the main memory provides large capacity at a low price. Whereas the assumption of good locality is usually true with general-purpose work loads, it can be wildly wrong in numerically intensive computing. Often, numerical applications sweep through large arrays such that a particular element is rereferenced only after all other elements have been referenced. Except in the case of toy problems, the arrays tend to be comparable to the main memory in physical size and considerably larger than any realistic cache. Consequently, each word is displaced from the cache before it is next referenced, resulting in a low hit ratio.

The processor is now working directly out of the main memory, which typically is underdesigned for this situation, since the design assumption was that only a small fraction of the references would come through to the main memory. Worse yet, if the stride with which the processor is referencing memory is equal to or greater than the cache line size, the cache will fetch an entire line for each reference that the processor makes, and all but one word of the line is wasted. Far from helping the situation, the cache is now compounding the problem by amplifying the request rate to an already underdesigned main memory. This phenomenon has been researched and reported by Abu-Sufah and Mahoney.[12]

In the case of a really high-performance processor, a further problem makes using a data cache difficult. In addition to the bandwidth needed for instruction fetching, it is necessary to perform two or three data references per processor cycle to keep memory bandwidth in balance with the processor's computational capability. This requires the use of either an interleaved cache or multiple caches, along with a cache coherency mechanism. At extremely fast clock rates, both alternatives present formidable obstacles. In view of these considerations, we elected not to use a cache for data references. We provided a 32-Kbyte cache for instructions, since cache performance for instruction references is not qualitatively different for numerical programs.

Sequentially interleaved memory architectures. The full operand request rate now had to be handled by the main memory, and we were back to the problem of providing a consistently high bandwidth and low access time using main-memory technology. In the context of a departmental supercomputer price objective, this meant that using fast, static, ECL RAM was precluded, and we had to use the relatively inexpensive but slow MOS DRAM technology. The only way to achieve high bandwidth with slow memory technology is to use multiple memory modules in an interleaved fashion. In a normal, sequentially interleaved memory, with an interleave factor of M modules, every Mth word is in the same memory module (Figure 8a). In the case of a sequential reference stream, this ensures high bandwidth, since all modules are referenced before the same module is referenced again (Figure 8b). If the degree of interleaving is large enough compared with the ratio of the memory cycle time to the processor cycle time, the memory module will be ready to handle another request by the time it is referenced again.

Although interleaved memories can provide the bandwidth requirements of high-performance processors, they do not address the desire for short access times. Without the use of a cache, the access time, even under the best of circumstances, can be no less than the access time of the memory technology used in the main memory. Since in numerically intensive computations processor performance is more rigidly linked to memory bandwidth than to memory access time, supercomputer architectures have evolved in such a way as to be relatively insensitive to memory access time. In vector processors the memory access time contributes only to the vector start-up penalty, not to the vector execution rate. Likewise, in the dataflow and directed-dataflow architectures, a longer access time is handled by scheduling the memory access earlier than it is needed.

Parenthetically, this explains why general-purpose processors are better "MIPS engines" than supercomputers running at the same clock speed. In general-purpose computing the emphasis is less on iterative computations and memory bandwidth and more on branching, procedure calls and returns, and memory access time. This makes cache memories indispensable. Without its cache memory, a high-speed scalar processor would slow down to a crawl.

The stride problem. In well-designed supercomputer architectures, the trade-off is always in the direction of ensuring consistently high memory bandwidth, even at the expense of increased access time. However, conventional, sequentially interleaved memories cannot guarantee even high bandwidth. They break down badly if the references have a stride that is a multiple of the degree of interleaving (Figure 8c). When this happens, every reference is to the very same memory module, and the bandwidth is degraded to that of a single memory module. The processor's performance drops proportionately. On existing supercomputers, the magnitude of this penalty is so large[13] that the user is forced to contort the algorithm to avoid this stride problem.

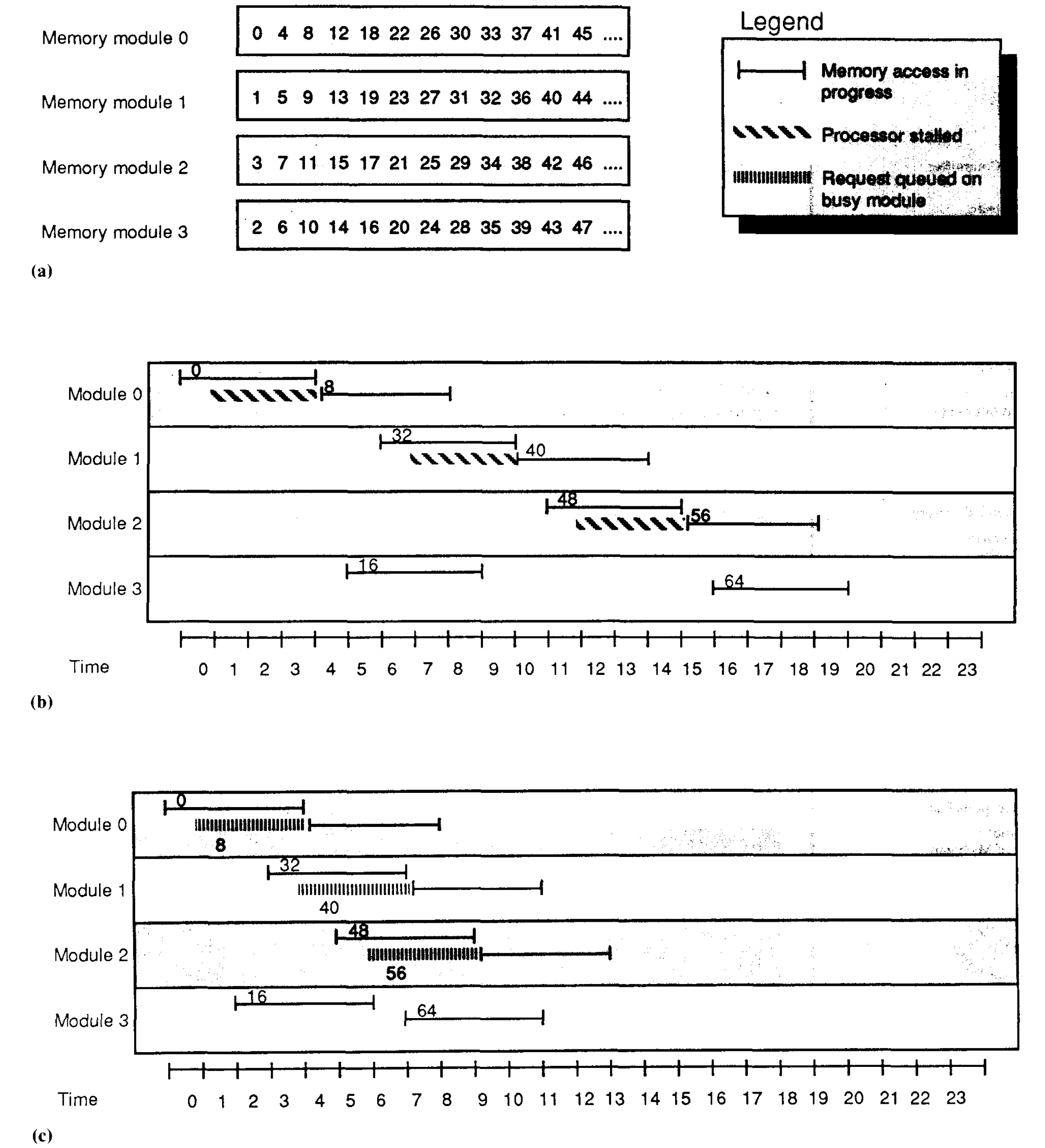
The ideal memory system for a supercomputer would provide large capacity, low price, high bandwidth, and very low access time.

Pseudorandomly interleaved memory architecture. We found this situation unacceptable and developed an interleaved memory architecture that is impervious to the stride problem. Instead of assigning every *M*th word to the same memory module, we assigned the memory locations to the memory modules in a carefully engineered pseudorandom fashion such that every reference sequence likely to occur in practice would be as uniformly distributed across the memory modules as would a truly random request sequence (Figure 9a). To someone familiar with the folklore of interleaved memory design, this might seem like exactly the wrong thing to do. It is a popularly held belief that an *M*-way interleaved memory with a random request sequence will only achieve a bandwidth proportional to $\sqrt{M}$ modules instead of getting the full benefit of the *M* modules. This is true (Figure 9b) if the memory system does not have the facilities to queue-up references to busy modules.[14] With sufficient buffering (Figure 9c), the full bandwidth of *M* modules can be achieved.[15] Furthermore, since every request sequence, whether sequential, of stride *M*, or totally scrambled, appears equally random to the interleaved memory, this high bandwidth is consistently achieved. The only exception is a situation in which the same location is repeatedly referenced (for instance, a scalar memory reference in a loop). Standard, machine-independent optimizations in the compiler get rid of such situations. Thus, high bandwidth is guaranteed regardless of how data is placed in memory and how it is referenced.

But, as always, there is no free lunch. The price of guaranteeing consistent high bandwidth is an increase in access time in high-bandwidth situations. When the request rate is high (close to the maximum bandwidth the memory system is designed for), the randomness of the pseudorandomized request sequence will cause queues to form every so often on busy modules. A request arriving at such a queue will experience a delay equal to the memory chip access time plus the time spent waiting in line. Thus, the access time perceived by the processor increases. Also, this increase in access time is a stochastic quantity, and the overall access time for a request now lies within a range of values. Whereas the limits on the range can be predicted, the exact value (within that range) for the access time of a specific request cannot. Moreover, this range shifts, depending on the request rate of the processor. Under light load conditions (as when executing scalar code), one can expect little queueing delay, but when the request rate increases (within innermost loops), so will the queueing delay.

The memory latency register. In and of itself, the increased access time is not a major problem; the second-order penalty due to the increased access time is more than compensated for by the first-order benefit of guaranteeing high bandwidth. (This does, however, cause a further polarization between a well-designed general-purpose processor and a well-designed supercomputer.) What *is* of concern is the nondeterministic nature of the access time in the context of a processor architecture in which every operation is rigidly scheduled at compile time and in which the latency of every operation, including the memory operations, must be deterministic. The way other architectures that use compile-time scheduling normally handle this is to "fake" a deterministic access time.[6] If the memory access occurs sooner than expected, the data can be buffered internally to the memory system and delivered to the processor at exactly the right time. If, on the other hand, the data takes longer than expected, the processor is "frozen" until the data is available, so that in the processor's "virtual time" the request always takes the same amount of time.

Yet another delicate trade-off exists here. If the compiler consistently underestimates the access time, the processor will spend a significant fraction of its time in a frozen state. If the compiler consistently overestimates the access time, the schedules generated at compile time are unnecessarily dilated. At either extreme, performance is less than optimal. We addressed this issue by simulating the memory system at various request rates and plotting performance against the nominal (assumed) memory latency. As expected, we found that for each request rate the curve peaked at a certain value of memory latency. In the vicinity of this optimum memory latency, performance was not particularly sensitive to the value of the memory latency. On the other hand, the value of the optimum memory latency



Memory module 0	0 4 8 12 18 22 26 30 33 37 41 45 ....
Memory module 1	1 5 9 13 19 23 27 31 32 36 40 44 ....
Memory module 2	3 7 11 15 17 21 25 29 34 38 42 46 ....
Memory module 3	2 6 10 14 16 20 24 28 35 39 43 47 ....

Figure 9. Pseudorandomly interleaved memory. (a) The assignment of memory locations to four memory modules in a pseudorandomly interleaved memory. (b) Even with a stride of eight, eventually the requests are evenly distributed across all four memory modules. However, every so often requests will encounter a busy module. If no buffering is provided at the memory modules, the processor must halt until the module is no longer busy. (c) Although individual requests might have to wait even with buffering, the processor need not. It can continue to issue a request every cycle, yielding full bandwidth.

was very sensitive to the request rate. This made us nervous about hardwiring the nominal memory latency value into the compiler and the hardware.

We solved this problem by incorporating a memory latency register. The MLR is a programmatically writable register that always holds the value of the memory latency assumed by the compiler when scheduling the currently executing code. The memory system uses the value in this register to decide whether the datum is early or late and, consequently, whether the datum should be buffered or the processor frozen. When executing scalar code with little parallelism and a low request rate, the MLR is set to the minimum possible memory access time of 17 numeric processor cycles (each cycle is 40 nanoseconds). When the program is in an innermost loop, the MLR is set to the optimum value of 26 cycles to reflect the expected delay due to the higher request rate. The MLR allows the compiler to treat memory accesses as having a deterministic latency but to use different values for the latency in different portions of the code so as to always deliver near-optimal performance.

Compiler-scheduled memory modules. One of the alternatives we considered, and decided against, very early in the design process was to place the memory modules under the explicit scheduling control of the compiler (much like the adder and multiplier) instead of treating the memory system like a black box. Ideally, in such a scheme the compiler must know *at compile time* whether or not a set of references are to distinct memory modules. It can then schedule the initiation of the memory requests in such a way that the referenced memory module is no longer busy by the time the request is made. This would eliminate the need for any buffering. Also, with knowledge of the (deterministic) memory latency that exists in the absence of queuing, operations that use the data from memory can be scheduled to occur no sooner than when the data is available. The processor would never need to be frozen, nor would access times need to be overestimated. Presumably, the hardware would be simpler and less expensive with no buffering facilities. Thus, near-optimal performance could be achieved if the appropriate information were known at compile time.

But commonly occurring program constructs can defeat such a strategy. In the case of references to $X(I)$ and $X(J)$, the manner in which the variables I and J are computed may be such as to preclude the compiler's being able to determine whether the modules referenced are distinct or the same. Another problem lies with subscripted-subscript references to arrays, such as $X(JA(I))$, where the index into one array, $X(\,)$, is determined by reading the contents of another array, $JA(I)$. Since the contents of $JA(\,)$ are determined at runtime, the compiler is once again unable to predict which module will be referenced. In such circumstances the compiler must either assume the worst and serialize all such references or assume that no conflict exists and count on the presence of some hardware mechanism that will freeze the processor if a request is submitted to a busy module. With the latter approach, the $\sqrt{M}$ law becomes applicable. In either case the program experiences a sizable drop in performance.

As we see it, the most serious drawback of compile-time scheduling of memory modules is that it does nothing to address the stride problem. With a bad stride, whether compile-time memory disambiguation works or not, the memory bandwidth collapses. Memory disambiguation merely confirms the bad news at compile time. Using pseudorandom interleaving to solve the stride problem would make the task of compile-time disambiguation next to impossible. So eventually the trade-off was one of simpler hardware, more complex software, and a reduced average access time versus a guaranteed, consistently high bandwidth. In view of our central objective of providing a product with minimal difficulty of use, we chose the latter.

Reflections

While developing this product, we became aware of certain broad truths, and we have tried to convey them in this article. The most important of these is that the behavior of general-purpose and numerically intensive work loads can be drastically different. In nonnumeric programs the emphasis is on branching and procedure calls; in numeric programs it is on loops. General-purpose jobs tend to access their data with a high degree of locality. This is not always so with numerically intensive jobs.

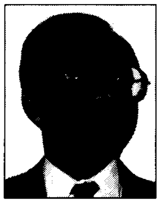
Consequently, the design decisions and trade-offs steadily push the well-designed scalar processor and the well-designed numeric processor apart. In a scalar processor the emphasis is on short pipeline latencies rather than on extensive parallelism. In a numeric processor the emphasis is on multiple, parallel pipelines even at the expense of very deep pipelines and, hence, reduced scalar performance. Whereas the use of caches for data is virtually mandatory for good performance in a scalar processor, it is far less beneficial, and sometimes even detrimental, to the performance of a numeric processor. A numeric processor is more sensitive to memory bandwidth and less so to access time. The opposite is true for a scalar processor. Thus it remains as difficult now as it has been in the past to design a single machine that is best for both numeric and nonnumeric work loads.

Much has been said over the past few years about the relative merits of RISC and CISC approaches to hardware design. We feel that the same issue could well be raised regarding software, specifically compiler software. With the headlong rush to move complexity out of hardware and into software, compilers are beginning to groan under the burden of newly acquired responsibilities. It is possible to go too far and to end up increasing the total complexity of the hardware-software system. The designers' responsibility is to minimize overall complexity, not just that of the hardware. In the Cydra 5 we decided not to transfer complexity from hardware to software in some areas such as the context register matrix, conditional scheduling control, and hardware-scheduled memory modules. To paraphrase Einstein, "Hardware should be as simple as possible, and no simpler."

Designing and developing a product of this performance level and with these capabilities necessitated a break with architectures of the past so that we could incorporate a more powerful model of computation. This meant that we often had to fly by the seat of our pants, there being little experience or data on which to base our decisions. Fortunately, we have not yet discovered any major blunders, although it is quite possible that the machine has been overdesigned in various places because of our tendency to err on the safe side. These areas of overdesign will reveal themselves slowly as we build up our experience in the use of this new architecture.

Today, a number of Cydra 5 systems are in use at customer sites. The performance of these systems has met our expectations. On widely quoted industry-standard benchmarks such as Linpack[16] and the Livermore Fortran Kernels,[17] the Cydra 5 delivers 15.4 Mflops and 5.8 Mflops, respectively. This is the highest performance of any minisupercomputer (even those whose peak performance is twice that of the Cydra 5) and about one-third the performance of a Cray X-MP supercomputer, which has nine times the Cydra 5's peak performance. On the 24 Livermore Fortran Kernels taken as a group, the Cydra 5 can achieve 23 percent of its peak performance as opposed to 15 percent and 8 percent for VLIW and vector processors, respectively. On Linpack, which is considerably more vectorizable, the Cydra 5 achieves 60 percent of its peak performance. The VLIW and vector processors achieve only 40 percent and 20 percent, respectively.

On other, less vectorizable, benchmarks, such as ITPack[18] (an iterative sparse-matrix solver), the Cydra 5 achieves half the performance of the Cray X-MP. We have even encountered a couple of extremely nonvectorizable applications in which the Cydra 5 has actually achieved parity with the Cray X-MP. In general, across a spectrum of applications, the Cydra 5 can achieve between one-fourth and two-thirds the performance of a Cray X-MP, depending on the extent to which the application is vectorizable. However, there is still room for improvement, since the compiler has not yet peaked in its ability to wring performance out of code. □



B. Ramakrishna Rau is a cofounder and the chief technical officer of Cydrome, Inc. He is the chief architect of Cydrome's Cydra 5 minisupercomputer based on the company's directed-dataflow architecture. Previously he held positions at Elxsi and TRW, and he taught electrical engineering at the University of Illinois, Urbana-Champaign. His research interests include parallel architectures, compiler techniques for high-performance computing, and analytical methods for computer performance evaluation and prediction.

Rau received a B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, in 1972 and MS and PhD degrees in electrical engineering from Stanford University in 1973 and 1977, respectively. He is a member of the IEEE, the IEEE Computer Society, and the Association for Computing Machinery.

Acknowledgments

The Cydra 5 owes its existence to the group of bright and dedicated individuals who make up Cydrome. Each has left his or her mark on this machine. We wish to acknowledge the especially important contributions of Stimson Ho, Gary Beck, Joe Bratt, Ed Wolff, Mike Schlansker, and John Brennan.

References

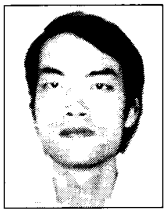
1. "Cydra 5 Departmental Supercomputer Product Summary," Cydrome, Inc., Milpitas, Calif., 1988.
2. B.R. Rau, "Cydra 5 Directed Dataflow Architecture," *Proc. Compcon Spring 88*, No. 828, Computer Society Press, Los Alamitos, Calif., pp. 106-113.
3. B.R. Rau, C.D. Glaeser, and R.L. Picard, "Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support," *Proc. Ninth Ann.*

David W.L. Yen cofounded Cydrome, a minisupercomputer manufacturer, in 1984. He contributed to the Cydra 5 architecture design and project planning and served as director of hardware development. He joined Sun Microsystems in October 1988. In addition, Yen has engaged in research and design for the IBM San Jose Research Laboratory, TRW Array Processors, and the Coordinated Science Laboratory at the University of Illinois. His interests include computer architecture, special processors for high-performance requirements, computer-aided design automation, and product development.

Yen received a BS from National Taiwan University in 1973 and MS and PhD degrees from the University of Illinois, Urbana-Champaign, in 1977 and 1980, respectively, all in electrical engineering. He is a member of Phi Kappa Phi and Eta Kappa Nu, and he served as secretary of the IEEE Computer Society's Computer Standards Committee from 1983 to 1984.

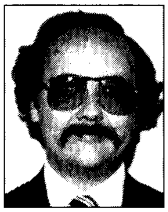
Int'l Symp. Computer Architecture, M411 (microfiche), Computer Society Press, Los Alamitos, Calif., 1982, pp. 131-139.

4. W.C. Yen, D.W.L. Yen, and K.S. Fu, "Data Coherence Problem in a Multicache System," *IEEE Trans. Computers*, Vol. C-34, No. 1, Jan. 1985, pp. 56-65.
5. J.A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," *Proc. 10th Ann. Int'l Symp. Computer Architecture*, M473 (microfiche), Computer Society Press, Los Alamitos, Calif., 1983, pp. 140-150.
6. A.E. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family," *Computer*, Vol. 14, No. 9, Sept. 1981, pp. 18-27.
7. Y.N. Patt, W.-M. Hwu, and M. Shebanow, "HPS, a New Microarchitecture: Rationale and Introduction," *Proc. 18th Ann. Workshop Microprogramming*, M653 (microfiche), Computer Society Press, Los Alamitos, Calif., 1985, pp. 103-108.
8. D. Cohen, "A Methodology for Programming a Pipeline Array Processor," *Proc. 11th Ann. Workshop Microprogramming*, M204 (microfiche), Computer Society Press, Los Alamitos, Calif., 1978, pp. 82-89.
9. J.R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, Mass., 1986.
10. Arvind and K.P. Gostelow, "The U-Interpreter," *Computer*, Vol. 15, No. 2, Feb. 1982, pp. 42-49.
11. D.J. Kuck, *The Structure of Computers and Computation*, John Wiley and Sons, New York, 1978.
12. W. Abu-Sufah and A.D. Mahoney, "Vector Processing on the Alliant FX/8 Multiprocessor," *Proc. Int'l Conf. Parallel Processing*, M724 (microfiche), Computer Society Press, Los Alamitos, Calif., 1986, pp. 559-563.
13. J.M. van Kats and A.J. Van der Steen, "Minisupercomputers, a New Perspective?" Report TR-24, Academisch Computer Centrum Utrecht, University of Utrecht, Utrecht, Netherlands, May 1987.
14. H. Hellerman, *Digital Computer System Principles*, McGraw-Hill, New York, 1967, pp. 228-229.
15. F.A. Briggs and E.S. Davidson, "Organization of Semiconductor Memories for Parallel-Pipelined Processors," *IEEE Trans. Computers*, Vol. C-25, Feb. 1977, pp. 162-169.
16. J.J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment," Tech. Memo No. 23, Argonne National Laboratory, Argonne, Ill., Jan. 1988.
17. F. McMahon, "The Livermore Fortran Kernels," National Technical Information Service, Ann Arbor, Mich., Dec. 1986.
18. T.C. Oppe and D.R. Kincaid, "The Performance of ITPACK on Vector Computers for Solving Large Sparse Linear Systems Arising in Sample Oil Reservoir Simulation Problems," *Comm. Applied Numerical Methods*, Vol. 3, 1987, pp. 23-29.



Wei Yen is vice president for development, in charge of engineering, at Cydrome. He is the primary system architect for the Cydra 5 minisupercomputer and was extensively involved in the design and development of Cydrome's operating system and compilers. He has also worked on development projects at the Fairchild Advanced R&D Laboratory and Hewlett-Packard Laboratories. His technical interests include multiprocessor systems, distributed systems, programming environments, and dataflow compiler design.

Yen received a PhD in electrical engineering from Purdue University in 1981. He is a member of Phi Kappa Phi, Sigma Xi, and the Association for Computing Machinery.



Ross A. Towle is cofounder and president of Apogee Software, a company specializing in compilers for RISC and VLIW architectures. He was also a cofounder of Cydrome, where he served as manager of languages. His interests include global optimization, parallelization, and instruction scheduling.

Towle received BS and MS degrees in mathematics and a PhD in computer science, all from the University of Illinois, Urbana-Champaign.

The authors can be contacted through B. Ramakrishna Rau at 159 Belvue Dr., Los Gatos, CA 95032.