JULY 1989

WRL Research Report 89/8

A Unified Vector/Scalar Floating-Point Architecture

Norman P. Jouppi, Jonathan Bertoni, and David W. Wall

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
DARPA Internet:	WRL-Techreports@decwrl.dec.com
CSnet:	WRL-Techreports@decwrl.dec.com
UUCP:	decwrl!wrl-techreports

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

A Unified Vector/Scalar Floating-Point Architecture

Norman P. Jouppi, Jonathan Bertoni, and David W. Wall

July, 1989

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

In this paper we present a unified approach to vector and scalar computation, using a single register file for both scalar operands and vector elements. The goal of this architecture is to yield improved scalar performance while broadening the range of vectorizable applications. For example, reduction operations and recurrences can be expressed in vector form in this architecture. This approach results in greater overall performance for most applications than does the approach of emphasizing peak vector performance. The hardware required to support the enhanced vector capability is insignificant, but allows the execution of two operations per cycle for vectorized code. Moreover, the size of the unified vector/scalar register file required for peak performance is an order of magnitude smaller than traditional vector register files, allowing efficient on-chip VLSI implementation. The results of simulations of the Livermore Loops and Linpack using this architecture are presented.

This is a preprint of a paper that will be presented at the
3rd International Conference on Architectural Support for Programming Languages and Operating Systems,
IEEE and ACM, Boston, Massachusetts, April 3-6, 1989.
An early draft of this paper appeared as WRL Technical Note TN-3.

1. Introduction

Specialized vector hardware has been available on supercomputers since the mid 1970's [13]. Recently, specialized vector hardware has also been appearing on traditional mainframes [7], minisupercomputers [15, 9] and even workstations ("solo supercomputers") [6]. This specialized vector hardware often adds substantial complexity to the scalar machine, especially at the high and low end (i.e., supercomputers and workstations).

Vectorization improves the peak performance of an application. However, even if the portions that do vectorize run arbitrarily fast, the net performance only improves by the percentage of vectorizable code. Since the range of vectorization in general-purpose scientific computing is typically 0.3 to 0.7 [16], infinitely fast vector performance would only improve the performance of the entire benchmark by 1.4 to 3.3 times. Rather than trying to improve performance by increasing the peak vector computation rate, in the MultiTitan FPU we obtained more leverage by improving scalar performance, while broadening the range of vectorizable applications. This leads to better overall performance for most applications with small or modest amounts of classically vectorizable code.

In this paper we present a unified approach to vector and scalar floating-point computation. This floating-point architecture was developed as part of the MultiTitan project from 1985 through 1987. The approach stresses very high performance scalar operation. It then adds an insignificant amount of hardware to provide a relatively small (2x) performance improvement for classically vectorizable code. However, this additional hardware also yields improved performance for operations not normally vectorizable, such as reductions and recurrences. The net result is to improve the non-peak performance as well as broadening the range of peak performance.

The hardware architecture of this approach is given in Section 2. Section 3 presents results of simulation studies for vectorized and non-vectorized versions of various benchmarks. Section 4 concludes the paper.

2. Hardware Architecture

Figure 1 shows the system context for the floating-point architecture. The floating-point unit (FPU) consists of one chip of approximately 120,000 transistors. A 64-bit data and 32-bit address bus are shared by the CPU chip and the FPU chip. At this level in the design the vector

capability of the FPU is not visible. All loads and stores of FPU registers take place from the 64KByte direct-mapped data cache shared with the CPU. This cache has 16 byte lines and a 14 cycle miss penalty.

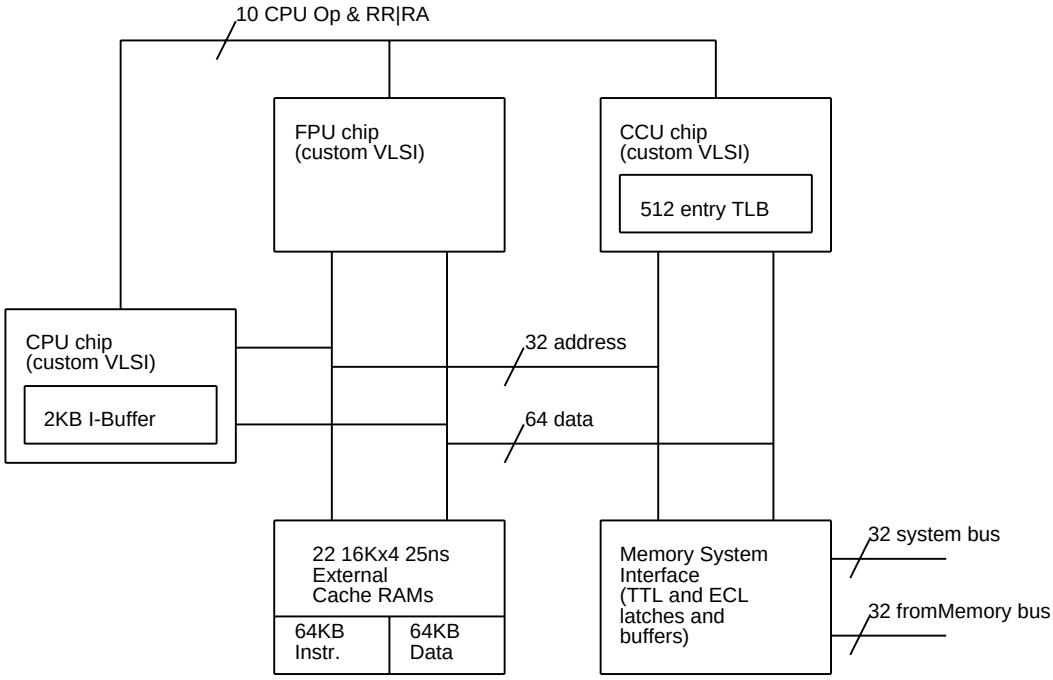

Figure 1: Block diagram of one MultiTitan processor

Figure 2 gives an overview of the architecture of the FPU. The FPU has three fully pipelined independent functional units: add, multiply, and reciprocal approximation. (Reciprocal approximation, coupled with use of the multiply unit, is used to implement division.) Any functional unit can accept a new set of operands each cycle and produce a new result each cycle. The latency of the functional units is three cycles for all operations (i.e., the result of a computation is available three cycles after it is issued to the functional unit.)

A register file, containing 52 general-purpose 64-bit registers, sits between the functional units and the data cache. The register file has four ports: two ALU source operands are read from the A and B ports, ALU results are written on the R port, and loads write and stores read the memory (M) port. In addition the FPU PSW is conceptually in the register file.

There are two separate instruction registers for controlling the operation of the FPU. One holds Load/Store instructions which are transmitted from the CPU over a 10-bit coprocessor instruction bus. The 10 bits supply an opcode (4 bits) and source or destination register specifier (6 bits). The second instruction register is 32 bits wide and holds FPU ALU instructions. These are transferred over the address bus from the CPU. The separate Load/Store and ALU instruction registers allow FPU loads and stores to proceed in parallel with the issue of FPU ALU operations.

Three key features distinguish our work in floating-point architecture: a unified approach to scalar and vector processing, low latency floating-point functional units, and simplicity of organization.

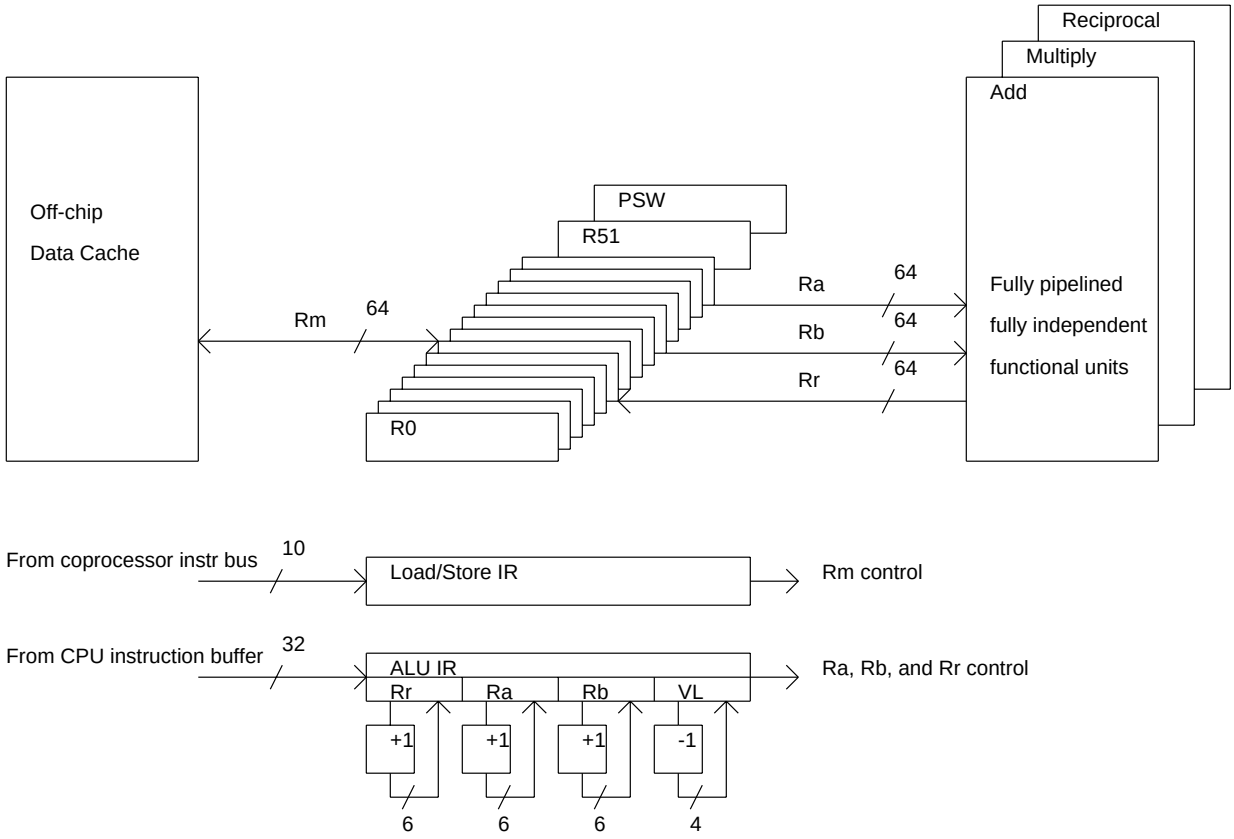

Figure 2: Microarchitecture of the FPU

2.1. A Unified Vector/Scalar Register File

Traditionally, machines that support vectors and use a load/store architecture (supporting only register-to-register arithmetic) provide separate register sets for vector and scalar data. This creates a distinction between elements of a vector and scalars, where none actually exists. This distinction makes mixed vector/scalar calculations difficult. For example, when vector elements must be operated on individually as scalars they must be transferred over to a separate scalar register file, only to be transferred back again if they are to be used in another vector calculation. This distinction is unnecessary. The MultiTitan floating-point architecture provides a single unified vector/scalar floating-point register file. Vectors are stored in successive scalar registers. This allows individual vector elements to be addressed and accessed with scalar operations, unlike classical vector machines. Each arithmetic instruction contains a vector length field, and scalar operations are simply vector operations of length one.

2.1.1. Vector ALU operations

The format of FPU ALU instructions is given in Figure 3. Figure 4 summarizes the operation of the func and unit fields. Vector instructions are issued by merely incrementing register fields in the instruction register and issuing the resulting instructions with the same mechanism used for scalar operations. The vector length field specifies the number of elements in the vector, from 1 to 16. The only means of specifying vector length is statically in the instruction itself; there is no dynamically loadable vector length register. After issuing the first instruction in the vector, the vector length field is checked. If it is zero, the instruction is cleared from the instruc-

tion register. If it is non-zero, the vector length field is decremented and the appropriate register specifiers are incremented. This instruction is then treated the same as any other instruction newly placed in the instruction register. If the SRa (stride Ra) bit is zero, register source field Ra does not increment (i.e., it is a scalar). If the SRb bit is zero, register source field Rb does not increment. If both bits are zero then "vector := scalar op scalar" is performed.

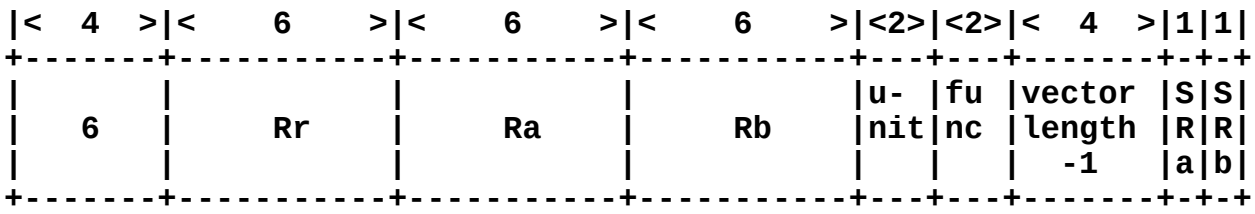

Figure 3: FPU ALU instruction format

With this organization, operations that are not vectorizable on other machines can be vectorized. Since the normal scalar scoreboarding is used for each vector element, reduction and recurrence operations can be naturally expressed in vector form. For example, the inner loop of matrix multiplication consists of a dot product in which the elements of a vector multiply must be summed (i.e., a reduction). Since there is no distinction between vector and scalar registers, the reduction can be performed with scalar operations without moving the data from the result registers of the vector multiply. In fact there are several different ways in which the reduction may be performed.

operation	unit	func
reserved	0	X
add	1	0
subtract	1	1
float	1	2
truncate	1	3
multiply	2	0
integer multiply	2	1
iteration step	2	2
reserved	2	3
reciprocal	3	0
reserved	3	1-3

Figure 4: Func and Unit field operation

First, consider an implementation of the sum utilizing scalar add instructions. Assume the elements to be summed are in registers R0 through R7 (see Figure 5). We can sum adjacent pairs of elements of the vector multiply result, placing the partial sums in temporary registers R8 through R11. For each sum we need to transfer a FPU ALU instruction over the address bus from the CPU. (We could also reuse the vector multiply result registers, but extra temporary registers have been used for clarity.) Next we can pairwise sum R8 through R11 into R12 and R13. Note that the sum of R10 and R11 cannot issue in cycle 5 since R11 is not yet available. A simple result reservation mechanism stalls the issue until cycle 6 when both operands are available. If some other independent CPU or FPU instruction is available, it would typically be scheduled before the sum of R10 and R11 to prevent the cycle from being wasted. Similarly, the final addition of R12 and R13 to complete the sum cannot issue until cycle 9. The total time required to sum 8 elements of a vector is 12 cycles.

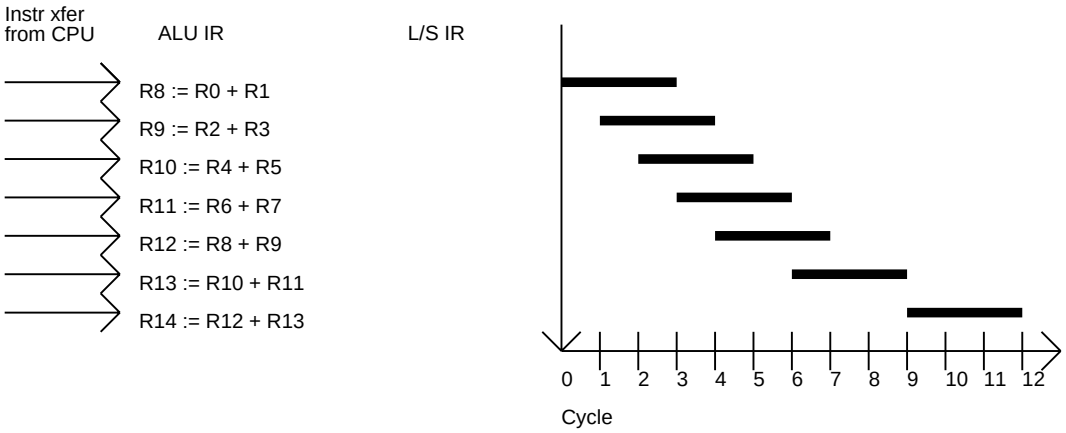

Figure 5: Summing with a tree of scalar operations

A second way to implement the reduction is with a vector ALU instruction. (Actually, all of the scalar adds in the previous example were vectors of length 1.) This is illustrated in Figure 6. In this example, we initialize R8 to 0 and sum each of the registers R0 through R7 with R8. This is not a particularly time-efficient way of performing the sum but it illustrates an important point. Since the normal scalar issuing hardware is used for issuing each element of the vector, arbitrary data dependencies between elements of a vector are possible. This is in contrast with classical vector architectures, where arbitrary data dependencies between elements are not allowed. In this case, each element depends on the result of the previous element's computation, so the overall computation takes 24 cycles. While the vector instruction is executing, other instructions may be issued by the CPU. However, while the vector is issuing the FPU ALU instruction register is occupied, so no other FPU ALU instructions can be issued.

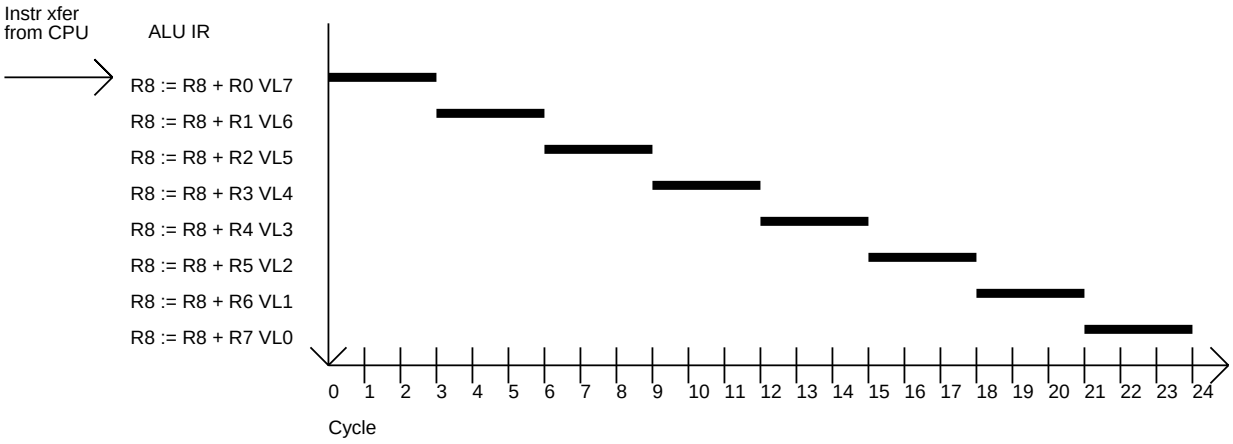

Figure 6: Summing with a linear vector

A third way to implement the sum is to use a tree of vector operations (see Figure 7). This computation is identical to the scalar tree version of the computation with two exceptions. First, since the register specifiers are incremented only by 0 or 1 between elements of a vector, the

pairs summed must be (R0,R4), (R1,R5), (R2,R6), and (R3,R7) instead of (R0,R1), (R2,R3), (R4,R5), and (R6,R7) as in the scalar example. Second, only three instructions must be issued by the CPU to perform the sum. This frees the CPU to issue more instructions concurrent with the summation. In this example there are 9 cycles out of the 12 in which the CPU may issue other instructions. In this matrix multiply example it would allow the 8 elements of the next row to be loaded in parallel with the reduction of the current row.

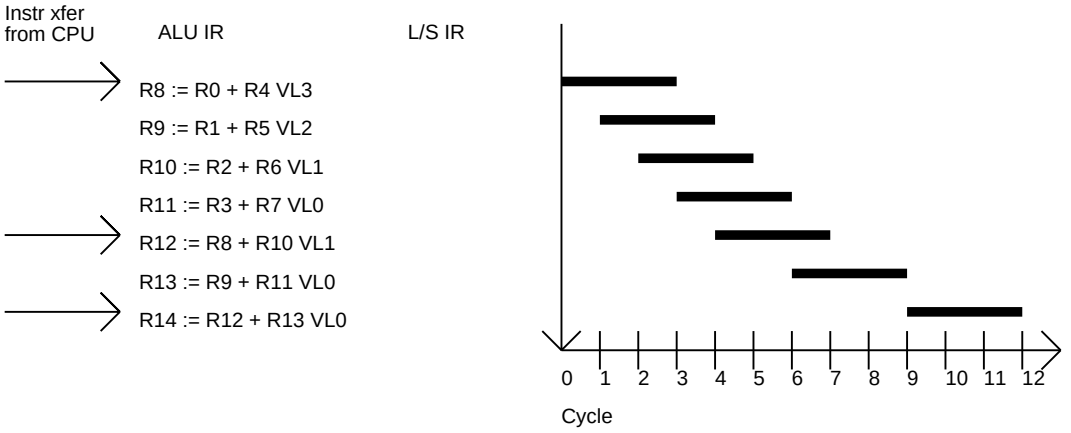

Figure 7: Summing with a tree of vector operations

Since data dependencies are allowed between vector elements, recurrences can also be expressed in vector form. For example, the first 10 Fibonacci numbers (i.e., a recurrence) can be computed by initializing R0 and R1 to 1 (Fib_0 and Fib_1) and executing R2 <- R1 + R0 (length 8) (see Figure 8).

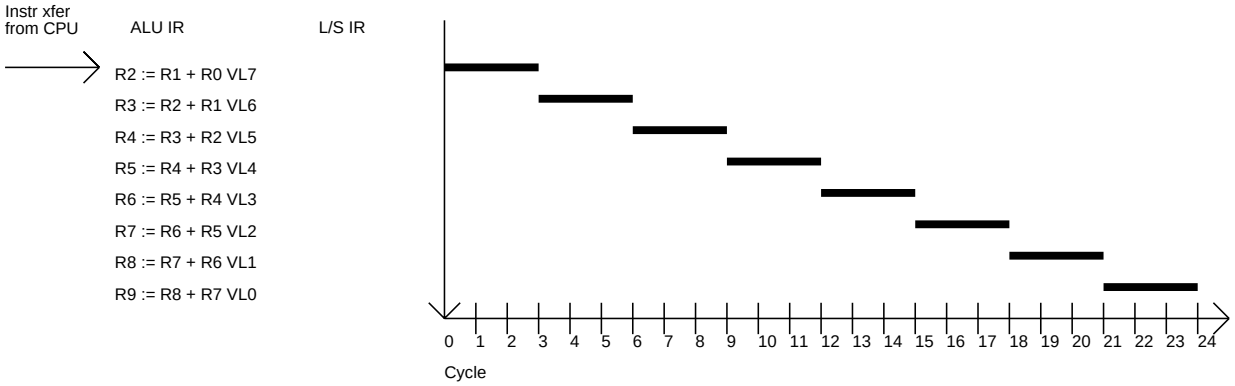

Figure 8: Vectorization of recurrences

2.1.2. Vector Loads and Stores

The FPU registers can be loaded or stored individually, but the MultiTitan does not have vector load or store instructions. Vector register load or store instructions in a virtual memory environment share many problems with multi-word memory references present in CISC machines. For example, the vector load can cross a page boundary, and the machine must save enough state to properly restart it. Vector memory references can result in a significant performance improvement for vectorizable portions of code on machines with large memory bandwidth. However, the MultiTitan has more limited bandwidth than these machines, in keeping with the goal of maximizing average (i.e., scalar) and not peak (i.e., vector) performance. Also, in many applications the most important advantage of vector instructions is the ability to overlap floating-point computations, memory references, and normal loop overhead. In the MultiTitan, this is possible to a large extent without the use of vector memory references. Once a vector arithmetic operation is begun, the CPU is free to issue loads for future computations, stores of previous results, and loop overhead instructions.

For fixed stride applications, the MultiTitan can issue one load per cycle by folding the stride into the load offset (see Figure 9). Combined with the ability to independently issue one FPU ALU operation per cycle during vector operations, this allows a peak issue rate of two operations per cycle. Since the loading and storing of vector elements is performed under program control, full flexibility is retained and operations such as scatter and gather are easily implemented. Vector elements could even be gathered from a linked list with only a doubling of the time otherwise required, even though loads have a one cycle delay slot. This is illustrated in Figure 9. (Loads that follow the linked list alternate between an even and odd temporary register (even^ and odd^) so that the load of the floating-point data can use the pointer concurrent with the load of the next pointer.)

```
Fixed stride c                From a linked list
Load R0,0(base)               Load  even^,0(odd^)
Load R1,c(base)               Load  R0,4(odd^)
Load R2,2c(base)              Load  odd^,0(even^)
Load R3,3c(base)              Load  R1,4(even^)
Load R4,4c(base)              Load  even^,0(odd^)
Load R5,5c(base)              Load  R2,4(odd^)
Load R6,6c(base)              Load  odd^,0(even^)
Load R7,7c(base)              Load  R3,4(even^)
```

Figure 9: Loading of vectors with scalar loads

Traditional vector register banks, where registers are grouped into vectors of fixed length and operated on as a group, reduce the opcode space required to represent instructions but also limit the flexibility of use of the individual registers. For example, in these static allocation schemes, the user cannot select between 8 banks of 64 registers or 64 banks of 8 registers. In MultiTitan, the user can dynamically partition the 52 64-bit registers into any number of 1 to 16-element register groups on an instruction-by-instruction basis. The MultiTitan FPU register file requires 3.3K bits of dual port storage (time multiplexed to be four ports). This easily fits on the same chip as the functional units. Other vector register architectures require much larger amounts of storage. For example, 8 64-element 64-bit registers would require 32K bits of storage, or about ten times that of the unified vector/scalar register file. It is not possible to put 32K bits of multiported register (in which each cell is larger than a SRAM cell) on the same chip as the func-

tional units in today's technologies. Systems with large vector register files thus require off-chip accesses, increasing the latency of operations. The benefits of the small unified vector/scalar register file size will continue into the future. When technology has advanced enough to put a large vector register set on the same chip as the functional units, the unified vector/scalar register set will fit on the same chip as the functional units, the integer portion of the CPU, the instruction buffer, etc. A final benefit of the small register file size is that the context switch cost is smaller than that of traditional vector machines when the vector register state must be saved.

2.2. Low Latency Functional Units

The net performance of a vector operation is a function of its peak performance as well as its latency. If a pipelined functional unit has a latency of l cycles, then it is not operating at peak performance unless its pipeline is filled with l operations. At the beginning and end of every vector operation the functional unit will be operating at less than its peak rate. The functional unit will never attain its peak performance if the vectors are shorter than its latency. The *vector half-performance length* ($n_{1/2}$) [3] is the vector length required to achieve half of the maximum performance.

Low latency functional units are essential in the MultiTitan for two reasons. The first is specific to the unified vector/scalar register file of the MultiTitan, and the second is from the applications executed.

2.2.1. Latency Constraints of the Unified Vector/Scalar Register File

In a machine with a unified vector/scalar register file all of the FPU registers must be directly addressable. Thus, there must be a limited number of them in order for 3-operand FPU ALU instructions to be encoded in 32 bits. The 6-bit MultiTitan register addresses form a constraint on the maximum vector size. Since this six bit field is also used to address registers in other coprocessors, the actual FPU register address space is limited to 52 registers. Often the 52 registers are used as six vectors of length 8 and four scalars.

Thus, in order for good performance to be obtained, the vector half-performance length on the MultiTitan ($n_{1/2}$) must be kept to less than 8. The vector half-performance length achieved by the MultiTitan is approximately 4. This is due to the single-cycle load/store latency from the cache and the three cycle latency of FPU ALU operations. This minimum vector length for half-performance is much smaller than the minimum for traditional vector machines, such as the CDC Cyber 205 ($n_{1/2}$=100), array processors such as the ICL DAP ($n_{1/2}$=2048), or even the Cray-1 ($n_{1/2}$=15).

2.2.2. Latency Constraints from the Applications

Low latency operations are essential for high performance on scalar applications with data dependencies. The latency of operations also determines the minimum vector half-performance length. Many applications will always have very short vectors. For example, 3-D graphics transforms are expressed as the multiplication of a 4 element vector by a 4x4 transformation matrix.

2.2.3. Implementation Latencies

In the MultiTitan FPU the latency of all floating-point operations is three cycles, including the time required to bypass the result into a successive computation. This is very short in comparison to most machines. (Division is implemented as a series of six 3-cycle operations.) The functional units support only double precision floating-point operations, simplifying the design of the functional units. It also enables special cases specific to the double-precision format to be exploited, further reducing functional unit latency.

Each functional unit uses novel structures to reduce the latency of its operations. For example, the add unit uses separate specialized paths for aligned operands and normalized results [2], as well as specialized paths for positive and negative results. The multiply unit uses a novel "chunky binary tree" which is faster in practice than a Wallace tree. The reciprocal approximation unit uses linear interpolation to develop a 16-bit reciprocal approximation. Additional details of the functional unit design are beyond the scope of this paper, but may be found in other documents [4].

As a reference, the latency of various operations in the Cray X-MP (with a 9.5ns cycle time) are compared with the latency of the functional units in the FPU in Figure 10. However, due to the MultiTitan's 4 times slower vector element issue rate, lack of chaining, and less powerful memory subsystem, the peak performance relative to the Cray X-MP can be significantly less than that implied by the latency ratios.

Operation	FPU Latency	X-MP Latency
Addition, Subtraction	120ns	57ns
Multiplication	120ns	66.5ns
Division (via 1/x)	720ns	332.5ns

Figure 10: MultiTitan FPU and Cray X-MP latencies

2.3. Simplicity of Organization

There are two types of control logic in the floating-point architecture: logic that supports both fast scalar execution and vector execution, and logic specific to vector execution. The only vector-specific hardware required by the architecture is three six-bit incrementers for the register specifiers, one four bit decrementer for the vector length, and a very small amount of pipeline control to reissue instructions whose count is non-zero.

2.3.1. Control Logic for Fast Scalar Execution

The FPU control logic provided in the MultiTitan for scalar execution is much simpler than in most high-performance machines. For example, the FPU has a lock step pipeline like the CPU, greatly simplifying the control logic. Also, since all functional units have the same latency, the functional unit write port to the register file need not be reserved or checked for availability before instruction issue.

Vector instructions that overflow on one element discard all remaining elements after the overflow. The destination register specifier of the first element to overflow is saved in the PSW. Note that vector ALU instructions may continue long after an interrupt. For example in the case of vector recursion (e.g., r[a] := r[a-1] + r[a-2]) of length 16, the last element would be written 48 cycles later, even if an interrupt occurred in the meantime.

The FPU control is split into two parts. The first part manages the operation of FPU loads and stores. The second part manages FPU ALU instructions. These two parts of the machine communicate through the register file, the inter-chip pipeline control signals, and the scoreboard.

Central to the scoreboard is a *register write reservation table*. This table consists of one bit for each register in the register file. The bit is set when there is an outstanding operation which will write the associated register. This bit is used to prevent subsequent instructions from reading the register before it has been written. Five ports are required on the register write reservation table every cycle:

- 2 read with source operands for ALU operations
- 1 set for destination upon ALU operation issue
- 1 cleared for destination of retired ALU operation
- 1 read for loads and stores

Of the five required scoreboard ports, all ports are accessed at the same time as their associated data, except for the port that sets a bit on issue of FPU ALU instructions. For example, the ALU source operand reservation bits are read at the same time as the ALU source operands. Moreover, one of the writes is always a set, while the other is always a clear. We will take advantage of these restrictions in the following implementation. This implementation has the advantage that it requires only one extra decoder besides those already required for the register file, and for a single reservation bit the decoder area greatly exceeds that of the RAM cell.

Reservation bits are stored as an extra bit on each word in the register file. The register file R port word line of the extra bit is partitioned into two separate word lines. One segment is controlled by the same word line as the rest of the word. The other is controlled by the destination of the provisionally issued instruction. Since we will never want to write a reservation bit with an arbitrary value, but only set it or clear it, we can do both by single-ended writes. The true bitline can be used to clear a bit at the same time as the complement bit line is used to set another bit.

The FPU uses a distributed result bypass in which each functional unit in the FPU does its own bypassing. If the bypass logic were centralized at the register file, results would have to be put out on the global result bus, then transferred to a global source bus. But since the result bus goes to all functional units, they can select between each source and the result bus based on control signals from the scoreboard. Thus, with distributed bypass logic, the delay from driving the result to the latching of a source is only one global wire delay, not two.

2.3.2. Control Logic for Support of Vector Execution

The reservation of vector result registers is a difficult problem. Three approaches exist:

1. Reserve all elements of the result register at once before issue of the first element.
2. Handle reservations in software.
3. Reserve each element’s result register upon issue of the element.

Note that in traditional vector machines like the Cray-1, vector registers are treated as an indivisible resource and the vector result register reservation problem is simplified to reserving a single resource.

Two difficulties occur if all result registers of a vector operation are reserved at once before issue of the first element's computation. First, special hardware must be provided beyond that required for scalar operations in order to reserve more than one register at a time. Second, additional hardware must be provided to check for prior reservations on all result registers simultaneously. Otherwise the vector register reservation may reserve an already reserved register, in which case the second reservation will be lost on the retiring of the first reservation. One solution would be to compute the remaining source and result register ranges of in-progress vector instructions each cycle, and compare load and store register operands against these ranges before issuing them. This would add a fair amount of hardware, and is the approach taken in the recently announced Ardent Titan.* [6]

Reservation of vector result registers could also be handled by code reorganization. In most machines, floating-point operations have relatively long latencies (e.g., 7-30 cycles). This coupled with potentially long vector lengths makes the scheduling of operations well enough to prevent insertion of NOP's very unlikely. Although all operations in the MultiTitan FPU have a three cycle latency, and the maximum vector length is 16, vector recursion can yield vector execution times of up to 48 cycles. This is still far too long to pad with NOPs. Instead we must rely on a hardware reservation mechanism.

Reservation of individual vector element result registers upon issue of each element can provide hardware interlocks with very low cost. This is the approach used by the MultiTitan. By reserving result registers at the issue of each element, the reservation hardware already in place for scalar operations can be used. Unfortunately this causes a synchronization problem: while the elements of the vector are issuing one by one, a load or store instruction may issue and retire. In particular, the register operand of the load or store may be the same as a source or result register operand of a vector element which has not already issued, but whose vector instruction was issued before the load or store. In order to prevent out-of-order execution (with non-deterministic results), execution constraints must be placed between the vector instruction and any following loads and stores that issue before every element of the vector has issued.

If dependencies occur between loads and stores or elements in a vector other than the first, the compiler must break the vector into smaller vectors so that the normal scalar interlocks are effective. However, in most code sequences this will not be necessary: for example, if a vector operation is followed by stores of each result register, the stores can be performed in the same order as the result elements are produced. Only when operands must be stored out of order, or when the first elements of a vector are not stored but later elements are, will the compiler need to break a vector in order for the normal scalar interlocks to be effective. Note that if an entire vector were required to issue before loads or stores were honored, most useful overlap of transfers and computations would be precluded.

In summary, when the scalar issue logic is used to issue vector instructions, only a very small amount of extra logic (a few counters) is required to support vector execution.

*The Ardent Titan should not be confused with the unrelated DECWRL Titan or MultiTitan.

2.4. Benefits of Additional Vector Support

There are several obvious limits of the performance of this floating-point architecture:

- only 1 FP ALU operation may issue per cycle
- only 1 Load/Store may issue per cycle
- only a total of two operations may issue per cycle

In the MultiTitan concurrent issue is only supported between loads and ALU operations, or ALU operations and stores, but not between multiple operations. Why isn't the simultaneous issue of multiple ALU operations supported? There are two basic reasons.

First, the cost of issuing multiple ALU operations per cycle is very high. Each additional operation would require 3 additional register file ports, and three additional busses. The three busses already present consume about 10% of the chip area.

Second, and more important, the ability to issue multiple ALU operations per cycle would not significantly improve performance. The additional hardware resources would improve peak performance, but not scalar performance. If the basic vector capability already places the machine past the point of diminishing returns, investing heavily to push it even further past is pointless.

This argument also applies to many other suggestions for additional hardware, such as multiple load/store ports to the cache, etc. For example, consider Figure 11, where the overall performance obtained for various degrees of vectorizability is plotted versus the ratio of peak vector performance to scalar performance. This shows that the speedup of two obtained by the basic vector capability in the MultiTitan obtains a significant portion of performance improvement available from vectorization, at least for code with low to average amounts of vectorizable operations. Supercomputers, on the other hand, are biased towards higher peak performance. The ratio of peak vector performance to scalar performance is about 10 for the Cray-1S and the Cray-XMP.

Finally, further increasing the size and complexity of the machine to support higher peak performance is dangerous. If it slows down non-peak performance significantly, the overall performance of the machine can easily be reduced.

3. Simulation Results

To experiment with the MultiTitan vectors, we extended the Mahler [14] intermediate language for our compiler system with a primitive vector capability that corresponds fairly closely to the machine. Vector variables can be declared with a specified constant number of elements, and any consecutive subsection of this vector can be used in a vector operation, provided that the offset and size of the subset is fixed at compile time. Moreover, memory may be referenced directly as a vector with a size and stride fixed at compile time.

The usual scalar floating-point operations may be performed on two vectors of the same length, or on a vector and a scalar. We also added an operator that summed a vector, by performing a vector sum to add its two halves and then doing the same thing to the resulting smaller vector, until left with one or two scalar additions. Loads and stores of the vector variables could be specified by an assignment from or to a memory vector, respectively. Such an assignment

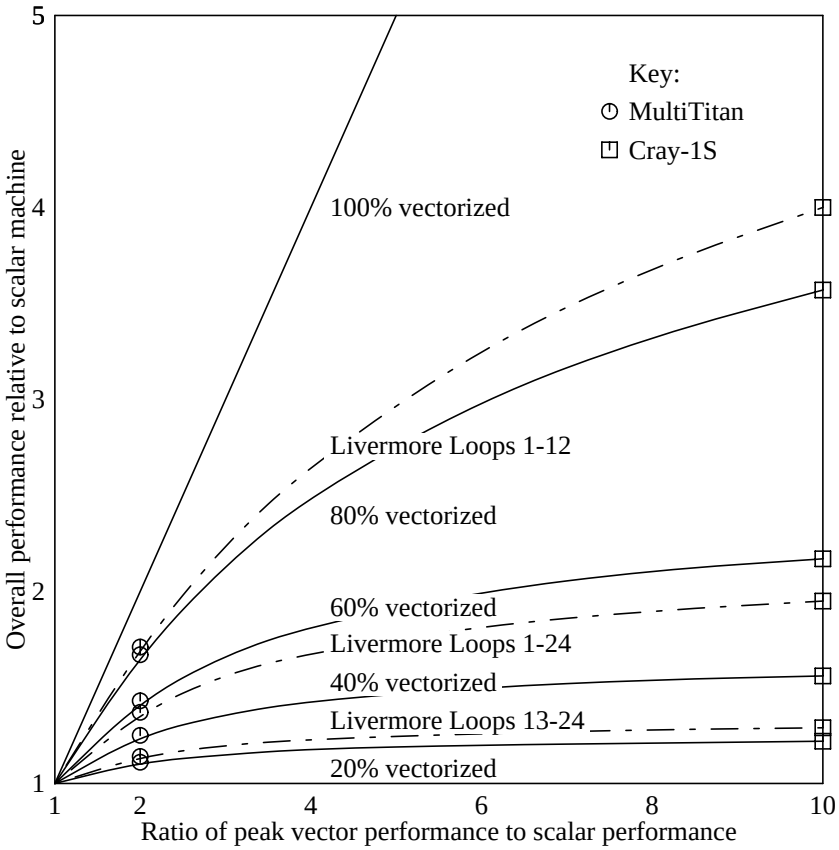

Figure 11: Potential vector performance obtained

was implemented by a series of loads or stores. Memory vectors could also be used directly as operands or destinations, in which case the values would be loaded into or stored from a series of registers not used for vector variables.

Each vector mapped directly to a group of registers. Registers were allocated on a per-procedure basis, on the assumption that they would be used only for the duration of an inner loop. If the total amount of space needed for the declared vectors and temporaries was too large, a compile error was raised. In most cases this meant that our vector operations had lengths of 4 or 8.

These primitives were then used to manually recode the benchmarks we studied. The recoded benchmarks were then simulated on an instruction-level simulator. The Mahler code was intended to represent what the compiler would generate from a program written in an extended version of Modula-2 that provided vector primitives, not what could be produced from the original FORTRAN sources. Our strip-mining followed standard techniques [8]. (Strip-mining is the process of dividing a vector computation of possibly indeterminate length into a loop that performs independent vector computations of a fixed length. Code also must be provided to compute any remainder of the original vector computation not handled by the loop.) Register allocation was done by checking lifetimes of subexpressions, which gave the number of vector values live at any point in the code. Knowing that value and the number of registers on the FPU allows a compiler to choose vector lengths. The Mahler code was produced without extensive hand optimization other than induction variable analysis, strip-mining, and careful vector register allocation.

Because the vectors had to be short, we were not hampered by the fact that they also had to be of fixed length. When a loop could not be unrolled an integral number of times, the leftover was always small. When the leftover was of known size, it could be done as a shorter vector operation. However, even when it had to be done as a scalar loop, it was still fast because the scalar operations are themselves fast.

3.1. Graphics Transform

This section describes a graphics routine to transform a point by multiplying a vector by a transformation matrix. It is representative of many possible applications for the FPU. The problem and register allocation are given in Figure 12. Assume that many points will be transformed by one matrix. Thus the transformation matrix will already be loaded into R0..R15. If the transformation matrix is not loaded, this will require an extra 16 cycles (assuming no cache misses).

```
Problem:

[x y z w] * |a11 a12 a13 a14| = [x' y' z' w']
            |a21 a22 a23 a24|
            |a31 a32 a33 a34|
            |a41 a42 a43 a44|

Register allocation:

[R32 R33 R34 R35] |R0 R4 R8  R12| = [R36 R37 R38 R39]
                  |R1 R5 R9  R13|
                  |R2 R6 R10 R14|
                  |R3 R7 R11 R15|
```

Figure 12: Graphics problem and register allocation

For each element of the initial point vector we will load it and issue a vector floating-point multiply of the element by a column in the transform matrix, resulting in a total of 16 result elements. Once these multiplications have been issued, we will start adding together rows of the 4x4 result elements. Each row is added together in a binary tree, and the four trees are summed in parallel using four element vectors. Finally we will store the result vector. Figure 13 gives the code sequence and cycle timings for this routine. Each instruction requires one cycle, with two exceptions. First, back-to-back stores require two cycles. Second, arithmetic operations cannot issue until a previous vector operation has completely issued all of its elements at a maximum rate of 1 element per cycle. There is only one scoreboard stall for data dependencies in the routine. It is assumed that there are no instruction buffer misses during the routine. This example has been run on the MultiTitan simulator and achieves 20 MFLOPS. The total latency in this example is 35*40ns cycles (1.4us), for double precision computations. This performance is better than that often provided by special-purpose graphics hardware [4].

```
Solution:                          Cycles:
/* load and multiply initial vector. */
R32:=(x);                            1
R[16..19]:=R32*R[0..3];              1
R33:=(y);                            1
R[20..23]:=R33*R[4..7];              3 (issue busy)
R34:=(z);                            1
R[24..27]:=R34*R[8..11];             3 (issue busy)
R35:=(w);                            1
R[28..31]:=R35*R[12..15];            3 (issue busy)

/* sum products in parallel binary trees. */
R[16..19]:=R[16..19]+R[20..23]       4 (issue busy)
R[24..27]:=R[24..27]+R[28..31]       4 (issue busy)
R[36..39]:=R[16..19]+R[24..27]       4 (issue busy)

/* store result vector. */
(x'):=R36;                           3 (wait for result)
(y'):=R37;                           2
(z'):=R38;                           2
(w'):=R39;                           2
Total latency:                      35
```

Figure 13: Code and timing for graphics transform

3.2. Livermore Loops

The simulation results obtained for the Livermore Loops running on the MultiTitan are shown in Figure 14. The performance of each loop was highly dependent on whether the data referenced by the loop was present in the cache. The performance figures for the warm cache were obtained by running the loops twice, thus preloading the code and the data. The numbers shown for the cold cache performance assume that both the instruction and data caches are empty at the start of the simulation. The performance of the cold cache simulations is quite low compared to the warm cache numbers, by factors of about three to six. Because the MultiTitan lacks the pipelined memory access of the Cray, its performance suffers greatly from cache misses. The actual cache miss rate depends on the size of the cache and the size of the data set. Studies of some scientific workloads indicate that good cache hit ratios (much greater than 90%) can be obtained [10, 11], so we expect numbers closer to the warm cache numbers for real programs.

The primary bottleneck keeping the MultiTitan from obtaining higher performance in these benchmarks is its limited memory bandwidth. Even when a cache hit is made, only one load can issue per cycle, and stores can only issue every other cycle. For a two-operand vector add this requires about 4 cycles per result - two loads, a compute, and then a partially overlapped store. (Stores take two cycles, but half of the time for the stores can be overlapped with the computation). However, since memory bandwidth in excess of one operand per cycle is very expensive and primarily improves peak performance, the current design seems to be very cost-effective. The benchmarks that do better than 4 cycles per result do so because operands can be kept in the registers and used multiple times across a single vector expression. For this reason,

loop	MultiTitan cold cache	MultiTitan warm cache	Cray -1S (from [5])	Cray X-MP & [12])
1*	4.3	19.0	68.4	164.6
2*	2.8	17.3	16.4	45.1
3*	2.8	17.3	63.1	151.7
4*	2.3	14.5	20.6	65.9
5	2.0	8.0	5.3	14.4
6*	3.4	5.2	6.6	11.3
7*	6.9	23.4	82.1	187.8
8*	6.0	19.9	65.6	145.8
9*	3.6	20.3	80.4	157.5
10*	1.5	7.1	28.1	61.2
11	1.7	6.6	4.4	12.7
12*	1.4	7.9	21.8	74.3
harmonic mean				
1-12	2.5	10.8	14.4	35.8
13	1.4	1.8	4.1	5.8
14	2.6	3.1	7.3	22.2
15	1.5	1.6	3.8	5.2
16	2.3	2.5	3.2	6.2
17	4.0	4.9	7.6	10.1
18*	7.4	14.8	54.9	110.6
19	2.6	4.2	6.5	13.4
20	4.5	4.7	9.6	13.2
21*	15.9	21.4	32.8	108.9
22*	2.4	2.7	39.9	65.8
23	3.0	7.4	10.4	13.9
24	1.1	1.6	1.6	3.6
harmonic mean				
13-24	2.4	3.2	5.6	10.0
harmonic mean				
1-24	2.5	4.9	8.0	15.6

* indicates loop vectorized on Cray

Figure 14: Uniprocessor Livermore Loops (MFLOPS)

even some of the cold cache performance figures are good, particularly Livermore loops 1 through 3 and 7 through 9. In these loops a performance improvement of at most 25-33% would be gained from additional load and store bandwidth. Note that the warm cache MultiTitan had better performance than the Cray-1S on Livermore Loops 5 and 11, which were not vectorized on the Cray.

Livermore Loops 13-24 in general have larger and more complex kernels than loops 1-12. The difference in performance between the cold cache numbers and the warm cache results were less than that for the first set of loops. This is because the later loops contain more branching and index calculations, so the relative data bandwidths were less, and hence the effects of cache misses diluted. Due to the complexity of the later loops, loops 13, 15, 17, 19, 20, 22, and 23

were coded in Modula-2 instead of Mahler. Since the loop induction-variable elimination was not effective in Modula-2, the performance of these loops on the MultiTitan would improve for more sophisticated compiler technology. Also, because of the complexity of the procedures, some routines written in Mahler are not well-tuned. Loops 14, 16, and 18 suffer the most in this regards. The performance of the MultiTitan on loop 22 is the worst in proportion to the numbers for the Crays. This is because it contains an exp() call and is vectorized by the Cray, but the MultiTitan version is implemented with a scalar subroutine call.

Overall, the warm-cache MultiTitan performance was about one-half that of the Cray 1-S and about one-third that of the Cray X-MP.

3.3. Linpack

Linpack has been run on the MultiTitan simulator. The scalar Linpack performance obtained was 4.1 MFLOPS, while the vector Linpack performance obtained was 6.1 MFLOPS. The scalar performance is approximately 25 times the performance of a VAX 11/780 with FPA. However, the vector performance is only 1/4 that of the Cray 1-S Coded BLAS and 1/8 that of the Cray X-MP [1]. The MultiTitan vector performance for Linpack is worse in relation to the Cray than for the Livermore Loops because Linpack has a higher degree of vectorization and increased memory bandwidth requirements in comparison to the Livermore Loops.

4. Conclusions

The MultiTitan unified vector/scalar floating-point architecture is a very powerful yet simple and cost-effective architecture. This architecture emphasizes improved scalar performance while broadening the applicability of vectorization. This results in higher and more cost-effective overall performance for most applications than emphasizing peak vector performance. Only an insignificant amount of additional hardware (a few incrementers) is required to provide a vector capability. Rather than issuing vector operations as a whole, each vector element is issued with the existing scalar issue hardware. This enables reduction operations and recurrences to be expressed as vectors, unlike most traditional vector machines.

The unified vector/scalar register file, coupled with low latency functional units, allows a much smaller register file to be sufficient for peak performance compared to traditional vector register machines. Thus the unified vector/scalar register file can easily fit on the same chip as the functional units in today's CMOS technology. (In the next CMOS technology they could easily fit on the CPU chip.) The pipeline control is extremely simple. For example, all floating-point operations take the same amount of time, simplifying the scoreboard logic.

Separate Load/Store and ALU instruction registers provide load/store bandwidth that is well balanced with the computation bandwidth. This also enables execution of two operations per cycle during vector execution. Sustained execution rates of 15 double-precision MFLOPS with vectorization and 7 MFLOPS without vectorization are attainable for many applications with current CMOS technology.

5. Acknowledgements

The authors wish to thank Michael L. Powell for helpful discussions, his encouragement, and Linpack simulations. Mike Nielsen contributed the graphics transform simulations. Mary Jo Doherty, John Ousterhout, and Richard Swan provided valuable comments on an early draft of this paper.

References

[1] Dongarra, Jack J.
Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment.
Computer Architecture News 16(1):47-69, March, 1988.

[2] Farmwald, P. Michael.
On the Design of High-Performance Digital Arithmetic Units.
PhD thesis, Stanford University, 1981.

[3] Hockney, R. W., and Jesshope, C. R.
Parallel Computers.
Adam Hilger, 1981.

[4] Jouppi, Norman P., Dion, Jeremy, Boggs, David, and Nielsen, Michael J. K.
MultiTitan: Four Architecture Papers.
Technical Report 87/8, Digital Equipment Corporation Western Research Lab, April, 1988.

[5] McMahon, F. H.
The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range.
Technical Report UCRL-53745, Lawrence Livermore National Laboratory, December, 1986.

[6] Miranker, Glen S., Rubinstein, Jon, Sanguinetti, John.
Squeezing a Cray-Class Supercomputer into a Single-User Package.
In *Compcon Spring '88*, pages 452-456. February, 1988.

[7] Padegs, A., Moore, B. B., Smith, R. M., and Buchholz, W.
The IBM System/370 Vector Architecture: Design Considerations.
IEEE Transactions on Computers 37(5):509-520, May, 1988.

[8] Padua, David A., and Wolfe, Michael J.
Advanced Compiler Optimizations for Supercomputers.
Communications of the ACM 29(12):1184-1201, December, 1986.

[9] Perron, R., and Mundie, C.
The Architecture of the Alliant FX/8 Computer.
In *Compcon Spring '86*, pages 390-393. March, 1986.

[10] Smith, Alan J.
Cache Memories.
ACM Computing Surveys 14(3):473-530, September, 1982.

[11] So. Kimming, and Zecca, Vittorio.
Cache Performance of Vector Processors.
In *The 15th Annual Symposium on Computer Architecture*, pages 261-268. IEEE Computer Society Press, May, 1988.

[12] Tang, J.H., and Davidson, Edward S.
An Evaluation of Cray-1 and Cray X-MP Performance on Vectorizable Livermore Fortran Kernels.
In *1988 International Conference on Supercomputing*, pages 452-457. July, 1988.

[13] Theis, D. J.
Vector Supercomputers.
IEEE Computer 7(4):52-61, 1974.

[14] Wall, David W., and Powell, Michael L.
The Mahler Experience: Using an Intermediate Language as the Machine Description.
In *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 100-104. IEEE Computer Society Press, October, 1987.

[15] Wallach, S.
The CONVEX C-1 64-bit Supercomputer.
In *Compcon Spring '86*, pages 452-457. March, 1986.

[16] Worlton, Jack.
What is the Right Benchmark for Your System?
Supercomputing Review 1(2):16-23, December, 1988.

WRL Research Reports

''Titan System Manual.''
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

''Global Register Allocation at Link Time.''
David W. Wall.
WRL Research Report 86/3, October 1986.

''Optimal Finned Heat Sinks.''
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

''The Mahler Experience: Using an Intermediate Language as the Machine Description.''
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

''The Packet Filter: An Efficient Mechanism for User-level Network Code.''
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

''Fragmentation Considered Harmful.''
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

''Cache Coherence in Distributed Systems.''
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

''Register Windows vs. Register Allocation.''
David W. Wall.
WRL Research Report 87/5, December 1987.

''Editing Graphical Objects Using Procedural Representations.''
Paul J. Asente.
WRL Research Report 87/6, November 1987.

''The USENET Cookbook: an Experiment in Electronic Publication.''
Brian K. Reid.
WRL Research Report 87/7, December 1987.

''MultiTitan: Four Architecture Papers.''
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

''Fast Printed Circuit Board Routing.''
Jeremy Dion.
WRL Research Report 88/1, March 1988.

''Compacting Garbage Collection with Ambiguous Roots.''
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

''The Experimental Literature of The Internet: An Annotated Bibliography.''
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

''Measured Capacity of an Ethernet: Myths and Reality.''
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

''Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.''
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

''SCHEME->C A Portable Scheme-to-C Compiler.''
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

''Optimal Group Distribution in Carry-Skip Adders.''
Silvio Turrini.
WRL Research Report 89/2, February 1989.

''Precise Robotic Paste Dot Dispensing.''
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

''Simple and Flexible Datagram Access Controls for Unix-based Gateways.''
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

''Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.''
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

''Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.''
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

''A Unified Vector/Scalar Floating-Point Architecture.''
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

''Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.''
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

''Integration and Packaging Plateaus of Processor Performance.''
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

''A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.''
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

''Leaf: A Netlist to Layout Converter for ECL Gates.''
Robert L. Alverson and Norman P. Jouppi.
WRL Research Report 89/12, July 1989.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

Table of Contents

List of Figures