TRACE /300 Series:

Technical Summary

MULTIFLOW

TRACE /300 Series: Technical Summary

TRACE /300 Series: Technical Summary

MULTIFLOW

Multiflow Computer, Inc.
31 Business Park Drive
Branford, CT 06405
203–488–6090
800–777–1428

June 1989

Printed in U.S.A.

CHAPTER 1
INTRODUCTION

1.1 TRACE Computer Systems

Multiflow is committed to providing complete solutions to engineering and scientific computing needs. Multiflow's TRACE /300 series systems are a family of general purpose computers, suitable for a wide variety of compute-intensive applications and environments. Major application areas include engineering analysis and simulation of solids, fluids, plasmas, and electronics; computational chemistry applications from *ab initio* calculations through molecular mechanics and dynamics; signal and image processing computations; and geophysical computations. TRACE systems are built on an underlying technology which permits their high peak performance to be delivered on a much wider range of algorithms and applications than traditional vector and multiprocessor supercomputers; all problem domains in which computation time is an issue are likely to be well-served by TRACE systems.

The /300 series departmental supercomputers extend Multiflow's commitment to a new standard of performance, and provide supercomputer performance without supercomputer costs. They offer:

- A large fraction of the CPU, memory, and I/O performance of top-end supercomputers
- Departmental computer system acquisition, maintenance, and environmental costs
- Top-end supercomputer performance without application code restructuring
- Extensive internal parallel hardware for low-cost, high-speed execution on all applications: scalar, vector, and parallel codes
- Full-featured UNIX and VMS user, programming, and networking environments;
- Supercomputer performance in an air-cooled, highly reliable, trouble-free system.

Like their predecessors, the members of the /300 series are based on Multiflow's Very Long Instruction Word (VLIW) architecture and Trace Scheduling compacting compiler technology. The TRACE /300 series consists of three field-expandable and upgradable members:

- The **7/300** provides peak performance of 53 million operations per second (MOPs) and 30 million floating-point operations per second (MFLOPS) in full precision. It has a 256-bit instruction word and executes up to 7 operations simultaneously in each machine cycle.
- The **14/300** provides peak performance of 107 MOPs and 60 MFLOPS. It has a 512-bit instruction word and executes up to 14 operations simultaneously in each machine cycle.
- The **28/300** provides peak performance of 215 MOPs and 120 MFLOPS. It has a 1024-bit instruction word and executes up to 28 operations simultaneously in each machine cycle.

1.2 Complete State of the Art Computing

Raw performance alone seldom addresses computing requirements: megaflop ratings and Linpack numbers don't tell the entire story. Performance must be coupled with the right languages, operating system environment, I/O horsepower, and networking connectivity and compatibility to solve computing problems, rather than create computer problems.

TRACE systems provide a balanced, functional environment for solving large problems, not just a fast central processor. Unparalleled memory and I/O bandwidth complements CPU speed. TRACE systems excel where many other computers fail: running the extremely large jobs that are needed to do real work in a research or development environment.

TRACE systems incorporate the state-of-the-art in flexibility and ease of use. Native, rich UNIX and VAX/VMS environments coexist. TRACE/UNIX is an enhanced implementation of 4.3 BSD UNIX, with its wealth of program development tools and utilities; enhancements include performance features for the supercomputer domain and such functional extensions as batch processing and accounting systems. DECLARE offers a fully functional VMS environment, including networking, programming languages, command language, magtape handling, and text editing. Programs running under DECLARE suffer no performance penalty, and can coexist and communicate with programs running in the UNIX environment.

TRACE systems support a range of programming languages including industry standard FORTRAN 77 (with VAX/VMS FORTRAN extensions), C (Kernighan and Ritchie, with proposed ANSI extensions), PASCAL, Common LISP, and ADA. An available high-performance mathematical library speeds development and porting of programs doing a range of vector and matrix computations.

TRACE systems fit easily into your current computing environment; they communicate and exchange data with existing systems from many vendors, and so enhance your current investments in existing systems, software, and user training. TRACE/UNIX features full TCP/IP networking; the Network File System is available as an option. TRACE/DN is a robust implementation of DECnet Phase IV protocols, offering transparent integration of TRACE systems into DECnet networks.

Extensive use of industry standards in all aspects of TRACE system design protects your investment in the future. Multiflow's choices in user environments (native UNIX and VMS compatibility), programming language features (upholding IEEE standards and supporting DEC extensions), data format (binary compatibility with workstations), and I/O subsystems (interfacing to standard VMEbusses), are all widely supported. This allows you to grow your installation over time and select the best computing, visualization, and peripheral equipment from the best suppliers without problems with compatibility or efficiency.

1.3 Easy, Productive Software Porting and Development

Multiflow employs a unique, software-based approach to high performance which allows the TRACE to attain supercomputer execution speeds on scalar as well as vector code, and on standard software packages as well as on user code. Multiflow's compiler technology allows you to port existing software to the TRACE without rewriting it. You don't need to vectorize or parallelize code to receive the TRACE's performance advantage. TRACE systems free you to do what you do best: understand the technical problems you want to solve. They allow you to focus attention on your task, rather than on your tools.

1.4 Software–Driven Performance

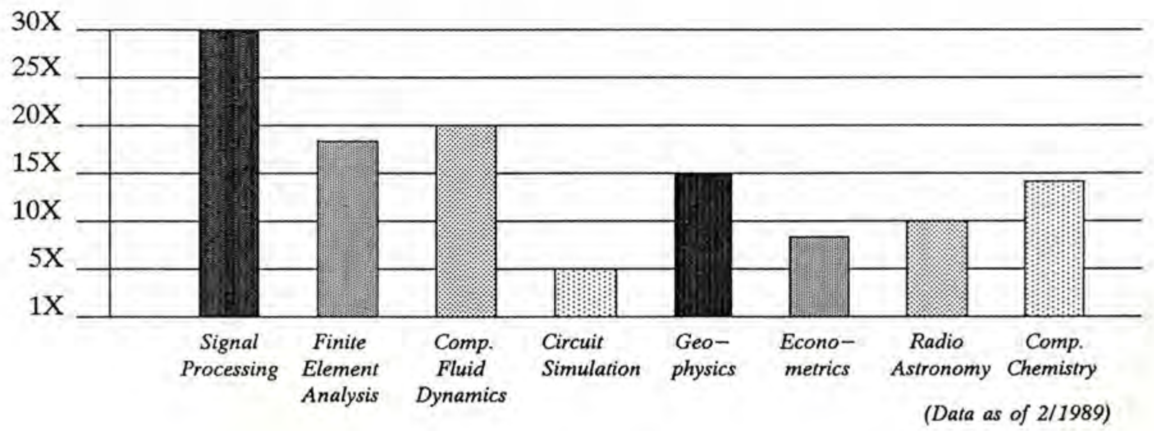
Multiflow's Trace Scheduling compacting compiler technology represents a fundamental shift in emphasis from hardware to software. Earlier approaches to high performance were hardware-dominated: specialized processors were built to solve a certain class of problems (for example, vector processors). To realize any performance gain from such an approach, users were forced to restructure their code into patterns or "chunks" the processor could execute at high speed. This process of adapting code is often called *vectorization* or *parallelization*. Multiflow has chosen a different, software-based approach. Multiflow's sophisticated compiler analyzes programs before they execute to find "fine-grained" parallellism among individual machine operations. It then schedules operations one by one for simultaneous execution on a Very Long Instruction Word (VLIW) processor. The compiler packs many independent operations into each wide instruction word.

Previous vector and multiprocessor approaches to high performance are based on complex instructions (vector operations) and complex hardware (multiprocessor management and synchronization hardware). Multiflow's compiler technology delivers equivalent or better achieved performance with dramatically simpler hardware; in this way, Multiflow brings RISC principles to the world of supercomputing.

Multiflow's approach to concurrent execution is effective; it yields a fourfold to forty-fold advantage over traditional CISC and RISC scalar processors, depending on the problem and the VLIW model chosen. Compared to vector computers running fully vectorized problems, Multiflow's approach always finds more practical concurrency.

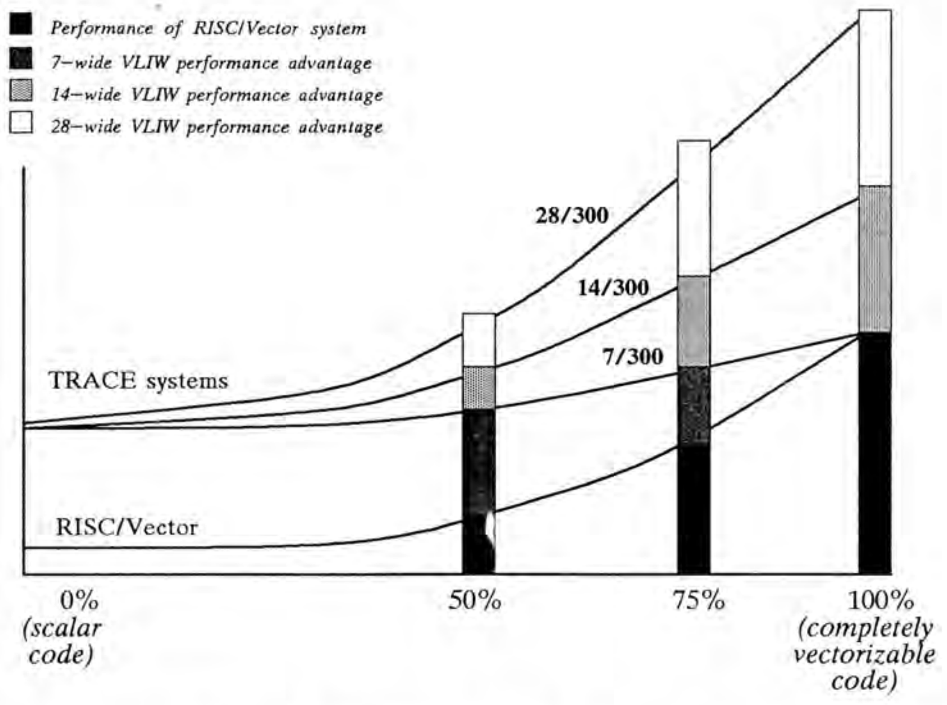
1.4.1 Results

Multiflow's new technologies--the Trace Scheduling compacting compiler, VLIW architecture, and high performance memory and I/O subsystems--produce outstanding results for scientific and engineering computing. TRACE systems are adapted for the large computing tasks and heavy workloads that characterize the research and industrial environments. The chart below shows the TRACE's performance advantage over the VAX 8700 on a number of different applications:



(Data as of 2/1989)

This performance advantage applies to applications that are amenable to parallel and vector techniques (like finite element analysis and computational fluid dynamics) as well as applications that don't vectorize well (like signal processing and circuit simulation). The next diagram compares the performance of the TRACE series with a typical vector processor:



On completely vectorizable code, the 7/300 and the vector system have roughly similar performance. In practice, however, very few applications are completely vectorizable. As the percentage of scalar code increases, the vector processor's performance degrades to the speed of its scalar unit. At 75% vectorization, the TRACE has a significant performance advantage. This advantage is significantly greater when only 50% of the code vectorizes. Larger TRACE systems (the 14/300 and 28/300) provide even better performance.

Multiflow's performance advantage arises because the Trace Scheduling compacting compiler is able to find and exploit fine–grained parallelism even when other forms of parallelism are not available. Because vector parallelism is a subset of fine–grained parallelism, the TRACE's performance equals the performance of a vector machine on vector codes. But vector parallelism is *only* a subset of fine–grained parallelism. On the many programs and parts of programs where vector parallelism is unavailable, Multiflow's compiler still finds enough fine–grained parallelism to deliver significant performance improvements.

1.5 Product Summary

Multiflow's TRACE /300 series includes the following major product features:

- **VLIW CPU**: Multiflow's innovative CPU design employs many features of traditional scientific computers, such as multiple functional units, pipelining, and RISC architectural ideas, while avoiding almost all of their limitations (complex control hardware, algorithm-sensitive performance, etc.).
- **Trace Scheduling Compacting Compilers**: Multiflow compilers take full advantage of the VLIW design by compacting simple machine operations into very-long-instruction-words. This advanced compiler technology results in significant execution speedups without programmer intervention. Available languages include FORTRAN 77 (with VMS extensions), C, PASCAL, Common Lisp, and ADA.
- **TRACE/UNIX Operating System**: Multiflow's implementation of the 4.3 BSD UNIX operating system contains significant functional and performance enhancements, providing the command and interactive environments users have come to expect.
- **VAX/VMS Operating System**: Multiflow's DECLARE Compatibility Suite provides a full VMS-compatible user environment, allowing the TRACE to fit comfortably into existing DEC installations.
- **Memory Subsystem**: Unique cooperation between Multiflow's Trace Scheduling compilers and the TRACE hardware architecture allows the construction of a memory system which can sustain the bandwidth necessary for balanced overall system performance.
- **HPIOP**: The TRACE's I/O design incorporates an independent I/O processor in order to minimize the impact of I/O operations on the CPU (reserving its capacity for computation) while delivering supercomputer level transfer rates to multiple industry-standard VMEbusses.
- **Networking**: The TRACE supports full TCP/IP and DECnet networking, providing connectivity to virtually any other computer system or workstation. NFS provides transparent file access across Ethernet. NQS provides network batch facilities.
- **Applications**: A full range of third party application packages run on TRACE /300 computers.
- **Program Development**: TRACE/UNIX offers a wide variety of program development tools, including enhanced versions of many standard UNIX utilities.
- **System Reliability**: TRACE systems are built for low cost of ownership. With their modest cooling and power requirements, TRACE systems have proven exceptionally reliable, with total system mean time between failures of over 4100 hours.

CHAPTER 2
VLIW ARCHITECTURE

2.1 Introduction

Traditionally, specialized hardware has been used as the means to increase execution speed. Whether the specific strategy involves vector processing units, multiple CPU's, or something else, it's always been taken for granted that more complex hardware structures are the way to performance. Once this decision has been made, it follows that software is secondary; programs will have to conform to the hardware, and performance will depend upon the degree to which they do. This places the burden for optimizing performance on the user. While substantial progress has been made in compilers which automatically vectorize or parallelize raw source code, there remain significant unsolved problems; in practice quite a bit of hand optimization is still necessary to achieve good performance.

Multiflow's approach places software first. A major advance in compiler technology makes it possible for Multiflow's compilers to analyze and schedule your application programs, and then fit them onto a very simple Very Long Instruction Word (VLIW) processor.

VLIW processors incorporate many of the execution hardware features found on more traditional supercomputers, such as interleaved memory and pipelined floating point. However, a VLIW processor has a dramatically simpler approach to the control and management of the execution hardware. Instead of building complex instruction decoding and scheduling hardware, which carries out vector instructions or synchronizes multiple independent processors, Multiflow does it the RISC way. A very long instruction word tells each portion of the computer what to do in each clock cycle, and a new VLIW is fetched each time the clock ticks. This places the entire task of managing the computer on software, and also allows software complete access to the computational hardware. The traditional problems with vector computers, wherein only certain loops can run at high performance, are completely bypassed; all code sequences can make full use of the system's high-performance memory and computational units.

This chapter provides an overview of the architecture. Chapter 3 provides a specific, in-depth look at the Trace /300 implementation, and Chapter 4 looks closely at Multiflow's compilers.

2.2 VLIW Architecture Overview

VLIW computers perform many program steps at once; many of the operations which would be performed one at a time on an ordinary computer are grouped together into Very Long Instruction Words and executed together. Separate fields of the wide instruction word directly control the operation of multiple functional units -- floating adders, floating multipliers, integer units, memory address units. The functional units are essentially the same hardware as might be found in an ordinary sequential computer; however, the long instruction word gives software the ability to use them simultaneously.

One field of the instruction word controls program branching; a single program counter controls the fetching of Very Long Instruction Words. Like a RISC machine, a register file holds operands and results for all computations. LOAD and STORE operations move data between registers and main memory.

For greater throughput without added cost, the functional units are *pipelined*. Each functional unit is designed so that it can start a new computation in every clock cycle. An assembly-line or pipeline handles each stage of complex operations, such as floating point arithmetic. While a single floating-point addition may require 3 cycles (for example) to complete, new adds may be started every cycle. Note that the memory system is also fully pipelined; new references may be started every cycle, although a single reference may require 3 cycles to complete.

VLIW architectures are expandable. More functional units can be added, with multiple register files, communicating via buses. Again, a single program counter and a single flow of control directs the fetching of long instruction words which specify the operation of each functional unit in each cycle. All the functional units run in lockstep, each initiating one operation per cycle as directed by its field of the instruction word. A very wide instruction cache holds the instructions which the machine executes. The instruction word width is not related to the width of the data busses used for computation; it is related to how many functional units there are in the machine.

Note that from one standpoint, VLIWs could be regarded as generalized, more efficient vector machines. The pipelined functional units are essentially the same as might be found in a vector machine, but no separation between scalar and vector hardware exists. Hardware control units which count out vector addresses have been replaced by wide instruction words, which specify each computation uniquely. Control hardware has been replaced with memory. Not only is the hardware lower cost to build, but with appropriate software technology, the computation units can be used much more efficiently.

Viewed from another standpoint, VLIWs provide overlapped execution in the extreme. The instruction words allow the expression of arbitrary execution overlap among scalar operations, with potentially very large numbers of operations executing simultaneously. Operation overlap is completely flexible, within a single CPU: one stream of execution. No runtime synchronization hardware or software is required.

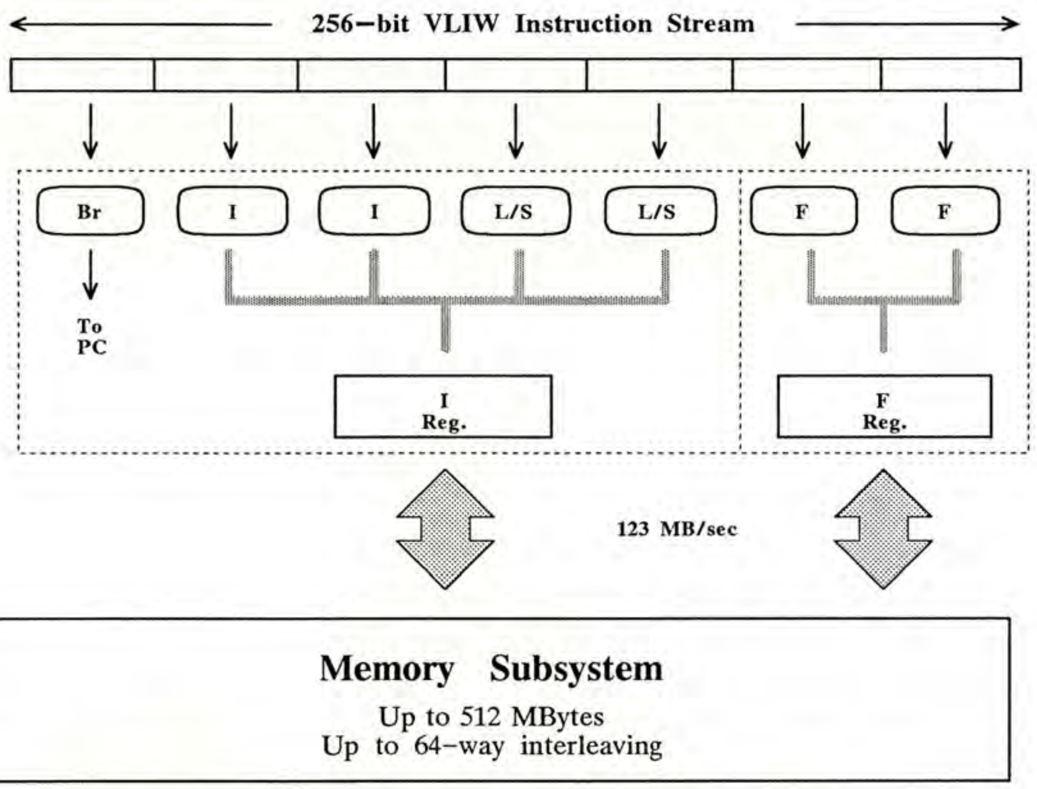
2.3 /300 Series Overview

TRACE /300 systems have been designed as ideal targets for Multiflow's Trace Scheduling compacting compilers. Their design is elegantly simple and, as a consequence, reliable and economical. The compiler's strength allows TRACE systems to deliver high performance from relatively conservative chip technology. The CPU is built primarily from CMOS components, minimizing cooling requirements and power consumption. Although it uses relatively conservative chip technology, its design includes state-of-the-art features:

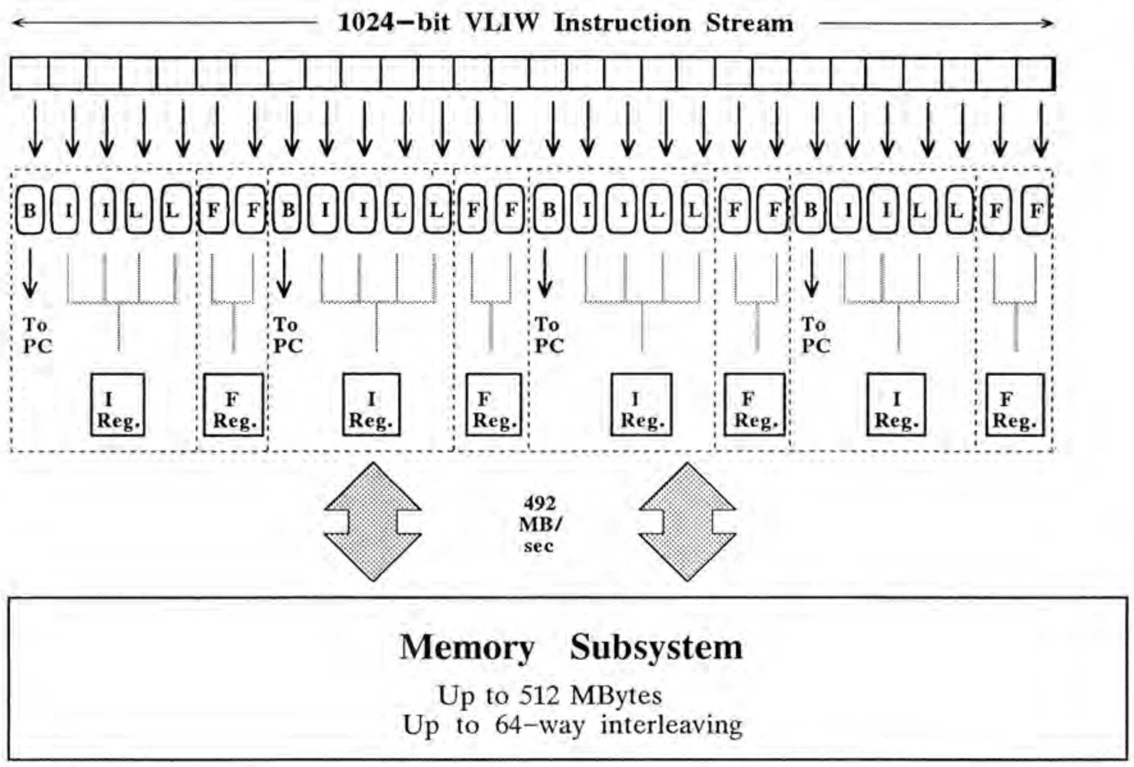
- RISC Architectural Ideas: The /300 Series CPU embodies RISC design principles. There is no microcode: all instructions are directly implemented in hardware. A new instruction exectues every clock cycle. Pipelines are used for "complex" operations (e.g. memory and floating point); the compiler fully schedules the pipelines. The architecture is *load/store;* memory references are explicit, and other operands are taken from registers or immediate constants. The compiler controls allocation of all system resources.
- Multiple Functional Units: TRACE CPUs contain many arithmetic logic units. These multiple computation units execute many simultaneous operations. Operations are scheduled into functional units at compile time, not by specialized hardware at runtime (as in other architectural approaches). Software scheduling keeps the hardware simple and reliable; multiple functional units provide high peak speed.

- Generalized Pipelining: The TRACE /300 series takes full advantages of pipelining, or assembly-line execution techniques, in its design. First introduced in vector supercomputers, pipeline execution allows a fast clock rate and the construction of high-throughput computational units. VLIW architecture controls pipelines in a flexible, one-operation-at-a-time manner. This allows any sequence of computational steps to run at full performance.
- Expandability: Three upward-compatible processor models span a factor of four in peak performance. Upgrades from model to model are easily accomplished in the field by installing extra circuit boards.

The block diagram below shows the structure of the TRACE 7/300 CPU, the smallest member of the TRACE /300 family:



In each cycle, the CPU executes seven operations: one conditional branch operation (Br), two integer operations (I), two memory operations (L/S), and two floating point operations (F). The 256-bit long instruction word contains separate fields to control each operation. This design is inherently expandable; to make a larger, more capable system, add more hardware for computation and lengthen the instruction word. The TRACE 14/300 executes fourteen operations each cycle; the 28/300 (shown in the next diagram) executes 28 operation per cycle:



Note that with all its internal parallelism and many functional units, the 28/300 is still a single CPU. It has one stream of execution; any single job has available to it the power of the entire machine.

2.3.1 Memory and I/O Subsystems

Multiflow's /300 series systems overcome limitations which many other computer systems have in memory and I/O system performance. Software control and hardware simplicity make possible exceptional memory and I/O system capacity and bandwidth. This is valuable for large real-world computations; memory and I/O bandwidth must complement CPU computational speed for overall time-to-solution.

The memory subsystem offers large capacity and high transfer rates. The /300 series has a maximum capacity of 512 Megabytes, and a maximum sustained throughput of 492 Megabytes per second. Main memory is spread across 64 independent memory banks which cycle simultaneously. Each instruction can issue up to eight 64-bit memory references, which are executed in parallel. The TRACE's software-managed interleaved memory system provides the required bandwidth without the limitations of a data cache; full performance is sustainable even when accessing matrices or datasets as large as main memory. By stressing intelligent software over complex hardware, Multiflow provides unequalled memory bandwidth and flexibility.

The TRACE's I/O subsystem delivers the data throughput necessary to support computation at supercomputer speeds. The focal point of the I/O subsystem is the High Performance I/O Processor (HPIOP). The HPIOP manages I/O operations without involving the CPU, employing an intelligent channel protocol to direct I/O operations. The HPIOP takes care of I/O interrupts, device management, error handling, and other low level operations, and makes large block transfers at 246 Megabytes/second to main memory. This I/O architecture provides the bandwidth needed for large computational tasks while reserving the CPU for computation. A TRACE system supports one or two HPIOP I/O subsystems; each HPIOP can support up to two VMEbusses, for a maximum of four VMEbusses per system.

Many enhancements to the TRACE/UNIX operating system allow it to take full advantage the TRACE's I/O and memory performance. TRACE/UNIX supports a per process virtual address space of 4 Gigabytes. It also incorporates many advanced memory management techniques including demand paging, filesystem paging and copy-on-write process creation. These features increase the flexibility as well as the efficiency of memory management. It also supports disk striping, whereby several physical disk drives can be combined into a single logical device, improving I/O performance--a four-way striped disk delivers over 8 Megabytes per second of sustained I/O throughput--and increasing filesystem capacity.

2.4 Performance Through Concurrency

At this point, the obvious question to ask is, "how much of this potential performance can the TRACE actually put to use?" How much work does each instruction do? One simple way of examining this can be had by looking at the standard double-precision Linpack benchmark (100x100, all FORTRAN) on several different computers:

System	**Architecture**	**Clock Speed**	**Linpack MFLOPs**	**Work/Cycle**
		(MHz)	(100 x 100)	(VAX 8800 = 1)
VAX 8800	CISC	25	1	1.0
MIPS M2000-8	RISC	25	3.8	3.8
Convex C-210	Vector	25	16	16.0
IBM 3090/108S VF	Vector	67	16	5.9
Cray Y-MP/832	Vector	160	79	12.3
TRACE 28/300	**VLIW**	**8**	**20**	**62.5**

Data as of 2/1989

The last column reports the amount of work done per cycle, relative to the VAX 8800. To compute this figure, we multiply the Linpack number for each system by the ratio of the VAX's clock rate to the computer's. This cancels the effect of different clock speeds and lets use compare the per-cycle efficiency of each machine. In each cycle, the TRACE 28/300 performs 62.5 times as much work as the VAX 8800, and five times as much work as the CRAY Y-MP. The compiler is the reason for this advantage: by packing many operations into each instruction and taking advantage of the TRACE's pipelines, it overlaps the execution of many operations. This overlap is both "horizontal" (between different operations packed into the same instruction) and "vertical" (between operations at different stages in the pipelines).

Multiflow's TRACE /300 series combines supercomputer performance with simplicity, connectivity, and reliability. It takes advantage of the most important ideas of the past (pipelining and overlapped scheduling) and the most sophisticated ideas of the present (software-driven parallelism and concepts from the RISC architectural paradigm). Multiflow *is* the future of high-performance computing.

CHAPTER 3
TRACE CPU IMPLEMENTATION

The TRACE /300 series is an upgradable family of VLIW architecture computer systems. Members of the /300 series offer peak performance from 53 to 215 MOPS and from 30 to 120 MFLOPS in 64-bit precision. The central processor consists of seven to 28 functional units, which operate simultaneously on a single, synchronous execution stream. The function units include multiple high-performance integer ALUs, floating point ALUs, and memory reference units. The family is designed for integrity and reliability using low-power, reliable 2 micron CMOS VLSI. The floating point units incorporate ECL logic for speed.

The TRACE /300 family features large, high-bandwidth, low-cost main memory, with up to 512 Megabytes capacity and sustained bandwidth of up to 492 Megabytes/second. Memory is demand-paged and virtually addressed, with a 4 Gigabyte virtual address space per process.

Simplicity and efficiency characterize the TRACE CPU. It executes instructions directly, without the overhead of intermediate interpretation or decoding. There is no microcode. All functional units are completely synchronized, controlled entirely by the instruction word; no queues, interrupt mechanisms, or other specialized hardware is needed to move data within the CPU. There are no penalties for synchronization or communications, unlike multiprocessor systems. Simplicity and efficiency translate directly into performance and reliability.

3.1 CPU Structure

The CPU consists of four major component groups: Integer Units, Floating Point Units, the Memory Subsystem, and the I/O subsystem. Integer units and floating point units work as pairs, executing the TRACE's wide instruction words. Each pair is called a *cluster*. Each cluster provides raw computational rates of 30 MFLOPS and 53 VLIW MIPS, and executes up to seven independent operations per clock cycle: four integer operations, two floating point operations, and one conditional branch.

Different TRACE models differ only in the number of clusters that are present: the 7/300 has one cluster, the 14/300 has two clusters, and the 28/300 has four clusters. The length of the instruction changes with the number of clusters that are present. The 7/300 has a 256 bit instruction word, the 14/300 has a 512 bit instruction word, and the 28/300 has a 1024 bit instruction word. Upgrades consist simply of installing additional clusters; for example, to upgrade the TRACE 14/300 to the TRACE 28/300 involves adding two additional clusters (two Integer Units and two Floating Point units), and updating a configuration file.

Five sets of global busses carry data between the major CPU subsystems; each set of busses provides many 32-bit wide data paths. The TRACE's major busses are:

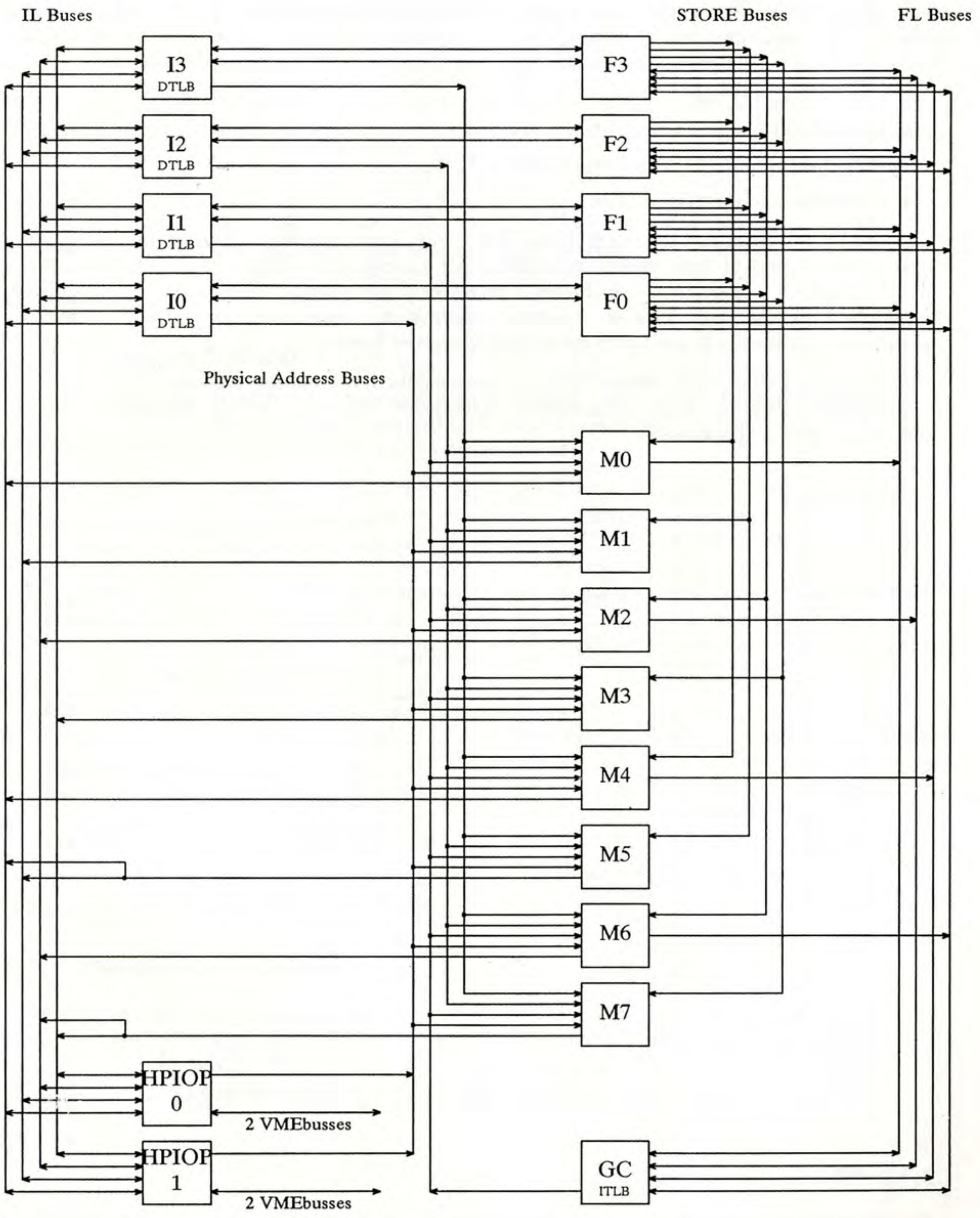
- Four 32-bit busses carry data to the integer units from memory.
- Four 32-bit busses carry data to the floating point units from memory.
- Eight 32-bit busses carry data between integer and floating point units.
- Four 32-bit busses carry data to memory from the integer and floating point units
- Four physical address busses carry addresses to memory.

Separate datapaths carry operands to and from ALU's. In addition to providing a memory bandwidth of 492 Megabytes/second, the TRACE 28 provides 492 Megabytes/second inter–cluster bandwidth (data transfers from one cluster to another), 492 Megabytes/second intra–cluster bandwidth (data transfers between the Integer and Floating Point units within each cluster), 768 Megabytes/second operand bandwidth, and 246 Megabytes/second I/O bandwidth.

Within each cluster, large general–purpose register files route incoming data and provide the connectivity needed to support many simultaneous computations. Each cluster contains 64 32–bit Integer registers, 64 32–bit Floating Point registers, and 32 32–bit "store" registers (an intermediate destination for data en–route to memory). This yields 160 registers on the TRACE 7/300; 320 registers on the 14/300; and 640 registers on the 28/300.

Each cluster executes instructions directly from an instruction cache. The instruction cache contains 8192 instructions. Instruction execution and instruction fetch are pipelined, so that the CPU never needs to wait for instructions to arrive from memory. The instruction cache is distributed between the different CPU modules, so that it expands as additional clusters are added to the CPU. On the TRACE 28/300, the cache's size is 1024 Kilobytes.

The following diagram illustrates these interconnections on a TRACE 28/300.


IL Buses
STORE Buses
FL Buses
I3
DTLB
I2
DTLB
I1
DTLB
I0
DTLB
F3
F2
F1
F0
Physical Address Buses
M0
M1
M2
M3
M4
M5
M6
M7
HPIOP
0
2 VMEbusses
HPIOP
1
2 VMEbusses
GC
ITLB

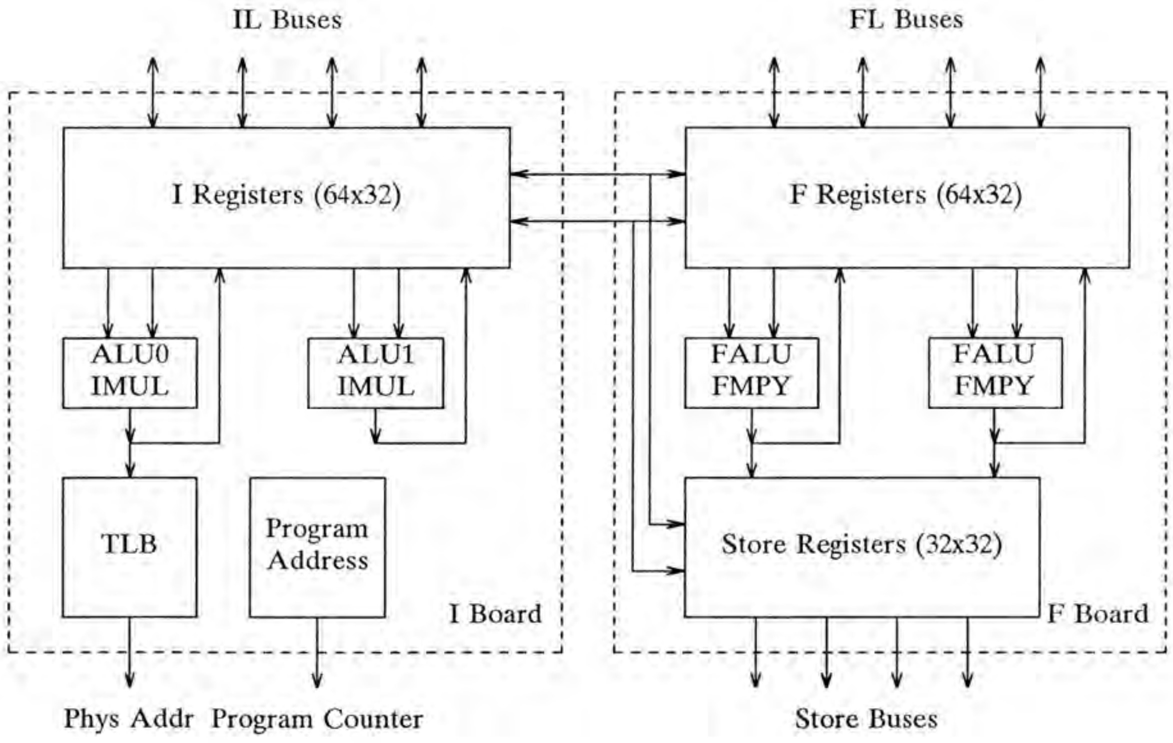
3.1.1 Integer Unit

The figure below shows the structure of the Integer and Floating-Point units. Two integer Arithmetic-Logical Units (ALUs), a Translation Buffer (TLB), and a Program Address generator comprise the Integer Unit. They perform the following functions:

- Integer computation;
- Initiation of memory operations;
- Virtual-to-physical address translation for all data references;
- Conditional branch operations.

Each ALU can execute two operations per cycle (one operation per minor cycle, or "beat"), for a total of four integer operations per instruction. These ALUs perform over ninety primitive operations, including branches, arithmetic operations, comparison operations, logical and bitwise boolean operations, and operating system support operations. All operands are taken from integer register banks.

On each Integer Unit, one integer ALU is responsible for initiating memory references. It is associated with a TLB (Translation Lookaside Buffer), which translates virtual addresses to physical addresses.



3.1.2 Floating Point Unit

The primary responsibility of the floating point unit is executing the floating point operations specified in the TRACE instruction. To fulfill this function, the Floating Point unit has two independent floating point ALUs (FALUs). Each floating point ALU can execute one operation per cycle, for a total of two operations per instruction. The operation set for each FALU contains over ninety primitive floating point operations, including 64 and 32-bit floating point arithmetic (addition, subtraction, multiplication, division, square root), comparison, and type conversion operations. A group of special "multiply/accumulate" operations allow two floating point computations (multiplication and cumulative addition) to be combined in a single operation. Floating point arithmetic conforms to the IEEE 754 standard.

The Floating Point unit in the TRACE /300 series includes low latencies for floating point operations, hardware for square root and division operations. Both floating point ALUs include the full opcode suite, unlike earlier TRACE family processors. These new features yield superior floating point performance.

Together, the integer and floating point units provide the well-balanced computational power that is necessary for general-purpose high performance computing. The ratio of integer to floating point operations per cluster reflects the ratio found in most calculations.

3.1.3 Memory Subsystem

TRACE systems provide per-process virtual address space of 4 Gigabytes per process. Large process-tagged address translation buffers provide excellent virtual memory performance in an environment of many large processes.

TRACE physical memory achieves the consistently high memory performance of large-scale supercomputers by incorporating a software-managed interleaved memory system. The completely synchronous nature of the TRACE CPU allows the compiler to determine, in advance, where and when every memory reference will take place. With this information, the compiler generates code that obeys the memory subsystem's resource restrictions while optimizing speed. Software management takes the place of the hardware bank scheduler, or "stunt box," required to obtain high memory bandwidth in other supercomputer architectures.

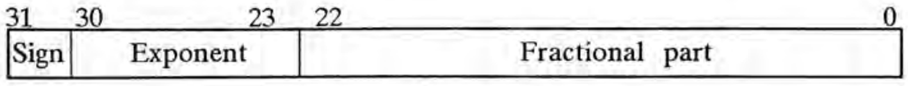
TRACE systems have many independent datapaths to service many memory references simultaneously. Twelve 32-bit busses carry data to and from memory. In a fully-configured system, eight memory controllers, providing 64 interleaved memory banks, service memory references. The memory system can handle up to 8 64-bit memory accesses per cycle, yielding a maximum bandwidth of 492 Megabytes/second. The maximum capacity of the memory system is 512 Megabytes.

The memory subsystem incorporates SECDED error correction hardware. All single bit errors are corrected, and all two-bit errors are detected.

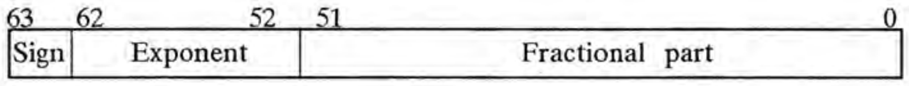
3.2 Primitive Data Types

The TRACE CPU supports four primitive data types: signed and unsigned 32-bit integers, 32-bit floating point numbers, and 64-bit floating point numbers. Byte ordering for all data is compatible with the 68000 microprocessor family and the IBM 370 series; the most significant byte is stored at the lowest address in memory.

The diagrams below show the IEEE representation for floating point numbers:



Single Precision



Double Precision

The table below shows the range of values that can be expressed by different primitive data types:

Type	Minimum	Maximum	Minimum Magnitude
Unsigned Integer	0	4294967295	0
Signed Integer	−2147483648	2147483647	0
32–bit Floating Point	$-3*10^{38}$	$3*10^{38}$	$1.2*10^{-38}$
64–bit Floating Point	$-1.7*10^{308}$	$1.7*10^{308}$	$2.3*10^{-308}$

3.3 TRACE /300 Series Instructions

3.3.1 Primitive Operations

The TRACE operation set includes over 90 primitive integer operations and over 90 primitive floating point operations. The following table summarizes the different operation set families.

	Integer:	*Floating Point:*	
Integer Arithmetic	**add sub mpy**	**add sub mpy div abs max min sqrt**	*Floating Point Arithmetic*
Logical and Bitwise Operations	**and or eqv neqv band bor bxor mv**	**macinit macadd macsub**	*Multiply/Accumulate Operations*
Integer Comparison	**ceq cne clt cle cge cgt ceq cne clt cle cge cgt**	**ceq cne clt cle cge cgt ceq cne clt cle cge cgt**	*Floating Point Comparison*
Select, Field Merge, Field Extract, Shift	**slct mrg ext lsrs**	**merge scale trunc**	*Merge, Scale, Truncate*
Memory Reference	**ld st**	**u32f32 u32f64 s32f32 s32f64 f64f32 f32u32 f64u32 f32s32 f64s32 f32f64**	*Type Conversions*
Flow Control Operations	**br jmp return**		

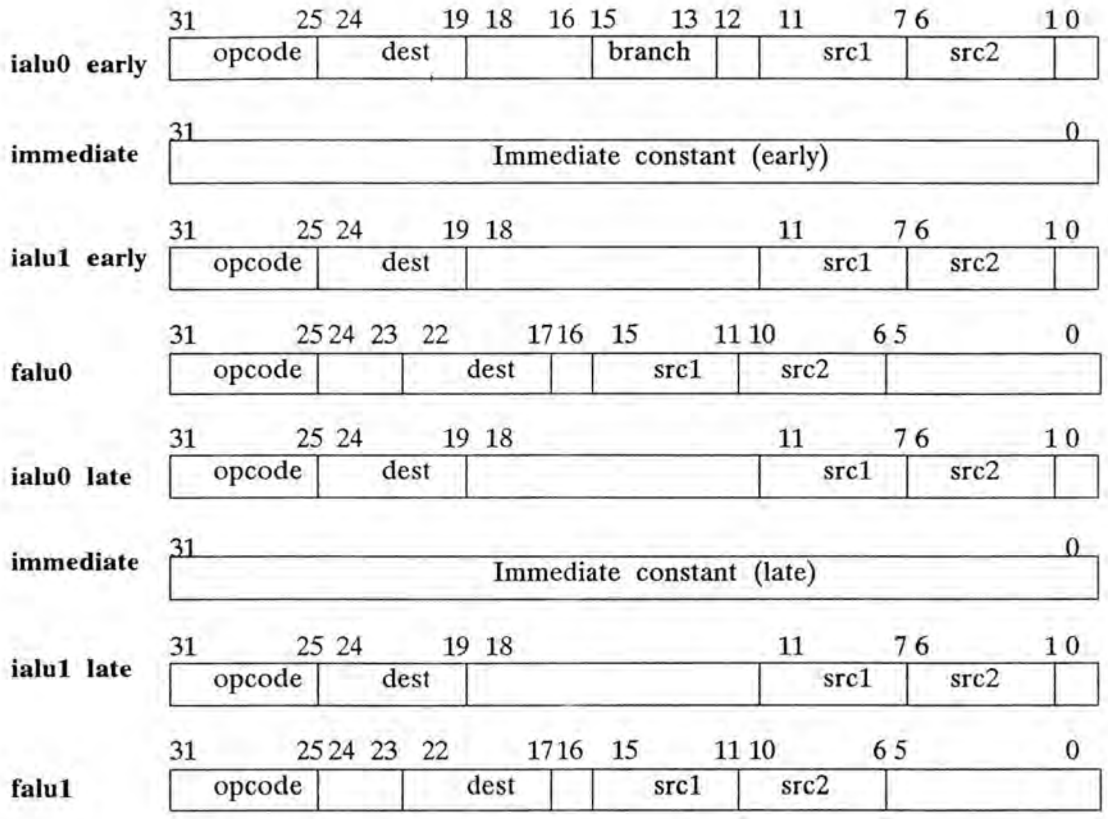
3.3.2 Instruction Format

The diagram below shows the basic instruction word for the TRACE 300 series. This instruction controls one cluster (one I–F pair); it contains fields for two F board operations (one on each FALU), four I board operations (early and late beat operations for each IALU) and two immediate constants (also used for branching). An instruction for the 7/300 consists of one such unit. 14/300 instructions consist of two of these units (16 words total), one controlling cluster 0, the other controlling cluster 1. 28/300 instructions consist of four of these units (32 words total).

In each field, the *opcode* specifies which operation to perform; *src1* and *src2* specify the operand; *dest* specifies the destination for the result; *d_b* specifies the destination register bank; and *btest* controls conditional branching.

The instruction is shown in the format which the TRACE processor stores it in its instruction cache. When stored in main memory or on disk, instructions are stored in a variable–length format to minimize instruction size. TRACE hardware expands instructions into the format below as they are fetched from memory.

TRACE 7/300 Instruction Word



3.3.3 Sample Code

This section discusses a sample of actual code compiled from a FORTRAN source program in order to illustrate the preceding concepts.

```
    DOUBLE PRECISION FUNCTION DOTPROD( N,X,Y )
    DOUBLE PRECISION X(*),Y(*),Z
    DO 10 I = 1, N
10      Z = Z + X(I)*Y(I)
    DOTPROD = Z
    END
```

This routine was compiled using aggressive loop unrolling on a TRACE 14/300. The loop was unrolled automatically 81 times, yielding a body 58 instructions long. As a result, almost every operation available in each instruction was used.

Because of the extensive unrolling, it is not practical to show the entire routine. Here are four instructions from the loop body. They are typical of compiler-generated code for a vector routine. At the beginning of the loop, the code is devoted almost completely to load operations (starting the memory pipelines) and comparisons, which are used for exit tests during the loop. In this part of the loop, the each instruction has eight load operations, using the memory system at its maximum bandwidth. After a few instructions, data begins arriving from memory and the actual computation begins.

The two instructions from the loop's computational core perform two additions (in the first instruction) and two multiplications (in the second). Together, these two instructions are equivalent to two iterations of the original loop's body.

```
       cluster unit  opcode  operands
instr  cl0 ialu0e dld.64 fb0.r32,r1,6#8        --initial setup; loads (to fill
       cl0 ialu1e cgt.s32 li1bb.r3,r32,6#10      registers) and comparisons
       cl0 ialu0l dld.64 fb1.r34,r1,6#24         (for exit tests)
       cl0 ialu1l cgt.s32 li1bb.r4,r32,6#8
       cl1 ialu0e dld.64 fb0.r0,r0,zero
       cl1 ialu1e cgt.s32 li1bb.r3,r32,6#11
       cl1 ialu0l dld.64 fb1.r2,r0,6#16
       cl1 ialu1l cgt.s32 li1bb.r4,r32,6#9;
instr  cl0 ialu0e dld.64 fb0.r36,r1,6#40
       cl0 ialu1e cgt.s32 li1bb.r5,r32,6#6
       cl0 ialu0l dld.64 fb1.r36,r1,6#56
       cl0 ialu1l cgt.s32 li1bb.r6,r32,6#4
       cl1 ialu0e dld.64 fb0.r4,r0,6#32
       cl1 ialu1e cgt.s32 li1bb.r3,r32,6#7
       cl1 ialu0l dld.64 fb1.r4,r0,6#48
       cl1 ialu1l cgt.s32 li1bb.r5,r32,6#5
       cl1 br true and r3 L1?3;
instr  ...;
instr  cl0 ialu0e dld.64 fb1.r12,r7,17#80      --instructions from the center
       cl0 ialu1e cgt.s32 li1bb.r3,r35,6#8       of the loop body, showing
       cl0 falu0e add.f64 lfb.r8,r8,r14          peak computing bandwidth
       cl0 falu1e add.f64 lfb.r42,r50,r42
       cl0 ialu0l dld.64 fb0.r48,r3,6#8
       cl0 ialu1l add.u32 ib1.r38,r38,17#96
       cl1 ialu0e dld.64 fb1.r46,r3,17#88
       cl1 ialu1e cgt.s32 li1bb.r4,r37,6#7
       cl1 falu0e add.f64 lfb.r8,r14,r8        --a pair of additions
       cl1 falu1e add.f64 lfb.r36,r34,r36
       cl1 ialu0l dld.64 fb0.r14,r5,zero
       cl1 ialu1l bor.32 ib0.r36,zero,r36
       cl1 br false or r4 L41?3                --these two operations
       cl0 br true and r3 L42?3;                 are a three-way branch
instr  cl0 ialu0e dld.64 fb1.r14,r5,17#80
       cl0 ialu1e cgt.s32 li1bb.r4,r35,6#6
       cl0 falu0e mpy.f64 lfb.r12,r12,r6       --a pair of multiplications
       cl0 falu1e mpy.f64 lfb.r38,r38,r36
       cl0 ialu0l dld.64 fb0.r6,r5,17#64
       cl1 ialu0e dld.64 fb1.r48,r4,17#88
       cl1 ialu1e cgt.s32 li1bb.r3,r37,6#5
       cl1 falu0e mpy.f64 lfb.r4,r4,r2
       cl1 falu1e mpy.f64 lfb.r38,r38,r40
       cl1 ialu0l dld.64 fb0.r36,r4,17#72
       cl1 ialu1l bor.32 ib0.r5,zero,r38
       cl1 br true and r3 L43?3                --another three-way branch
       cl0 br false or r4 L44?3;
```

3.4 TRACE CPU Specifications

The following table summarizes the major CPU specifications for all members of the the TRACE 300 series. All systems in the series share the Multiflow VLIW (Very Long Instruction Word) architecture.

System:	**7/300**	**14/300**	**28/300**
Architecture			
Technology:	2 micron CMOS VLSI and Advanced Schottky Logic		
Instruction length:	256–bit	512–bit	1024–bit
Operations per Instruction:	7	14	28
Operation rate (MOPS):	53	107	215
Floating Point rate (MFLOPS): (64–bit precision)	30	60	120
General Registers			
Number:	160	320	640
Register Bandwidth (MB/s)	948	1968	3692
Instruction Cache			
Instructions:	8192	8192	8192
Bytes:	256K	512K	1024K
Bandwidth (MB/s):	246	492	984
Error Control:	Single–bit soft–error correction		
Cycle Time (ns)			
Major cycle:	130	130	130
Minor cycle:	65	65	65
Main Memory			
Technology:	256–Kbit or 1–Mbit dynamic RAM		
Capacity (MBytes):	32 to 512	32 to 512	64 to 512
Bandwidth (MB/s):	123	246	492
Virtual address space (GB): (per process)	4	4	4
Interleaving:	16– to 64–way	16– to 64–way	32– to 64–way
Page size (KB):	8	8	8
Error control:	Single–bit error correction; double–bit error detection		
I/O subsystems			
Number:	1 or 2	1 or 2	1 or 2
Number of VME busses:	1 to 4	1 to 4	1 to 4
Memory to subsystem rate (MB/s):	246	246	246
Peak subsystem to device rate (MB/s): (per VME bus)	20	20	20
Maximum disk file size (GB):	8	8	8
Maximum I/O page size (KB):	32	32	32

CHAPTER 4
THE TRACE COMPILER

Multiflow's Trace Scheduling compacting compilers are the heart of its software-based approach to supercomputing. Multiflow's VLIW architecture and Trace Scheduling compacting compiler technology combine to produce supercomputer execution speeds. Multiflow compilers produce this level of performance by compacting machine operations into wide instruction words, without requiring programmer intervention.

The compiler operates by exploiting *fine-grained parallelism*: individual computational steps which can be executed at the same time. This parallelism exists in all programs, and the compiler finds it after the source code has been translated into machine operations. It does not depend on the structure of the original source code, and so programmers need not write code in any particular forms to achieve good performance. Vectorizable and parallelizable code form a subset of the code that Multiflow compilers exploit.

Multiflow's technology is uniquely able to deliver parallelism without user involvement because VLIW overlapped execution is below the level of the high-level programming language. Optimization and compilation proceeds automatically, without requiring user guidance or special programming techniques.

Multiflow currently supports a number of popular programming languages including FORTRAN, C, PASCAL, Common LISP, and ADA. All compilers share the same Trace Scheduling compaction technology. The compilers are designed to conform to industry standards in order to minimize porting requirements and system dependence. Language specifications may be found at the end of this chapter.

4.1 Using TRACE Compilers

Using compilers on the TRACE is no different than using compilers on any VAX/VMS or UNIX system. The command line and user interface to the compiler are entirely standard. The material described later in this chapter, the internal workings of the compiler, is entirely hidden from the user.

The command interface gives the programmer complete control over the compilation process, from both the Unix and VMS-compatible command shells. User switches include:

- Compilation Mode: ranging from *checkout* mode, which rapidly compiles programs without optimization; to *debug* mode, which retains full symbolic information in the object file for the debugger; to *optimized* mode, which performs the optimization and compaction described in this section.
- Optimization Level: ranging from -O1, selecting minimal optimization but full compaction, to -O4, including heavy loop unrolling and automatic inlining of subroutines, for peak performance.
- Compatibility: programs can be checked for compatibility with FORTRAN-66 and for a variety of other constructs which might be misinterpreted on a UNIX system.
- Performance: programmers can request various performance-enhancing actions, such as memory reference alignment and automatic conversion to double precision.

- Preprocessing: various preprocessors may be applied to source files prior to compilation. These may be used to compile different source versions depending on external conditions, and to generate and compile different versions of a program from a single set of source files.
- Cross References: the compiler can produce a cross reference listing if desired.
- Program Analysis: programs may be compiled to produce extra data for use by various performance monitoring utilities.

4.2 Compiling for a VLIW Computer

TRACE computers deliver their performance by executing many independent operations simultaneously. To compile code for this sort of computer, the compiler breaks the program down into long chains of basic operations (*traces*), determines which operations can be executed together, and packs them efficiently into the TRACE's wide instructions. The process is known as *compaction*.

To pack operations efficiently, the compiler rearranges the program's basic operations wherever necessary to get the most out of each instruction, provided that reordering doesn't change the program's behavior. Two constraints guarantee that the compiler will always produce correct code:

- The compiler must observe data precedence; it cannot schedule an operation until the data needed to perform the operation is ready. For example, the operation C=A+B must not take place until the operands A and B are available.
- The compiler must observe resource constraints. For example, a TRACE 28/300 instruction has room for eight memory operations. The compiler therefore cannot pack 9 or more memory operations into any instruction. Other resource constraints include rules about bus usage and memory usage.

To see how the compiler performs these operations, let's consider a simple example. Later in this chaper, we will discuss the techniques used to analyze, schedule, and generate code in detail. Assume that we want to compile the following two lines of FORTRAN on the TRACE 7/300, with seven operations per instruction:

```
C = ( A*2 + B*3 ) * 2*I
Q = ( C+A+B ) - 4*( I+J )
```

In addition, assume that it takes three cycles for data to arrive from memory, and that all other operations take a single cycle. First, the compiler decomposes these statements into a sequential chain of basic operations. The figure below shows this chain of sequential operations on the left. Next, it analyzes the dependencies between these operations, and creates a *dependency graph*, such as the one shown in the center of the diagram.

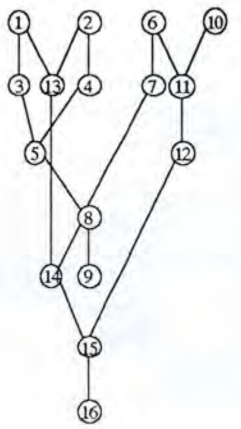
FORTRAN Code:

```
C = (A*2 + B*3) *2*I
Q = ( C+A+B )-4*(I+J)
```

Sequential Code:

```
[1]  LD A
[2]  LD B
[3]  t1 = A*2
[4]  t2 = B*3
[5]  t3=t1+t2
[6]  LD I
[7]  t4 = 2*I
[8]  C = t4*t3
[9]  ST C
[10] LD J
[11] t5 = I+J
[12] t6 = 4*t5
[13] t7 = A+B
[14] t8 = C+t7
[15] Q = t8-t6
[16] ST Q
```

Dependency Graph:



Wide Instruction Words:

LD 0	LD 1	Int 0	Int 1	FP 0	FP 1	Branch
LD A	LD B					
LD I	LD J					
				A*2	B*3	
		2*I	I+J	t1+t2	A+B	
		4*t5		t4*t3	C+t7	
ST C				t8−t6		
ST Q						

Finally, the compiler packs the operations into a sequence of wide instruction words, using dependency relationships and information about the computer to guarantee · that result will execute correctly. In this case, it packs two LOAD operations into the first instruction. It then waits three cycles before scheduling A*2 and B*3, because it takes three cycles for the memory pipeline to deliver this data from memory. When it has finished, we have a sequence of eight wide instruction words, which take eight cycles to execute.

How does this compare with the performance of the same code on a sequential computer? Assuming that the convential computer takes the same amount of time to execute any individual operation (three cycles for a load or a store; one cycle for everything else), and has the same set of basic operations, this code would require 28 cycles. In this case, the VLIW computer has an advantage of over three-to-one.

Furthermore, the sixteen basic operations fill under half of the available slots in the sequence of seven wide instructions. If we had more code to compile, we could put these additional fields to use. Let's assume that the next statement in the program is Z=M+N*2. This requires five more sequential operations, and increases the sequential computer's execution time to 39 cycles; however, it doesn't require any more wide instruction words. In this case, the VLIW has an advantage of almost five-to-one. And there is room to go futher. In general, the longer the chain of operations the compiler can compact at once, the more efficient and effective code it will be able to generate.

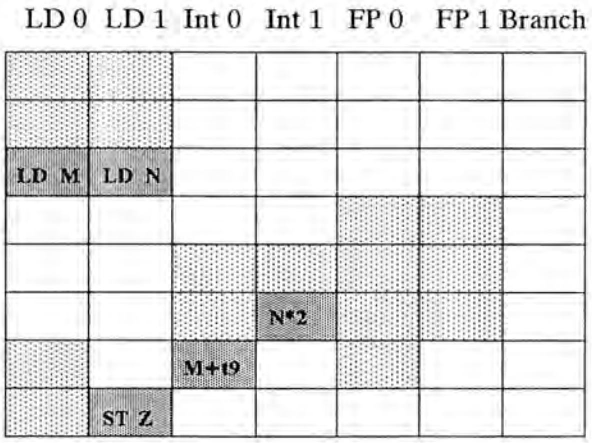
FORTRAN Code:

```
C = (A*2 + B*3) *2*I
Q = ( C+A+B )-4*( I+J )
Z = M + N*2
```

Sequential Code:

```
LD A              LD M
LD B              LD N
t1 = A*2          t9 = N*2
t2 = B*3          Z = M+t9
t3 = t1 + t2      ST Z
LD I
t4 = 2*I
C = t4*t3
ST C
LD J
t5 = I+J
t6 = 4*t5
t7 = A+B
t8 = C+t7
Q = t8-t6
ST Q
```

Wide Instruction Words:



Speedup was achieved without relying on any regularity or structure in the original statements, which were a fairly random assortment of operations. The compiler found low-level parallelism between the basic hardware operations required to execute these statements. By packing many operations into a chain of wide instruction words, the compiler is performing the same function as the hardware schedulers for the IBM 3090, the Cray supercomputers, and other machines. Compaction is not a new idea; by performing compaction in software, however, Multiflow has greatly decreased the cost and complexity of the hardware and greatly increased its ability to use this parallelism effectively.

4.3 Compacting Long Streams

Good use of VLIW architectures requires compacting many source code operations into single instructions; this is possible only when long streams of source code are available to be compacted together. This is evident in the above examples: better machine use resulted in the second example, in which more FORTRAN operations could be considered together for compaction.

The ability to find long streams of operations which can be compacted together is one of Multiflow's key technologies. All other attempts to work with low-level parallelism have failed to find speedups beyond a factor of two or four over sequential execution because they have failed to find long streams of operations to compact. Most earlier attempts have been hardware-based (early machines such as the CDC 6600 and IBM 360/91, and their heirs, the Cray and 3090); attempting to manage schedules while programs are running severely limits the number of operations and schedules which can be considered. A more crucial limit, however, has been the presence of conditional branches in programs.

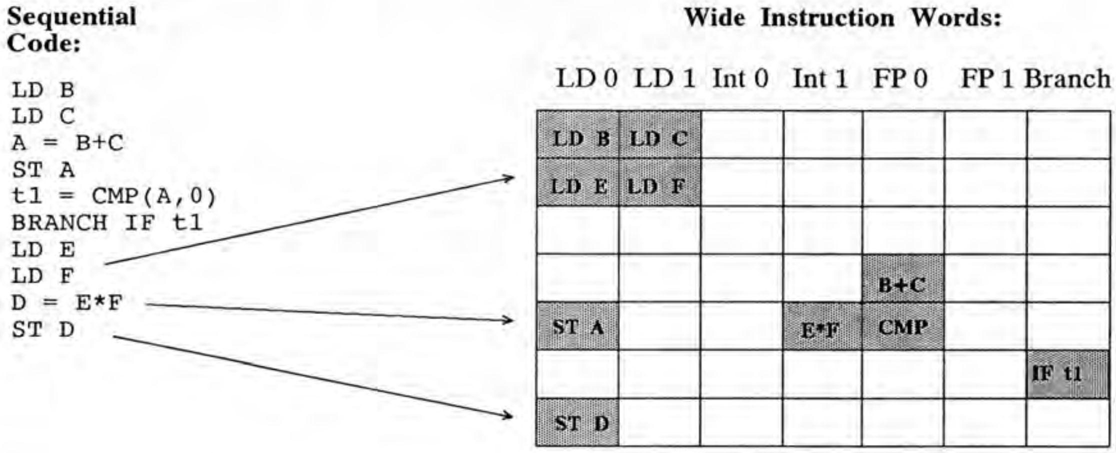
Consider the following example:

```
A = B + C
IF (A .LE. 0) GOTO 100
D = E * F
```

Where do we schedule the operations after the branch? To get the most efficient code, we would like to schedule them as early as possible. However, if we move these operations before the branch, we will execute them unconditionally. If it turns out that A is greater than 0 when we run the program, all is well and good; the program will accomplish the most work while executing the fewest possible instructions. However if A is less than 0, and the program branches, then we have committed an error: we have computed a new value for D when we should have left it untouched.

All prior compaction technologies have dealt with this problem by being conservative; no operations move beyond conditional branches. This has the practical effect of limiting the number of operations which can be scheduled together to 5 or 8 in the average program.

Multiflow's Trace Scheduling compacting compilers solve this problem in a different manner. First, we compact entire streams together, as if the branches were not there; in this particular case, the compiler schedules the load operations and the multiplication before the branch.



Sequential Code:

```
LD B
LD C
A = B+C
ST A
t1 = CMP(A,0)
BRANCH IF t1
LD E
LD F
D = E*F
ST D
```

Wide Instruction Words:

LD 0	LD 1	Int 0	Int 1	FP 0	FP 1	Branch
LD B	LD C					
LD E	LD F					
				B+C		
ST A			E*F	CMP		
						IF t1
ST D						

By treating this code as a single sequence, the compiler can schedule many operations at once and therefore generate more efficient code. It also means that the program always computes D, whether or not it branches. However, the compiler schedules the store operation for D *after* the branch. Although D is always computed, the program only stores D if it falls through. Therefore, the compiled code has exactly the behavior that the original FORTRAN code specified, even though the compiler has radically changed the order in which operations are performed.

This technique is called *compensation*; other kinds of compensation allow the compiler to move operations after branches, or to handle situations in which two streams of code join. We discuss compensation more thoroughly later in the chapter. The ability to rearrange primitive operations in an optimal way, and then compensate for this rearrangement, allows the compiler to work on extremely blocks of code containing conditional branches.

This small example scales up to large-scale compaction on your programs, without intervention on your part. With ordinary programs, Multiflow's compilers find opportunities for scheduling hundreds of operations together, achieving massive performance improvements.

4.4 Compiler Organization

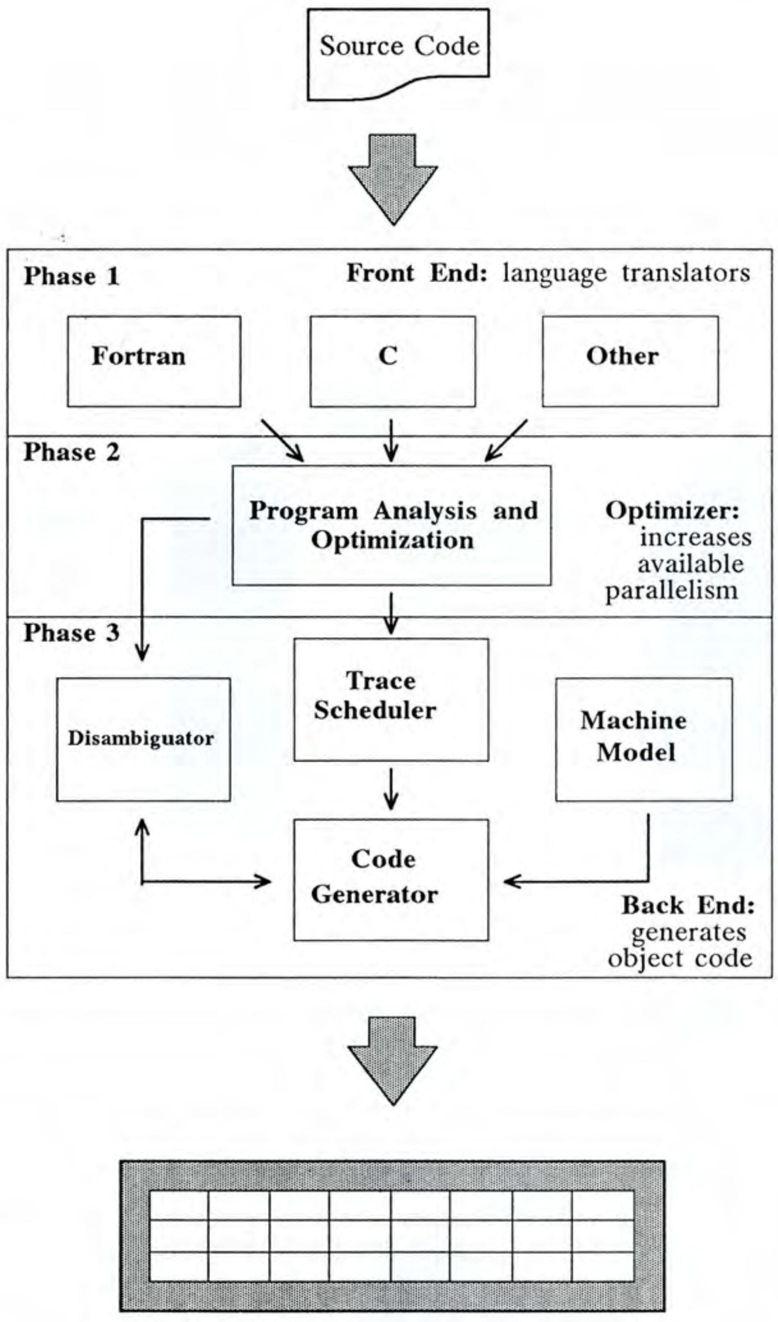
The compiler is organized in three major sections, known as Phase 1, Phase 2, and Phase 3. In order to compile a program, each phase processes the code in sequence. The three phases are:

- **Phase 1**: a language–specific *front end*. This part of the compiler interprets the program, which is written in a high–level language like FORTRAN or C, and translates it into a common intermediate language. The intermediate language representation is then passed to Phase 2.
- **Phase 2**: an enhanced optimizer that analyzes the program's structure (in its intermediate representation) and performs a many optimizations on it. Its output is another intermediate representation of the program, which is passed to Phase 3.
- **Phase 3**: uses Trace Scheduling compaction to generate wide instruction words. It takes Phase 2's intermediate representation, which is still serial, and compacts it into a sequence of wide instruction words, producing TRACE object code.

Together, Phase 1 and Phase 2 form an advanced optimizing compiler. Phase 3 is the heart of Multiflow's unique technology. Phase 3 is itself composed of four important modules: the trace scheduler, which selects traces (long blocks of code) from Phase 2's output; the code generator, which generates wide instruction word object code from these traces; the disambiguator, which analyzes memory references; and the machine model, which is a database containing detailed information about every aspect of the TRACE hardware.

The same compiler is used for all supportec' languages. Each language has a different front end (Phase 1); all share the same back end (Phase 2 and 3). Phase 1 is the only language–specific part of the compiler. All programs therefore receive the benefit of optimization and compaction, regardless of the source language.

The next diagram summarizes the compiler's organization.


Source Code
Phase 1
Front End: language translators
Fortran
C
Other
Phase 2
Program Analysis and Optimization
Optimizer: increases available parallelism
Phase 3
Disambiguator
Trace Scheduler
Machine Model
Code Generator
Back End: generates object code

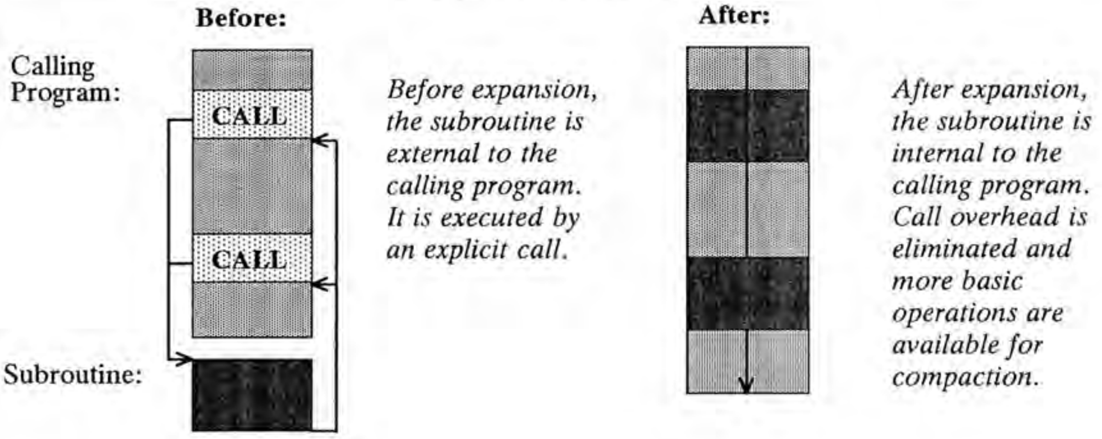
TRACE Instructions (object code)

4.5 Phase 1: Language Support

Phase 1 of the compiler reads the source code and translates it into an intermediate representation. It also performs storage allocation, assigning a location in memory to each variable and array the program uses. To aid in program development, Phase 1 generates cross-reference listings, error and warning messages, and other information about the program.

Phase 1 performs one optimization: automatic inline substitution. Inline substitution can replace a call to a subroutine or function with a copy of the function itself. This copy takes into account all the semantic details so that the substitution doesn't change the behavior of the subroutine in any respect. The following diagram represents graphically the effect of inline substitution:

INLINE Substitution


Before:
After:
Calling Program:
CALL
CALL
Subroutine:
Before expansion, the subroutine is external to the calling program. It is executed by an explicit call.
After expansion, the subroutine is internal to the calling program. Call overhead is eliminated and more basic operations are available for compaction.

Inline substitution eliminates the overhead involved in calling a subroutine. It also allows the code generator to schedule operations from the subroutine together with operations from the calling program. Inline substitution can be performed three ways: it can be performed automatically, in which the case the compiler uses heuristics to determine which functions and subroutine to expand inline; it can be performed on the basis of directives within the program's text; and it can be performed on the basis of command-line options that specify which program modules to substitute.

The output from Phase 1 is a representation of the program in an intermediate language. Programs written in FORTRAN, C, and all other supported languages are translated into the same intermediate representation. Because all front-ends produce the same intermediate language, all languages use the same back end (Phases 2 and 3), and get the full benefit of Multiflow's compaction technology. The intermediate language is similar to an assembly language for a sequential RISC computer, representing individual machine-level operations (loads, stores, multiplications, etc).

4.6 Phase 2: Program Analysis and Optimization

Phase 2 takes Phase 1's output and improves it. The improvements have two goals:

- Simplifying and eliminating calculations to reduce execution time.
- Reducing data dependencies between optimizations. This helps Phase 3 do a better job of compaction, producing faster object code.

As a first step, Phase 2 builds a *flow graph* of the program. The flow graph represents all possible courses of execution through the program. It then performs the following kinds of analysis on the program:

- **Loop structure analysis.** Phase 2 finds all loop structures within the program. This includes all DO loops, of any type, and also includes loops formed by GOTO statements and IF statements. All loops (not just DO loops) will eventually be candidates for loop unrolling.
- **Live variable analysis.** Phase 2 looks at all variables and finds all live variables (variables that are used more than once).
- **Data dependency analysis.** This analysis looks at the program's live variables and defines relationships between the statements in which a variable is assigned and statements in which the assigned variable is used. This information is used to determine the data dependencies between operations. More formally, data dependency analysis includes three separate analyses: reaching definition analysis, reaching use analysis, and reaching copy analysis.

4.6.1 Optimization

After performing these analyses (the results of which are used in optimization, and passed to Phase 3), Phase 2 carries out its optimizations. They include:

Induction Variable Simplification

Induction variable simplification simplifies loops by replacing repetitive multiplications with additions wherever possible. For example, the two loops below are equivalent:

```
                                     K = 0
        DO 10 I = 1,J                DO 10 I = 1,J
        K=I*10                       K=K+10
        A(I)=B(K)                    A(I)=B(K)
10      CONTINUE               10    CONTINUE
```

The loop on the right yields faster code because an integer multiplication has been replaced by an addition, a significantly faster operation.

Common Subexpression Elimination

Common subexpression elimination simplifies calculations by finding like expressions that appear in several statements and rearranging the program so that these expressions are only computed once. For example:

```
                            t1 = A*4
X = A*4 + B*3               X = t1 + B*3
Y = A*4 + D*3               Y = t1 + D*3
```

The compiler introduces a temporary variable to hold the result of A*4. Then it uses this result to compute both X and Y, rather than calculating A*4 twice.

Constant Folding

Constant folding consists of evaluating constant expressions during compilation, and replacing variables that are used as constants by the constants themselves. For example:

```
PI = 3.14159265
FOURPI=4*PI
RESULT=SERIES(X)/FOURPI     RESULT=SERIES(X)*.0795774716
```

The single statement on the right is equivalent to the three statements on the left. By evaluating constant expressions, the compiler eliminates several memory references, a multiplication, and a division.

Global Dead Code Removal

Dead code removal consists of identifying code that does not effect the program's result. and eliminating it. For example, the last statement in the program below can be executed, but does not have any effect on the program's result:

```
.....
PRINT *,ANSWER
ANSWER=SQRT(EARLIER)/4
END
```

Branch Elimination

Branch elimination replaces certain simple branch statements with in-line *select* operations. A select is a basic hardware operation whose result is equal to one of its two operands, depending on the results of an earlier comparison. A select chooses one of two values for its result, and therefore can be used to eliminate simple a conditional branches. For example, the IF statement in the code below can be replaced by a single select operation:

```
IF (A .GT. 1.0E12) A = 1.0E10
```

Register Variable Detection

Register Variable Detection identifies variables that can be kept entirely in registers, where they can be accessed without memory references. This optimization is particularly effective for the TRACE systems because of the large number of general registers they provide (640 general registers on the TRACE 28/300).

Loop Invariant Motion

Loop invariant code motion removes unchanging expressions from loops by moving them outside of the loop, where they can be calculated once. For example, consider the two loops below:

```
                                             t1 = 4*T/B
         DO 10 I = 1, 10                     DO 10 I = 1, 10
         A(I) = (4*T/B) + D(I)               A(I) = t1 + D(I)
10       CONTINUE                    10      CONTINUE
```

The expression 4*T/B is the same for every iteration of the loop. Therefore, the optimizer pulls this expression outside of the loop and assigns it to a temporary variable. After this optimization, the program only computes 4*T/B once, rather than computing it ten times. This is a version of *Common Subexpression Elimination* (above).

Variable Renaming

Variable Renaming introduces new names for unrelated uses of the same variable. For example, consider the two code sequences below:

```
1        A = Z*X                     1       A1 = Z*X
2        B = A*4                     2       B  = A1*4
3        A = Q*Y                     3       A  = Q*Y
```

In the code on the left, there is no data dependency between the values of A in statements 1 and 3. Therefore, the compiler can rename A in statement 1 without changing the program's behavior. The code on the right is exactly equivalent, except that we have replaced the name A with A1 in statements 1 and 2. This optimization increases the parallelism available to the code generator. After variable renaming, the assignments to A and A1 may be scheduled in parallel.

Loop Unrolling

Loop unrolling modifies the program's inner loops by copying their bodies several times. After make use ofthis optimization, the compiler can find parallelism between different loop iterations. For example, the code on the left is a simple loop as it might appear in the program; the code on the right represents the same loop, after the compiler has unrolled it four times:

```
      DO 10 I = 1, J                         DO 10 I = 1, J, 4
      A(I) = B(I)*C(I)                       A(I) = B(I)*C(I)
10    CONTINUE                               A(I+1) = B(I+1)*C(I+1)
                                             A(I+2) = B(I+2)*C(I+2)
                                             A(I+3) = B(I+3)*C(I+3)
                                       10    CONTINUE
```

In this case, unrolling the loop's body yields a trace with much greater parallelism. When Phase 3 generates code, it can make use of the parallelism in the body of the unrolled loop, scheduling computations for all four iterations together.

Loop Reduction

Loop reduction handles a class of recurrences. It eliminates apparent dependencies between loop iterations. For example, the compiler might replace the simple dot product computation on the left with the "reduced" dot product on the right:

```
      DO 10 I = 1, J                         DO 10 I = 1, J, 4
      S = S + X(I)*Y(I)                      t0 = t0 + X(I)*Y(I)
10    CONTINUE                               t1 = t1 + X(I+1)*Y(I+1)
                                             t2 = t2 + X(I+2)*Y(I+2)
                                             t3 = t3 + X(I+3)*Y(I+3)
                                       10    CONTINUE
                                             S = t0 + t1 + t2 + t3
```

In the original dot product, each addition depends on the previous iteration of the loop. Loop reduction exposes the parallelism inherent in the dot product by introducing several partial sums, all of which are independent. In the code on the right, computations of t0 through t3 are all independent, and can be overlapped during code generation.

Intrinsic Function Optimizations

Intrinsic function optimizations recognize patterns of intrinsic function use and improve the calling pattern, eliminating redundant internal calculations and finding more parallelism among the internal calculations of the intrinsics. These optimizations take into account several different kinds of phenomena. The examples below show two different cases in which intrinsic optimizations are applicable:

```
      X1 = R*COS(THETA)
      Y1 = R*SIN(THETA)
```

When processing this code, the compiler recognizes that a cosine and a sine can be computed much more efficiently by calling a special function in the math library that computes both together. The compiler automatically generates a call to this function rather than generating one call apiece to the sine and cosine functions.

```
          DO 10 I = 1, J
          X(I) = R*SIN(THETA(I))
    10    CONTINUE
```

In this example, the compiler first unrolls the loop and then recognizes that all of the sine functions can be computed simultaneously. Therefore, it generates a call to a special function in the math library that computes batches of sines. This is sometimes called a *vector intrinsic function*. Use of these functions provides better parallelism than computing the sines independently, in addition to reducing the function call overhead. Vector intrinsics exist for most elementary functions, including all of the standard trigonometric functions, exponential functions, and logarithm functions. There is no need for users to use vector intrinsics explicitly; the compiler automatically recognizes where calls to them are appropriate.

The compiler also uses basic mathematical and trigonometric relations to optimize calls to trigonometric, logarithmic, and exponential functions.

4.7 Phase 3: Trace Scheduling and Code Generation

Phase 3 of the compiler takes the optimized intermediate representation produced by Phase 2 and produces object code for the TRACE. This requires building a flow graph for the program, picking traces, and generating packed wide instruction words for each trace.

This part of the compiler is divided into four major modules:

- The *trace scheduler*, which builds a flow graph of the program, selects traces for compilation according to their usage, and passes them to the code generator for compaction.
- The *memory reference disambiguator*, which determines the relationships between data items in memory. Together with the machine model, the disambiguator lets the compiler replace arbitration and scheduling hardware.
- The *machine model*, which provides information about the TRACE's resources. The machine model makes sure that the resulting code obeys all of the TRACE's hardware constraints.
- The *code generator*, which takes traces generated by the trace scheduler and generates wide instruction words for them. The code generator is responsible for packing individual operations into wide instruction words effectively.

4.7.1 Trace Scheduler

The trace scheduler controls and coordinates the rest of Phase 3. Its job is to select sequences of operations that are as long as possible (*traces*) and pass these traces to the code generator. The code generator compacts the trace into a *schedule* of machine code and returns the schedule. The trace scheduler then corrects any inconsistencies (performs *compensation*), and the process repeats until machine code has been generated for the entire program.

The trace scheduler starts its work by analyzing the control flow of the program. It builds a flow graph of the program. This graph represents each step required to execute the program. It then marks each item in the graph with its probability of execution, using heuristics about branch and loop behavior, information provided by directives, and information derived from loop trip counts.

A *trace* is then selected starting with the highest probability statement, working forwards and backwards following the highest-probability path. The trace scheduler then passes it to the code generator; since Phase 2 optimizations will provide opportunities for long traces, excellent machine code will result.

The large-scale rearrangement performed by the code generator moves operations in ways which could cause logical inconsistencies when the program branches off the chosen trace. The trace scheduler corrects the flow graph of the remaining program to correct these inconsistencies; this process is known as *compensation*.

Compensation takes into account three possibilities:

- If the code generator takes an operation that was originally located after a branch, and moves it before a branch, this operation will be performed unnecessarily whenever the program branches. For example, the compiler often moves LOAD operations to the beginning of a loop body so that computation doesn't have to wait for data to arrive from memory. When this occurs, the trace scheduler manages register assignment so that the operation's result is never used if it turns out to be unnecessary.
- If the code generator takes an operation that was originally located before a branch, and moves it after the branch, the compiler must make sure that the operation is performed whichever way the program branches. To do so, the trace scheduler adds a copy of this operation on the off-trace side of the branch; with this extra copy, the operation will always be executed.
- If two traces join each other, the compiler may also need to generate a few copies of operations to splice the code segments together correctly.

The trace scheduler takes appropriate action in each of these cases. Only the latter two involve adding any extra operations; the first case is by far the most frequent. Therefore, the number of compensation operations is always quite small. In addition, compensation operations (if any) are compacted into wide instruction words, like the rest of the code, and often occupy fields that would otherwise be unused. As a result, compensation operations account for an insignificant fraction of code size and execution time.

The ability to compensate for assumptions made during code generation lets the compiler rearrange the operations to get optimal efficiency from the computer's resources.

4.7.2 Code Generator

The code generator takes whole traces and compacts them into optimal sequences of wide instruction words. It is responsible for:

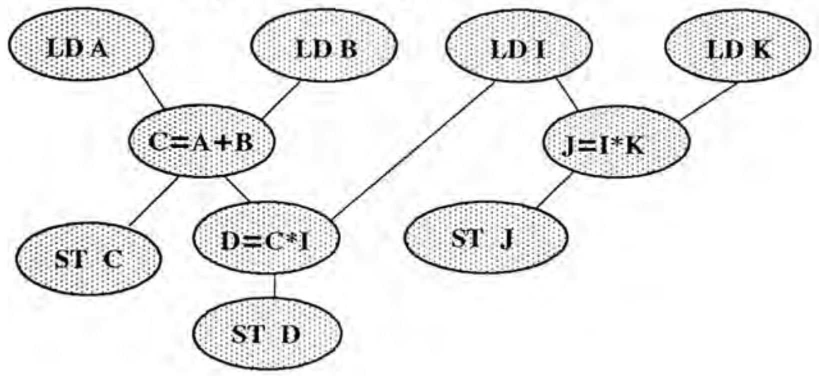
- Finding data dependencies between individual operations on the trace;
- Deciding which functional unit to use for each operation;
- Assigning registers;
- Scheduling each operation in an optimal way.

To analyze data dependencies, the code generator builds a data dependency graph (called a *directed acyclic graph*, or *DAG*) that shows each operation in the trace and its relationships to other operations. Prior to reaching this stage, Phase 2 optimizations like variable renaming and copy propagation have removed the program's superficial dependencies (i.e. dependencies that have to do with how the programmer wrote the code, but aren't basic to the algorithm itself). Only those data dependencies remain that are intrinsic to the computation. This simplifies the DAG and makes it easier to schedule operations efficiently.

The following diagram shows three FORTRAN statements and the primitive operations to which they correspond. The graph on the right represents each operation, showing all the data dependencies between them. Each operation is a node on the graph (represented by an oval); each data dependency is an edge (represented by a line):

Source:	**Primitive Operations:**
C=A+B	LD A
D=C*I	LD B
J=I*K	C=A+B
	ST C
	LD I
	D=C*I
	ST D
	LD K
	J=I*K
	ST J

Data Dependency Analysis:



This graph represents all dependency relationships within the program, and can be used to generate code. The edge between C=A+B and LOAD A shows that the addition cannot be performed until the data has returned from the LOAD operation. To analyze data dependencies among memory references, the code generator consults the *disambiguator*, discussed below.

After building a DAG, the compiler assigns each operation to a functional unit. In doing so, it tries to minimize the need to move data between different clusters. For example, the compiler will assign the operations D=C*I and C=A+B to the same cluster: this assignment allows the program to use C immediately, without copying it to another cluster.

After assigning functional units, the code generator compacts the operations represented by the DAG into a sequence of wide instruction words. Operations are scheduled into instructions using the DAG and the *machine model*, discussed below, to assign registers and generate the most efficient sequence of wide instruction words.

Subroutine and function calls require special treatment. All TRACE programs use a "caller-saves" calling sequence, in which the code making a subroutine call is responsible for saving the values of any registers that it cares about, and restoring these values after the subroutine has completed. Because Phase 3 performs register allocation, and has the results of Phase 2's data dependency analyses, it determines which registers are live across a call (i.e. contain data that will be needed after the subroutine has finished) and therefore need to be saved in memory. Data that won't be needed after the call isn't saved. Saving only the live registers decreases the overhead required to call a subroutine.

This calling sequence, together with the register passing strategy, has proven to be extremely efficient, reducing calling overhead to a minimum. The standardized calling sequence provides complete compatibility between code written in different source languages; FORTRAN program can call C language subroutines and library functions, and vice-versa, without difficulty.

4.7.3 Memory Reference Disambiguator

The memory reference disambiguator is a part of the compiler that determines whether or not any pair of memory references can possibly refer to the same location in memory. The code generator uses this information to determine data dependency relationships and build the data dependency graph from which it schedules operations.

It is often difficult to disambiguate two references to the same array. To do so, the disambiguator uses information gathered by Phase 2 to derive symbolic expressions for the array indices. It then determines symbolically whether or not the index expressions can ever be equal. For example, consider following unrolled loop:

```
          DO 10 I = K+1, N, 2
1         A(I,J) = A(I,J) + T*A(I,K)
2         A(I+1,J) = A(I+1,J) + T*A(I+1,K)
10        CONTINUE
```

To generate good code for the body of this loop, the compiler would like to move all of the LOAD operations to the beginning of the loop's body. This is only legal if there are no data dependencies between statement 1 and statement 2. That is, the compiler can only schedule a LOAD operation for A(I+1,K) before the STORE operation for A(I,J) if it knows that these two references cannot possibly refer to the same array element. Therefore, the compiler asks the disambiguator whether or not A(I,J) can refer to the same element of the array as either A(I+1,J) or A(I+1,K). The answer to the first question is trivial: A(I,J) and A(I+1,J) can never refer to the same element.

The answer to the second question is not trivial, because we don't know the value of K. To determine whether or not A(I,J) can refer to the same location as A(I+1, K), the disambiguator generates address expressions for each of these elements:

```
&A(I,J)  = &A(1,1) + 100*4*J + 4*I
&A(I+I,K)= &A(1,1) + 100*4*K + 4*(I+1)
```

We use the notation &X to mean "the address of X" (as in the C programming language); therefore, &A(1,1) means "the address of A(1,1)," which is the base address for the array. The constant 4 is the size (in bytes) of each array element, which is single precision. Setting these two expressions equal, we have the relation:

```
&A(1,1) + 100*4*J + 4*I = &A(1,1) + 100*4*K + 4*(I+1)
```

Simple algebra reduces this equation to:

```
100*(J-K) = 1
```

If there are any integer solutions to this equation, then it is possible for A(I,J) to refer to the same array element as A(I+1,K), and the compiler must schedule the LOAD for A(I+1,K) after the STORE for A(I,J). In this case, there are no integer solutions; this means that the compiler can schedule the LOAD as early as possible.

Multiflow's compilers handle many arbitration functions that normally fall to hardware. In particular, they must always generate code that obeys certain resource restrictions on memory and bus usage; the hardware can detect illegal situations, but cannot perform arbitration to resolve these problems. The disambiguator helps the compiler enforce three rules:

- Two memory references scheduled in the same beat must never refer to data on the same memory controller. This is called a card conflict.
- Two memory references scheduled in the same beat must never require the same bus to store or return data. This is called a bus conflict.
- Two memory references scheduled within three beats of each other that refer to the same memory bank incur a slight performance penalty. This is called a bank conflict.

The compiler must always observe the first two rules. For optimal performance, it observes the third wherever possible. To enforce these rules, the code generator asks the disambiguator another set of questions. Can two memory references refer to data on the same memory controller? Can they refer to storage in the same memory bank? Will the memory controllers need to overuse the data busses to service these memory references? The disambiguator determines whether conflicts can occur and, if so, what kind. The mathematics required to solve this problem are identical to those used to determine if two references can address the same element.

4.7.4 Machine Model

To perform the ambitious scheduling needed to use the TRACE's hardware resources efficiently, the compiler requires a thorough knowledge of the hardware. A portion of the compiler called the *machine model* is a database that manages this information. The machine model provides information about:

- Operation codes for each type of operation;
- Pipeline depths for each type of operation;
- Resource requirements for each type of operation;
- Resource availability and restrictions (which depend on configuration);
- Datapath availability.

The code generator uses this information to assign operations to functional units and to determine when operations can be scheduled legally. For example, when the compiler needs to schedule a double precision floating point multiplication, the machine model shows that the TRACE can perform this operation on either FALU0 or FALU1; that the operation requires two operands from the floating point register file, which is in use for two beats; that the multiplier itself is in use the beat after the instruction is issued; that the chosen FALU's output port is in use for two beats, starting two beats after the operation was initiated; and that the result may be used as an operand for other operations two beats after the multiplication was initiated. The code generator uses this information to determine whether or not the operation can be scheduled legally, and to track the availability of resources within the CPU for scheduling subsequent operations.

The machine model contains information about each TRACE model: the 7/300, 14/300, and 28/300, in addition to the TRACE /200 series. Any TRACE compiler can therefore cross-compile for any TRACE configuration.

4.8 Language Specifications

4.8.1 FORTRAN

Multiflow FORTRAN fully implements ANSI FORTRAN 77 (ANSI X3.9–1978), including features specified in the Department of Defense Supplement (MIL–STD–1753). It is compatible with the FORTRAN 66 standard (ANSI X3.9–1966), and includes extensions from the proposed FORTRAN 8X (ANSI X3J3/S8). The compiler has been fully validated by the National Bureau of Standards, and is in conformance with FIRMR 201–8.109.

The Multiflow FORTRAN compiler implements many features of VAX/VMS FORTRAN, allowing easy program portability between the TRACE/UNIX and VAX/VMS environments.

Format and syntax extensions include:

- 32–character variable names
- End–of–line comments
- INCLUDE statement
- DATA statement
- VAX syntax for octal and hexadecimal constants
- Tab character formatting
- Extended (132 character) source lines
- DO WHILE and END DO statements
- D debugging lines
- %VAL, %DESCR, %REF, and %LOC built–in functions
- OPTIONS statement
- VAX FORTRAN records
- command line options to request default SAVE status, common block padding, and default zero initialization
- command line option to specify inline substitution

Data type extensions include:

- INTEGER*1, and BYTE data types
- LOGICAL*1 and LOGICAL*2 data types
- Direct conversion of VAX data representations when reading or writing VAX/VMS files
- Command line option to select DOUBLE PRECISION as the default floating point data type, rather than REAL

I/O extensions include:

- Asynchronous I/O (ASYNCHRONOUS file type, WAIT keyword, and ASYNCWAIT statement)
- NAMELIST I/O statement
- TYPE, ACCEPT, DECODE, ENCODE statements
- Sequential or relative organization for files
- Fixed–length, variable–length, and stream record types

- Read-only files
- VAX files directly accessible through TRACE/DN (DECnet)
- Many miscellaneous VAX/VMS I/O keywords

Extensions to support fine-tuning include:

- ASSERTION directives
- IFPROB directives (assertions about probabilistic behavior of branches)
- UNROLL AMOUNT directives (explicit loop unrolling request)
- BEGIN and END INLINE directives (explicit inline expansion requests)

Other extensions for improved functionality and performance include:

- Default use of shared libraries, to minimize executable image size
- TRACE/UNIX system call library
- Enhanced mathematics library
- VAX/VMS system call library
- Automatic use of vector intrinsic functions
- Stack traceback upon abnormal termination
- Cross-reference listings

The directives for fine tuning allow the programmer to control the optimization process. These statements may be used to override heuristics used by the compiler, provide information about memory references that is not available to the compiler, and control loop unrolling and inline expansion optimizations.

4.8.2 C

Multiflow C includes the complete UNIX System/V standard C language, with the Berkeley standard extensions. The C compiler also implements features from the proposed ANSI C standard (X3J11/88). Multiflow has added extensions to support inline expansion. By default, ANSI and MFCI extensions are disabled, providing complete compatibility with standard C. All compilers share the same back end, and thus benefit from Multiflow's advanced optimization technology.

Runtime data formats are exactly compatible with the runtime data formats used on 68000-series workstations. Pointers, integer representation, and byte ordering are all compatible. For floating point numbers, the IEEE 754 representation is used.

4.8.3 Other Languages

A PASCAL compiler is available for the Multiflow TRACE family. This compiler supports the ISO standard 7185; options select either Level 0 or Level 1 compliance. Level 0 is equivalent to ANSI/IEEE standard 770X3.97-1983, and lacks the conformant arrays supported by Level 1.

A COMMON LISP compiler is available for the Multiflow TRACE family.

4.8.4 Libraries

TRACE/UNIX provides a complete set of enhanced math libraries; I/O libraries; string manipulation libraries; UNIX system call libraries; and a VMS-compatible system call library. These libraries may be used by programs written in both FORTRAN and C. The math libraries provide exceptional accuracy, and have been tuned for optimal performance on each TRACE 300 series model. An extremely wide range of functions has been implemented, including all of the standard power, trigonometric and exponential functions; hyperbolic functions; Bessel functions; and others. All of the math functions obey IEEE 754 rules for handling infinities, not-a-number, and other illegal conditions.

All standard TRACE/UNIX libraries are available both as shared libraries and as archived libraries. This includes all run-time, system call, I/O, and math libraries for C and FORTRAN. The standard TRACE/UNIX utilities are compiled with shared libraries. Programs written by users may use either the shared libraries or the archived libraries, as requested during compilation and linking. Users may create their own libraries, which may be either standard (archived) libraries or shared libraries.

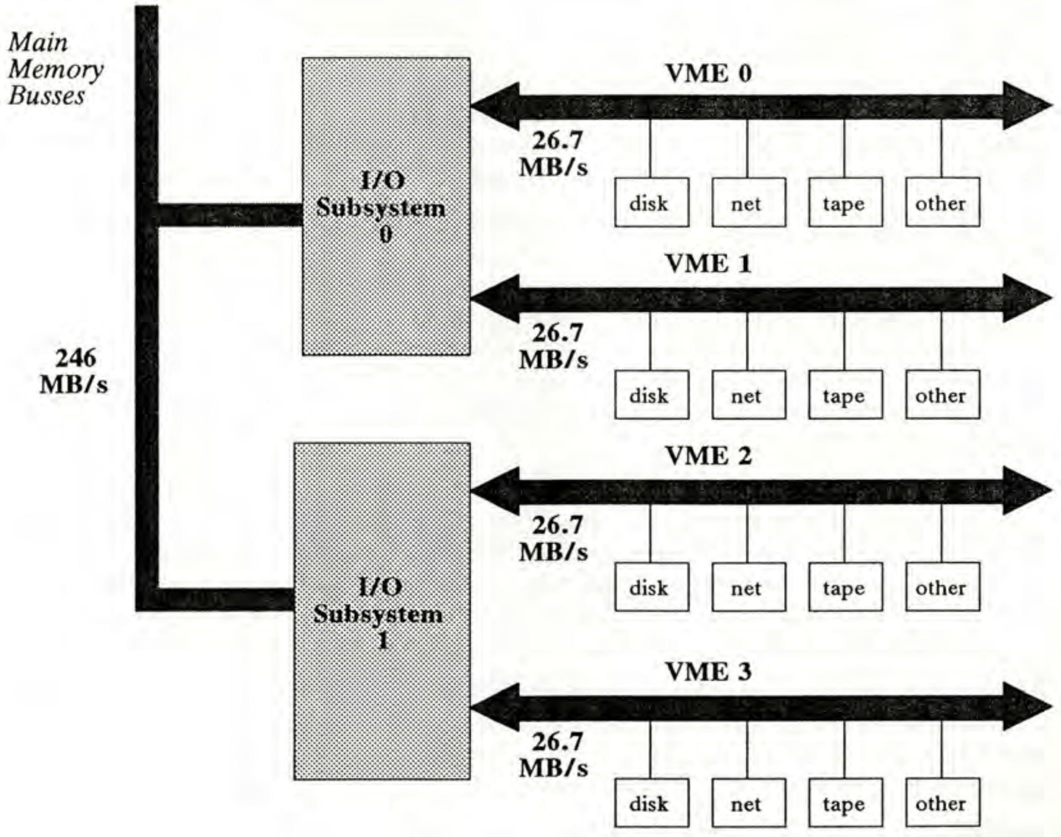
An optional library of advanced mathematical routines performs extremely efficient matrix operations, fast fourier transforms (FFTs), vector intrinsic functions, and other operations. All the routines in these libraries have been carefully tuned to deliver the maximum performance of which the TRACE is capable. They have been designed so that they can be dropped in to many standard programs without any conversion effort.

CHAPTER 5
TRACE I/O ARCHITECTURE

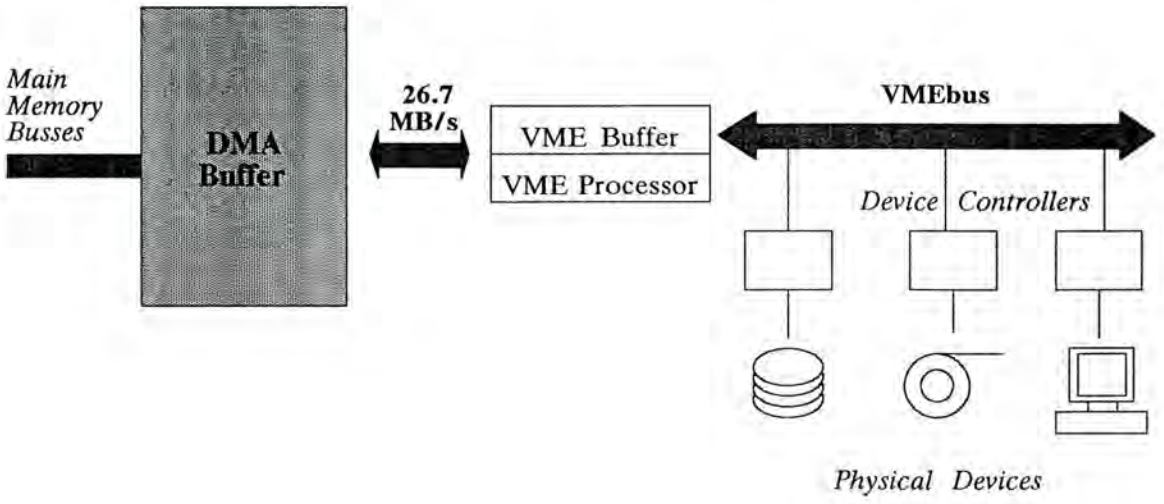
The TRACE 300 series has a high-performance I/O architecture, advanced peripherals, and system software designed for maximum throughput with minimum overhead. TRACE /300 systems provide a well-balanced supercomputing environment that delivers performance in every respect, including the data throughput needed to support computation at supercomputer speeds, eliminating I/O bound jobs. The TRACE I/O Subsystem is capable of performing under heavy loads without draining the CPU's resources. It also supports the extremely large files and file transfers typical of large scientific and engineering computations.

5.1 I/O Subsystem

The TRACE 300 series has an channel-based I/O architecture that is designed for high performance and maximum efficiency. Each TRACE may have one or two I/O subsystems. Each I/O subsystem moves data between the TRACE and main memory at 246 Megabytes per second. Each I/O subsystem can be connected to one or two VME busses; disk drives, tape drives, networks, terminals and other peripherals are interfaced via VMEbus device controllers. Data moves between the I/O subsystem and the VME busses at 26.7 Megabytes per second. The following diagram represents the I/O subsystem's basic connectivity:



The channel-based I/O architecture shields the CPU from direct interaction with peripherals; the main CPU only makes DMA transfers to the I/O subsystems, which take place at the full memory bandwidth. An independent processor on the VMEbus takes care of all low-level I/O management. The following diagram shows the design of the I/O subsystem, configured with a single VMEbus, in greater detail:



The main components of the high performance I/O subsystem (HPVME) are:

- **DMA Buffer**: A buffer that makes DMA transfers to and from the TRACE's main memory. It adapts the relatively low I/O transfer speeds to the memory subsystem's extremely high data bandwidth. It transfers data at 246 Megabytes per second between its internal buffer and main memory. This data rate minimizes TRACE CPU cycles lost to I/O. One CPU can be configured with one or two DMA buffers; each DMA buffer can be configured with one or two VMEbusses, for a maximum of four VMEbusses per system.
- **VME Buffer**: A buffer memory directly accessed by VMEbus devices. The I/O subsystem transfers data between this memory and the DMA buffer at 26.7 Megabytes per second. The size of the VME buffer is configurable from 1 to 4 Megabytes. An auxiliary bus transfers data out of the VME buffer at full speed, while leaving the VMEbus free for use by the I/O controllers and the VME processor.
- **VME Processor**: A single-board Motorola 68030-based processor with its own local memory. This processor runs the device drivers, freeing the CPU from low-level device management. It handles interrupts from devices, sends commands to the controllers, and controls the DMA engine that transfers data between the VME buffer and main memory. The VME processor never handles I/O data directly, since this would limit the transfer rate.

  The VME Processor also runs the Multiflow Diagnostic Executive (MDX). This is a lightweight real-time multi-tasking operating system that provides interrupt handling and context management for device drivers. It also provides a command environment for diagnostics, booting, and controlling device configuration.

Each VMEbus has its own interface, buffer, and processor boards. Each VMEbus can accommodate up to 18 peripheral controllers, providing maximum I/O connectivity for the entire system.

The I/O subsystem is also the platform for CPU diagnostics. One I/O subsystem acts as a master for the TRACE's serial diagnostic bus, which provides access to registers, memories, and major control signals on each CPU board. The VMEbus processor provides an interactive environment for running diagnostics.

5.2 Operating System Support

The TRACE/UNIX I/O system has been optimized for achieved I/O performance. TRACE/UNIX communicates with device drivers which run on the VME Processor, by using an intelligent channel protocol. As much as possible, the channel protocol reserves the CPU for computation, relying on the I/O control processors to do most of the work. The main CPU performs high level operations, services relatively few interrupts, and performs no physical device management. Thus large amoungs of I/O can be performed with minimal CPU overhead.

The VME processors take care of the low-level operations required to perform I/O. They service all device interrupts, prepare and transmit device commands, and control the DMA engine that governs data transfers between the I/O subsystem and main memory.

The channel protocol partitions I/O processing between the main CPU and the control processors. The CPU performs high level operations like buffer management and other tasks that benefit from the TRACE's speed and memory bandwidth. Real-time interaction that is limited by the speed of the I/O device is handled by the VME processors, which effectively shield the main CPU from most of the I/O overhead. This division of labor is superior to other architectures in which relatively slow front-end processors are responsible for all I/O handling. It yields excellent I/O performance and interactive response.

The TRACE CPU initiates I/O requests by preparing a channel command requesting a device driver to read or write data. When the command is ready, the CPU signals the I/O subsystem, which moves the command packet into the DMA buffer in one burst.

The I/O subsystem then delivers the command packet to a device driver running on one of its control processors. The device driver issues the relevant commands to move data from the VME buffer to the device, or to move data from the device to the VME buffer. The device driver never touches the data itself, relying on the I/O device to make high-speed block transfers directly into the VME buffer.

When the device driver has finished servicing the request, it prepares a channel response packet, which includes any data buffers that need to be moved back to the CPU. When the packet is ready, the device driver sends it back to the main CPU. The response returns to main memory in another burst. When the transfer has finished, the I/O subsystem interrupts the CPU to inform it that the data has arrived.

Both the CPU and the device drivers maintain queues of active I/O requests. By managing queues of outstanding I/O requests, and optimizing the order in which they act on these requests, the device drivers attain optimum throughput. For example, disk drivers perform seek optimization to minimize head travel and attain the best possible actual transfer rate from the disk.

5.3 Filesystem Enhancements

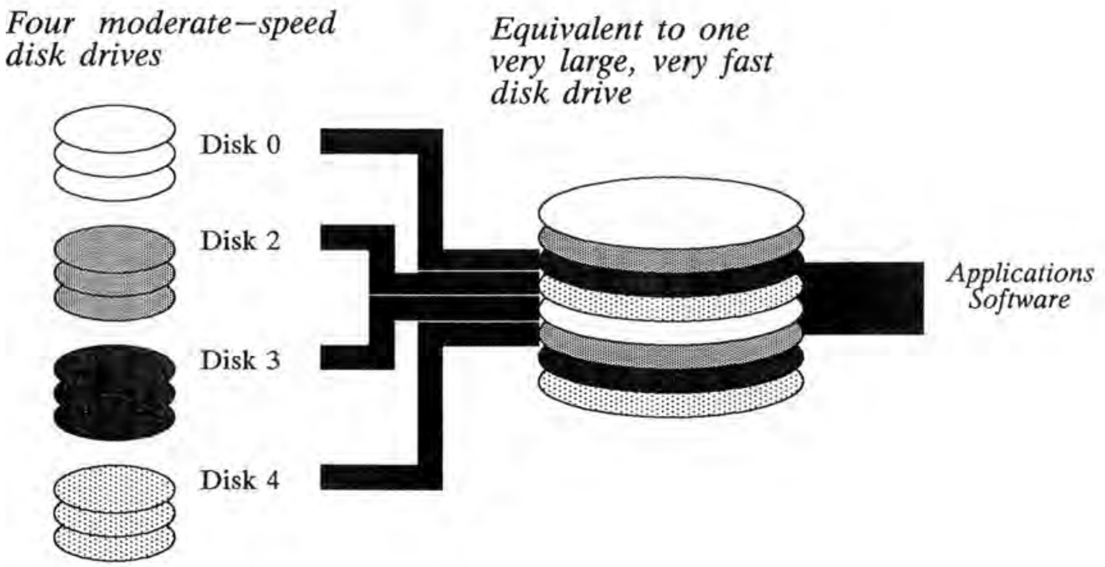
Scientific applications software often requires large amounts of data; simulation and other applications may require huge input files or generate gigabytes of intermediate results. These constraints place great demands on scientific computers, both in terms of the speed and disk storage required. The TRACE I/O system has been designed with these requirements in mind.

The TRACE/UNIX filesystem handles the very large files that are typical of large scientific applications. It uses disk block sizes up to 128 Kilobytes, a data cache of up to 32 Megabytes, directory look–up caching, file write–behind and read–ahead, and a tuned file allocation algorithm. Maximum file size is 64 Gigabytes; i.e. file size is limited only by the amount of storage available. The basic TRACE/UNIX disk configuration supports files up to 1 Gigabyte in length.

To further increase delivered performance, the TRACE/UNIX filesystem implements filesystem striping. A *striped disk* is a group of disk drives that have been configured to act as a single disk partition. Files are interleaved between the individual disks making up the filesystem on a block–by–block basis. Disk striping allows TRACE/UNIX to perform disk operations in parallel, retrieving data from all of the striped disk's components simultaneously and reassembling it. This allows sustained transfer rates much faster than a single disk drive can support, and file sizes much larger than a single disk drive can provide. Disk striping is completely transparent to the user and to any user–written software; accessing a file on a striped disk is identical to accessing any other file.

A striped disk drive can be formed from up to eight physical disk drives. A striped disk formed from four disk drives yields maximum sustained performance of over 8 Megabytes/second, and a maximum capacity of roughly 4 Gigabytes. A striped disk formed from eight disk drives yields a maximum sustained performance of over 10 Megabytes/second, and a maximum file size of roughly 8 Gigabytes.

Disk striping is extremely flexible. System managers can configure striped disks, and can change the configuration of these disks at any time. A single system can mix striped and non–striped disks. The diagram below illustrates disk striping:


Four moderate–speed disk drives
Equivalent to one very large, very fast disk drive
Disk 0
Disk 2
Disk 3
Disk 4
Applications Software

TRACE/UNIX provides an asynchronous I/O facility. Asynchronous I/O lets application programs explicitly overlap computation and I/O operations. It also minimizes I/O overhead by copying data directly to and from buffers in the program's address space. Asynchronous I/O has been integrated into the FORTRAN I/O library, allowing FORTRAN programs to manage asynchronous I/O directly.

TRACE/UNIX integrates all of these features into a single hierarchical filesystem that spans many different physical disk drives. A single, consistent naming strategy allows users to specify any local or remote file without concern for its locality, disk type, or striping. To a program, there is absolutely no difference between a local file, a file accessed through NFS, a file on a striped filesystem, or any other file type. The filesystem's delivered performance and flexibility makes TRACE/UNIX an excellent environment for general timesharing use, program development, and production.

5.4 Peripherals

Multiflow provides a full line of peripherals, including disks, tapes, terminal controllers, network controllers, and others.

5.4.1 Disk Drives

Multiflow is committed to using the most advanced and reliable disk drives and disk controllers available. The 1 Gigabyte disk drives use 8 inch sealed Winchester disk technology. These disks are extremely compact, and combine large capacity and high transfer rates with extremely high density. Up to 20 disk drives can fit into a single peripheral cabinet, providing 20 Gigabytes of storage in a 28″ by 45″ footprint. A single drive provides raw transfer rates of 3.0 Megabytes/second. Striped configurations provide much higher raw transfer rates and capacities. All Multiflow disk drives perform automatic error detection, correction, and recovery.

5.4.2 Tape Drives

Multiflow provides three tape alternatives: a 200 IPS (inches per second) start-stop 9-track tape drive, a 50 IPS start-stop 9-track tape drive, and a high-capacity cartridge tape drive.

Both 9-track tape drives are auto-loading tri-density tape drives that use standard 1 mil, 1/2 inch, 2400 foot tapes. By supporting 800, 1600, and 6250 BPI densities, these tape drives are compatible with virtually all tape systems in use. The 200 IPS drive is particularly useful in environments that use magnetic tape heavily as a medium for transferring or storing data.

The cartridge tape drive, standard on all systems, uses advanced helical scan recording technology (used in 8mm video cassette systems) to achieve tape densities well beyond the capabilities of nine-track tapes. Advanced error correction techniques and high-coercivity media result in data reliability substantially better than nine-track tapes. It is ideal for archives, for backups, and for data interchange between TRACE systems; a single cartridge provides up to 2 Gigabytes of storage in a package that measures only 4″ by 2.5″ by 0.75″. The tape cartridges themselves are inexpensive and widely available.

5.4.3 Terminals and Workstations

With flexible support for many families of terminals, workstations, and graphics devices, TRACE systems adapt well to any environment. They can be used alongside of equipement from all major computer manufacturers, ranging from simple ASCII terminals to the most powerful supercomputers.

TRACE/UNIX supports virtually all terminal types through the UNIX *termcap* database. In particular, all DEC terminals are supported, including the VT52, VT100, VT125, VT200, and VT220. Virtually all Tektronix terminals are also supported. One TRACE system can support up to 64 RS-232C terminal lines. DR-11W and Versatec interfaces allow the TRACE to support high-speed DMA-driven raster terminals.

TRACE/UNIX's strong networking features allow you to use SUN, DEC, HP, Apollo, Silicon Graphics, and other workstations as I/O devices. Features like the X Window System and Network Window System allow you to use these workstations to provide a graphics front end for programs running on the TRACE. The NETdisk service (part of the Network Filesystem) allows a TRACE to act as a SUN fileserver, letting diskless workstations boot and page from the TRACE's disks.

Standard Ethernet networking lets the TRACE communicate fluently with all other UNIX systems (using TCP/IP) and all DEC systems (via DECnet).

5.4.4 Peripheral Specifications

The following tables show specifications for selected peripheral devices that are available for the TRACE 300 series. For a complete list of the peripherals that are available and their specifications, contact your sales representative.

Disk Drives

Formatted Capacity:	1.0 GB
Transfer Rate:	3.0 MB/sec
Average Latency:	8.33 milliseconds
Maximum Latency:	16.83 milliseconds
Average Seek time:	16 milliseconds
Technology:	8" Winchester
Expansion:	20 drives per I/O cabinet
Error Checking:	Built-in on-the-fly correction via ECC polynomial

1/2 Inch Tape Drives

Speed:	50 IPS	200 IPS
Densities:	6250, 1600, 800 BPI	6250, 1600, 800 BPI
Rewind time:	120 seconds (2400' tape)	66 seconds (2400' tape)

Cartridge Tape Drive

Tape:	Standard Video 8 cartridge
Capacity:	Up to 2 GB per tape
Technology:	Helical Scan Recording
Interface:	SCSI
Transfer Rate:	225KB/s sustained; up to 1.5 MB/s burst
Rewind Time:	135 seconds (2 GB cartridge)
Error Checking:	Read-after-write and automatic rewrite

One cartridge tape drive is standard on all systems.

Network Controllers

Physical Link:	IEEE 802.3 (Ethernet)
Data rate:	10 Mbit/second
Supported Protocols:	TCP/IP, DECnet phase IV

Line Printers

Interface:	Centronics
Lines per Minute:	300, 600, 1200 lines per minute
Width:	132 columns
Character set:	Standard 96-character ASCII

Terminals and Graphics Devices

Terminal Interface:	RS-232C
Baud rate:	60 to 19200
Number:	Up to 64
Graphics interfaces:	DR-11W, Versatec interfaces supported

CHAPTER 6
THE TRACE/UNIX OPERATING SYSTEM

In the past few years, UNIX has gained acceptance as the de-facto standard for technical computing environments. UNIX has been implemented on an extremely wide range of computing systems, ranging from personal microcomputers to supercomputers. Every major computer manufacture, including those who have developed proprietary operating systems, supports a UNIX version. It is the only operating system currently available that provides portability and compatibility over such a wide range of computers. Standard networking features support reliable communications with computers of many types and architectures.

All UNIX implementations provide tools ranging from editors to typesetters to lexical analyzers and compiler generators. This repertoire includes debuggers, execution profilers, sort/merge utilities, a database facility, and many other programs and libraries. The list of UNIX tools grows constantly as new features, like the X Window System, the Network Filesystem, the Network Queuing System, and others are added.

Multiflow's UNIX implementation, TRACE/UNIX, is an enhanced version of Berkeley 4.3 BSD UNIX. Multiflow has extended the basic 4.3 release with additional functionality and performance enhancements. TRACE/UNIX demonstrates Multiflow's deep commitment to supporting, integrating, and maintaining new standards and new features; it includes the X Window System, the Network Computing System, the Network Queueing System (a batch queue facility), the Network Window System, and other features as standard parts of the product. The Network Filesystem is available as an option.

6.1 Performance Enhancements

TRACE/UNIX's ability to support many users and many concurent tasks is exceptional, and allows the TRACE to serve as a resource that is shared by a large community. The TRACE's efficient virtual memory implementation guarantees that it will perform effectively in environments that stress the ability to handle extremely large jobs.

6.1.1 Multiuser Efficiency

TRACE computers perform effectively in multiuser environments. They were designed for maximal efficiency under a heavy workload, minimizing the time lost to operating system overhead. As a result, a single TRACE can support thirty to sixty users comfortably in a time-sharing environment.

To support a large number of active processes effectively, the TRACE computer and the TRACE/UNIX operating system minimize the time required for a context switch. This is the overhead required to stop one process, and schedule another during general time-sharing. At a minimum, a context switch requires saving the contents of the TRACE's registers and doing whatever is needed to use a different virtual address space. Under a heavy workload, the context switch time can account for a significant portion of total system overhead.

The operating system's code for saving and restoring registers has been carefully tuned for optimal performance, using the TRACE's maximum memory bandwidth. The address translation hardware (the instruction and data TLBs) and the instruction cache (the ICACHE) have been designed so that changing from one address space to another requires virtually no overhead. Each entry in the TLBs and in the ICACHE is tagged with an 8–bit process number (*address space ID*, or *ASID*). Therefore, to execute a context switch, TRACE/UNIX only needs to load a new address space ID number into a special register. The old entries can remain; during normal execution, the hardware detects mismatches between the ASID associated with a cache entry and the ASID of the currently running process. Consequently, TRACE/UNIX only needs to replace elements in the ICACHE or TLB as needed. There is no need to purge the instruction cache or the TLB.

This design requires only 150 microseconds for a complete context switch. The result is effective performance with minimal time lost to system overhead, even under very heavy multiuser loads.

6.1.2 Virtual Memory Features

TRACE/UNIX provides a demand–paged virtual memory management system, with a logical address space of 4 Gigabytes per process. Several extensions in TRACE/UNIX increase the efficiency and flexibility of virtual memory management. Increased virtual memory efficiency greatly reduces the system overhead required to support large scientific programs.

TRACE/UNIX uses designated swapping files in the filesystem for all paging (swapping) activity, unlike most UNIX systems which require designated fixed size swapping partitions. Paging files can be created or deleted with a single administrative command. Filesystem paging allows system administrators to adjust the space devoted to swapping to their installation's requirements, and to change the size of the swapping area conveniently as those requirements change. It also allows individual users to create private paging areas for their own use, which can be created and deleted according to need. The ability to change the size of the paging area, move it to another disk, or create private paging areas means that system administrators can manage disk usage more effectively.

The text segment of any program (the executable image) is paged from the executable file itself. Paging files are only used for the program's private data. Therefore, TRACE/UNIX can swap a text page without copying it to a paging file; it can simply reassign the page, knowing that it a valid copy exists in the original executable file. This reduces startup time for large programs, increases the amount of virtual memory that a given configuration can support, and reduces the I/O activity needed for paging.

TRACE/UNIX provides *copy–on–write* process creation. A new process initially shares its virtual address space with its parent. Sharing continues until until the new process modifies (i.e. writes) to one of the shared pages, which are marked read–only. At this point, TRACE/UNIX creates a private, writeable copy of the page; all unmodified pages are still shared. This reduces the copying required to start a new process, making process startup quicker. It also reduces the total memory requirements, reducing paging activity and improving overall performnace.

6.1.3 Shared Libraries

TRACE/UNIX implements a shared library facility. The linker handles library references by inserting a branch to a single copy of a routine that is always present in virtual memory, rather than by extracting object code from an archive and inserting it in the executable file. This allows an unlimited number of programs to share a single copy of the library routine, rather than replicating the routine within every executable image that references it.

Shared libraries allow more efficient use of disk space and of virtual memory. Because library routines are not included within the executable image, executable files are significantly smaller. Only one copy of the shared library resides in virtual memory at any time, regardless of the number of programs currently running that use the library. This reduces the total demand for virtual memory, and hence reduces paging activity and improves overall performance.

Multiflow's shared library implementation has these features:

- Negligible run-time overhead; no run-time linking is required to use a shared library.
- Source level transparancy; programs do not require source code changes to use shared libraries.
- Updates; shared libraries may be updated without recompiling or relinking programs that use the library.
- Multiple versions; the shared library facility allows multiple libraries (and multiple versions of the same library) to coexist.
- User-created libraries; customers may define and create their own shared libraries.

All standard TRACE/UNIX libraries are available both as shared libraries and as archived libraries. This includes all run-time, system call, I/O, and math libraries for C and FORTRAN. The standard TRACE/UNIX utilities are compiled with shared libraries. Programs written by users may use either the shared libraries or the archived libraries, as requested during compilation and linking. Users may create their own libraries, which may be either standard (archived) libraries or shared libraries.

6.2 Workstations and Network Computing

TRACE/UNIX systems can communicate with virtually any workstation via Ethernet, using either TCP/IP or the DECnet protocol family.

The Network Filesystem option (NFS) allows TRACE users to share disks with other systems on the network. Access to files on other systems is completely transparent. Users never need to know if a given file is local or remote; all application programs will automatically work with remote files without any modification. Remote filesystems are completely integrated into the TRACE/UNIX directory hierarchy. The network filesystem also lets TRACE systems "export" their filesystems for use by other computers on the network. The network filesystem works across many different computer architectures and operating systems. Operating systems that currently support NFS include:

- Virtually all versions of UNIX, including Ultrix;
- MS-DOS for IBM-compatible PCs and others;
- VAX/VMS (support announced for version 5.0);
- IBM's MVS operating system (announced by Sun Microsystems);
- Apollo Computer's Aegis operating system.

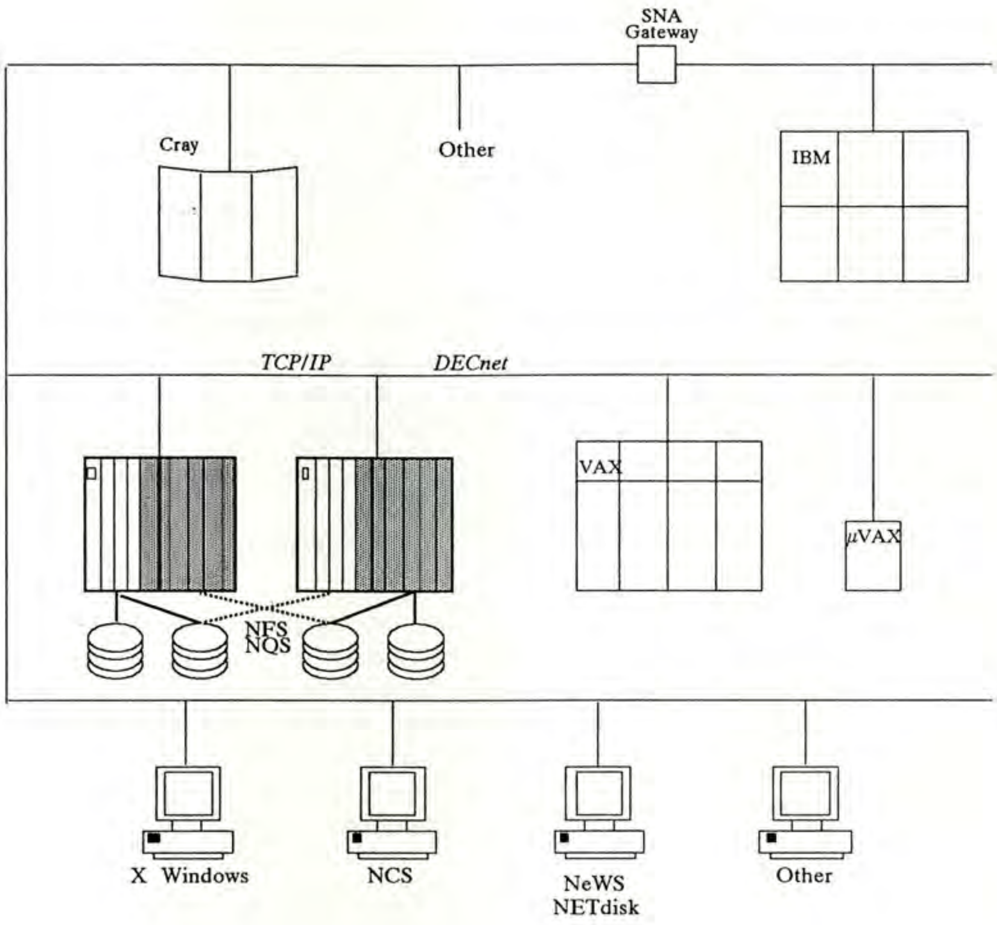
With NFS, the TRACE has completely transparent access to files on all of these different operating systems. It becomes a powerful network computing tool, uniting all of your computing facilities.

Facilities like NCS (the Network Computing System) and RPC (Remote Procedure Call; a part of NFS) let workstation users write software that uses the TRACE as a compute server. Both of these facilities let programs call subroutines and functions that execute on other computers. Both automatically handle data format conversions (for example, byte order conversions) that may be necessary when the caller and the callee are running on different machines of different architectures. Using either of these facilities, software running on a workstation can call a routine (for example, a matrix solver) that runs on the TRACE. This makes it possible to partition a program into different segments that run on different processors, using the capabilities of each to the best advantage.

Two other features, NeWS (the Network Window System) and X (the X window system, version 11) are distributed graphics protocols. They provide standard library interfaces that allow a client program to communicate with a graphics server. By using NeWS or X, TRACE software can effectively use many different graphics systems (including the major graphics workstations) to provide an interactive graphics front end. Both NeWS and X are designed to maximize portability, allowing you to move software between different computing environments with ease.

TRACE/UNIX supports diskless SUN workstations (NETdisk). Diskless nodes can boot from a TRACE, eliminating the need for a SUN fileserver. The size and performance of the TRACE/UNIX filesystem lets it support a large number of workstations gracefully. In this configuration, the TRACE is an ideal compute-server for a workstation network. By using tools like NeWS, X, and RPC, computation can easily be distributed between the TRACE and the workstations.

Other standard UNIX utilities let TRACE users log in to other UNIX systems; execute commands on other UNIX systems; and copy files to and from other systems. The following diagram illustrates the range of TRACE/UNIX communications capabilities:



6.3 System Administration

TRACE/UNIX system administration includes all the basic facilities needed to add new users, accounts, and account groups; change the TRACE's hardware configuration by adding disks, terminals, printers, network controllers, and other peripherals; assign and manage disk quotas; perform full and incremental backups, and restore individual files, groups of files, or entire filesystems from save sets; permit or restrict access to the system through telephone lines and networks; permit or restrict access to files; and other features. TRACE/UNIX may be configured so that only a small group of users can perform administrative tasks; administration can be further restricted so that it can only be performed from certain terminals.

Multiflow has enhanced the standard UNIX administration facilities by adding a project-based accounting system. This facility collects accounting information on a per-user and per-group basis. It reports total user time, system time, elapsed (job execution) time, and connection time. In addition, it reports memory and I/O usage statistics. With other UNIX tools, it can be used as the foundation on which billing programs can be based. Other tools provide printer accounting and connection time accounting.

The Yellow Pages (*yp*) enhances administration of a network of TRACE systems. The yellow pages is a distributed database system designed for administrative purposes. It manages the major UNIX administrative databases, including the user accounts database, the table of network hosts, and the database of user groups. All systems running the yellow pages service can share these major databases. As a result, each system in the network does not need its own copy of the database files; a change to the database (for example, addition of a new user account) can be made once and propagated automatically to the rest of the network. Workstations can also make use of same database and be centrally administered by a master system. *Yp* can also manage site–specific custom databases.

TRACE/UNIX provides the standard UNIX backup utilities (*dump* and *restore*). *Restore* has a new interface that allows administrators to search through an archive interactively and select a group of files for retrieval, using commands similar to the basic TRACE/UNIX file manipulation commands. Optionally, Multiflow supplies a compatible implementation of the VAX/VMS BACKUP utility. All backup and restore utilities support multivolume save sets, support incremental backup, and archive entire filesystem hierarchies. A cartridge tape drive with a capacity of 2 Gigabytes per tape is standard, and provides an ideal backup medium. With this tape drive, multi–volume backup sets are rarely needed. The small size of the cassette (2.5" X 4" X .75") greatly simplifies archival storage requirements.

CHAPTER 7
THE TRACE/UNIX USER ENVIRONMENT

The greatest strength of UNIX, and the primary reason for its acceptance as the standard programming environment for research and engineering, is the wealth of tools that it provides. These tools greatly speed program development. TRACE/UNIX provides all the tools that belong to the Berkeley 4.3 UNIX distribution, in addition to selected tools from the AT&T System V implementation.

7.1 Command Languages

Users interact with TRACE/UNIX through a *shell*. A shell is the operating system's primary user interface; shells are the command interpreters that provide the interactive environment you see when working at a terminal. TRACE/UNIX offers two shells: the Bourne shell (*sh*) developed originally by AT&T, and the Berkeley-developed C shell (*csh*). These shells provide simple, consistent, powerful command of the system.

Both shells perform similar functions and accept similar command languages. They provide:

- **Multiple process management.** Jobs can be run in the *background*, allowing you to run other tasks in the *foreground* while you wait for the background task to complete. The C shell additionally allows you to move jobs between the background and the foreground while they are running.
- **I/O redirection**. TRACE/UNIX defines three default I/O streams: standard input, standard output, and standard error. These streams normally read data from the terminal or send output to a terminal. Streams may be "redirected" to read from a file or write to a file.
- **I/O pipes**. Standard output from one job can be sent to another job's standard input by using a simple "pipe." This allows you to chain a sequence of commands together. For example, the command *job1* | *job2* takes the output from *job1* and "pipes" it to the input of *job2*. The character | represents a "pipe" on the command line.
- **Shell programs**. Both shells provide complete command languages with branching, looping and other control structures. Both shells allow the use of "shell variables." These features allow you to build complex commands from simple utilities by writing command files ("shell programs"). Shell programs are easy to debug, and help programmers to break complex tasks into a number of simple subtasks, each handled by a separate piece of software. In many cases, shell programs can let you perform new tasks with your current software and standard utilities without writing any new FORTRAN or C code.
- **Customization**. The shell lets you customize your working environment by renaming commands and providing your own names for commonly used combinations of commands. Your personal set of abbreviations may be stored in a customization file that is executed automatically when you log in.
- **On-line help**. The *TRACE/UNIX Programmer's Reference* manual is available on-line, and accessible through a simple command, called *man*. Another command provides a keyword search through an index to the manual.
- **History editing**. The C shell can be configured to remember the commands that you have typed. A simple notation lets you recall these commands, modify them, and re-execute them. The history may be extended across different terminal sessions.

- **Timing**. The shell contains a simple command for timing program execution.

7.2 Program Development Tools

TRACE/UNIX provides a range of utilities for software development, including compilers, libraries, debuggers, profilers, and source code management tools.

The *make* utility provides a convenient way to automate compilation for large, complex programs. To use *make*, the programmer describes how to compile the program in a "makefile." The makefile describes what options to use to compile each source file and the dependencies between object modules, header files, and source files. After writing a makefile, you can recompile the program by typing the command *make*. *Make* then looks at the dates of all the source files and object modules; decides which source files have been modified recently, and which object files need to be re–generated; and then executes the compilation commands that are needed to regenerate the relevant object files and link them to a new executable file. This greatly reduces the amount of compilation needed by eliminating needless recompilation of unmodified code; often, only a small part of the program needs to be recompiled after changes. *Make* also reduces the potential for errors in compilation because the makefile specifies which compilation options are required for each module. It therefore provides a record of all the command–line options needed to generate each version of the program.

Software configuration and revision control is provided by the Revision Control System (*rcs*) and the Source Code Control System (*sccs*), both of which are included in TRACE/UNIX. These tools manage archives of incremental revisions of source code files. Both let you reconstruct any old version of a file from the archive. They also serve as "librarians" for large projects, requiring programmers to "check out" files for modification and thus preventing two developers from modifying a program simultaneously.

TRACE/UNIX includes two debuggers: *adb* and *dbx*. *Dbx* is a source–level debugger for C and FORTRAN. It provides breakpointing, source–level code and data examination, stack tracebacks, address break, and other powerful debugging features. Address break is a feature unique to TRACE/UNIX; it allows *dbx* to suspend execution whenever the program access a specified location in memory, and is supported by an address break register in hardware. *Adb* is an assembly language debugging utility.

Three program profilers are part of TRACE/UNIX. The *prof* profiler generates a simple profile that totals the amount of time spent in each routine, and each trace within each routine. It points you directly to the part of a program (often, to a few lines within a program) that consume the most execution time, showing you where to begin your tuning efforts. The *gprof* profiler generates a call graph profile, which analyzes how the program's routines call each other; it shows how much time each routine in the program spends on behalf of each of its callers. This information can be invaluable in deciding where inline expansion is appropriate, and in making larger algorithmic improvements to your software. A third kind of profile, an *execution histogram*, is a list of program counter values that the program executed while running. The TRACE/UNIX linker (*ld*) uses this histogram to optimize the placement of routines within an executable file, minimizing cache conflicts. By optimizing instruction cache usage, the linker maximizes the efficiency with which your programs use the TRACE. A profiling tool, *pf*, lets you enable or disable any kind of profiling without recompilation or relinking. When disabled, profiling has no impact on program performance.

Other utilities include infinite–precision desk calculator programs (*dc* and *bc*); mail handlers; file handling utilities for sort, search, merge, and compare; a database facility (*dbm*); a printer spooling system; a mail system that can be configured to participate in a nation-wide mail forwarding network and bulletin board service; and many more features. More advanced tools include a pattern scanning and processing language (*awk*), a lexical analyzer (*lex*), and a compiler generator (*yacc*).

TRACE/UNIX includes an on-line help facility that provides complete documentation for all utilities, system calls, libraries, file formats, and management tools. The on-line documentation system also includes a keyword search facility.

7.3 Batch Processing

TRACE/UNIX includes the network queueing system (NQS). This is a batch processing system designed for use in a network environment. NQS provides batch job submission queues and print queues. Queues may span different computers on a network, allowing a user to submit a job on one TRACE for execution on another. Queues may be configured for load balancing, automatically submitting jobs for execution on the TRACE with the least activity. Jobs can be assigned different priorities within a queue; and queues may have different priorities relative to each other.

The batch processing system is robust. Queues remain configured after a system shutdown or a crash. After the system has rebooted, any jobs that were in progress when the system went down are restarted automatically. Jobs that were waiting in the queue but not yet running when the crash occurred are completely unaffected.

Queue management is provided by a tool named *qmgr*. This is a command-based program for creating, configuring, and manipulating queues. NQS queues may be configured to enforce limits on total CPU time, run-time scheduling priority, and virtual memory usage. NQS also enforces limits on the number of batch jobs that can run simultaneously.

By using the TRACE/UNIX *cron* facility, a system administrator can configure queues whose properties change dynamically (for example, nighttime queues, weekend queues, low priority queues).

7.4 Text Processing

In addition to the standard UNIX editor, *vi*, TRACE/UNIX supplies the *jove* editor (a version of the popular Emacs editor) and *GNU Emacs*, another version of *emacs* developed by the Free Software Foundation. All three editors are part of the standard TRACE/UNIX system. Both *emacs* versions are extremely flexible and powerful, letting users define their own commands and editing environments by writing customized commands and macros. *GNU Emacs* incorporates special modes for editing text, FORTRAN, C, and LISP source code. These modes automatically enforce stylistic conventions like indentation; they also make some simple correctness checks (for example, matching parentheses).

Other standard editing tools include *sed*, a programmable "stream editor" designed for use in shell scripts, and several line-oriented editors. A version of Digital Equipment's popular EDT editor is also available as part of TRACE/DECLARE.

As a standard feature, TRACE/UNIX includes AT&T's "writer's workbench" software. This package includes device-independent *troff*, an extremely flexible typesetting program that can prepare documents for most laser printers, and *nroff*, a tool for previewing *troff* documents on a terminal, or printing them on a standard line-printer. Standard macro sets for *nroff* and *troff* define easy-to-use formats for memoranda, UNIX-style reference manual entries, academic papers, and view-graphs. The writer's workbench includes a graphics preprocessor that can create line drawings from a simple description; an equation preprocessor that lets you use a set of abbreviations and commands to describe complex mathematical equations conveniently; and a table preprocessor that can produce complex tables and charts. All three preprocessors are integrated into the *troff* typesetting package. Spelling checkers, style analysis tools, and a bibliographic database utility complete the TRACE/ UNIX writing environment.

CHAPTER 8
THE DECLARE USER ENVIRONMENT

Multiflow's DECLARE Compatibility Suite integrates the TRACE seamlessly into the VAX/VMS environment. It demonstrates Multiflow's commitment to the user: TRACE/DECLARE adapts the TRACE to the needs of your site's user community, rather than forcing you to adapt to a new operating system. It eliminates the need for personnel retraining, code conversion, and specialized communications interfaces. VAX/VMS users can be productive on the TRACE immediately. Thus, TRACE/DECLARE minimizes lost time, eliminates conversion expenses, and protects your current investment in software and equipment while increasing your computational power. It links the power of TRACE supercomputers to the familiarity and convenience of the VAX environment.

The TRACE/DECLARE compatibility suite integrates the TRACE into a VMS installation in three ways: by providing a compatible user environment; by providing programming compatibility; and by compatible networking.

The TRACE/DECLARE compatibility suite consists of five major components:

- **TRACE/DCL** is an interactive command environment that provides a VMS-compatible user interface. It eliminates the need for retraining users in TRACE/UNIX.
- **TRACE/EDT** is a version of the popular EDT editor. Together with TRACE/DCL, TRACE/EDT provides a familiar and comfortable environment for users new to the TRACE.
- **TRACE/DN** is Multiflow's implementation of DECnet Phase IV networking. It gives the TRACE users access to VAX/VMS files, and lets VAX users access files that are stored on the TRACE.
- **Multiflow FORTRAN** is Multiflow's FORTRAN 77 compiler. Many features have been added to provide compatibility with VMS FORTRAN. A VMS-compatible system call library has also been added. As a result, programs written in VMS FORTRAN can be ported to the TRACE with minimal effort.
- **TRACE/DECLARE Utility Package** consists of implementations of the VAX/VMS BACKUP and COPY programs. These allow you to read and write tapes in all standard VAX/VMS tape formats, making data interchange between TRACE systems and VAX/VMS computers trivial.

Perhaps the most important feature of the TRACE/DECLARE environment is the high degree of integration between different tools. Without this integration, the package would be of limited use. All the tools within TRACE/DECLARE are compatible with each other, and with the standard TRACE/UNIX utilities. FORTRAN programs can use VMS file specifications and access VAX/VMS files via TRACE/DN without any special handling; TRACE/DCL uses TRACE/DN to access VAX/VMS files automatically; TRACE/DCL also provides access to the standard TRACE/UNIX utilities. TRACE/DECLARE is more than a random collection of tools; it is a complete working environment that has been designed to let VAX/VMS users be maximally productive immediately, without retraining.

8.1 TRACE/DCL

TRACE/DCL gives VAX/VMS users a familiar command environment. It allows users to give most commonly used VAX/VMS commands and qualifiers, in addition to providing access to native TRACE/UNIX commands and utilities. It provides an ideal environment within which to use the other parts of the DECLARE compatibility suite.

TRACE/DCL provides:

- Most user oriented commands from DCL, along with the most commonly used qualifiers.
- An on-line help facility that is equivalent to the VAX/VMS help facility. It provides detailed information about all available TRACE/DCL commands and their qualifiers.
- VAX/VMS file specification syntax, including wildcards and node names. It is integrated into TRACE/DN; file specifications that include a node name automatically refer to files on remote nodes.
- VAX/VMS symbols. Symbols can be used to define abbreviations for commonly used commands, TRACE/UNIX commands, etc. Symbols can also be used as variables within command files. TRACE/DCL provides full support for all VMS operators.
- Full support for VAX/VMS logical names, including system, group, process, and group logical name tables. This provides a way to abbreviate frequently used character strings, directory specifications, etc. Pre-defined logical names like SYS$OUTPUT and SYS$INPUT may be used to redefine defaults.
- Most VAX/VMS lexical functions. These provide system and process manipulation capabilities.
- VAX/VMS command editing and command history features. These features allow users to retrieve, correct (or otherwise modify) and re-issue earlier commands.
- Command files, and all DCL command file control structures (IF, GOTO, GOSUB, CALL, and others). Command files provide TRACE/DCL with a programming environment equivalent to DCL. Command files may mix TRACE/DCL commands and command names with TRACE/UNIX commands.
- System-wide and individual initialization (LOGIN.COM) files.
- A TEACH facility, designed to help VAX/VMS users learn TRACE/ UNIX. The TEACH facility automatically shows the TRACE/UNIX equivalent to each VAX/VMS command, helping new users to understand how the two operating environments correspond.

TRACE/DCL is integrated into the NQS, the TRACE's batch processing facility; it lets you submit batch jobs and manage batch job queues with the standard VAX/VMS queuing commands.

Within TRACE/DCL, the default editor is TRACE/EDT, if it is available. Users who prefer other editors may use any of the standard TRACE/UNIX editors, including the two Emacs versions, *jove* and *GNU Emacs*.

Users who want to work completely within a VMS-like environment can make TRACE/DCL their default command shell. Other users may switch freely between the TRACE/UNIX command shells and TRACE/DCL.

8.2 TRACE/EDT

TRACE/EDT is a complete implementation of the EDT editor, the most popular editor available under VAX/VMS. By using TRACE/EDT, new users can start working on the TRACE immediately, without learning a new editor.

TRACE/EDT implements virtually all of EDT's features, including:

- All three EDT command modes (keypad mode, nokeypad mode, and line mode).
- Gold Key editing, emulating all gold-key features. In keypad mode, all TRACE/EDT editing commands are available with two keystrokes or less.
- All screen editing features, including global search and replace, cut and paste, case changes, open line, search direction changes, etc.
- Entity-based editing, allowing the user to apply basic editing commands to characters, words, lines, regions, and other pre-defined entities.
- Text formatting commands, including automatic word wrapping, filling, justification, and structured tabs.
- Journal files. Journal files allow TRACE/EDT to reconstruct changes to a file if an editing session is interrupted by a power loss or some other failure.
- On-line help facility equivalent to the help provided by DEC's EDT editor. The help facility contains over 200 help screens, and provides general and detailed information about all three editing modes.
- Customization. Users may define special keys for frequently used command sequences. Users may write macros to perform specialized editing functions. TRACE/EDT allows both individual and system-wide customization files (EDTINI.EDT files), allowing administrators to define a site- and user-specific editing environment within EDT.

VAX/VMS users who are familiar with the widely-used Emacs editor may also use either *jove* or *GNU Emacs* within the TRACE/DECLARE environment. Both of these editors are part of the standard TRACE/UNIX distribution, and available at no extra cost. Conversely, TRACE/EDT may be used within the TRACE/UNIX environment, providing a familiar editing environment for users who prefer to work within UNIX.

8.3 TRACE/DN

TRACE/DN supports Phase IV of the Digital Network Architecture, commonly known as DECnet, as an end node. With TRACE/DN, TRACE systems can participate fully in DEC installations; a TRACE becomes a fully functional end node, with complete access to every VAX on the network. Users can use TRACE/DN to access files, to log-in to VAX/VMS systems, etc. TRACE/DN is fully integrated into TRACE/DCL and Multiflow FORTRAN. Integration into the FORTRAN I/O library gives FORTRAN programs direct access to files on VAX/VMS systems, without modification. Integration into the TRACE/DCL command shell lets users work with VAX/VMS files transparently; all TRACE/DCL commands automatically use TRACE/DN when given a file specification that refers to a file on a remote node. TRACE/DN may also be used within the TRACE/UNIX command environments, giving UNIX users complete access to VAX/VMS systems and files.

TRACE/DN implements the following functions:

- Server processes equivalent to DEC's NETACP, NML, REMACP, and FAL.

- NFARS (Network File Access Routines); a library that allows programs on the TRACE to access files on VAX/VMS systems, using DAP. NFARS is integrated into the FORTRAN I/O library, and is available as a separate C library.
- A version of the NCP administrative utility. *Ncp* is completely compatible with the VAX/VMS network management tool, NCP.
- Support for task–to–task communications via standard UNIX system calls. Programs running on a TRACE can exchange data with programs running on VAX/VMS systems.

All TRACE/UNIX systems also support, as a standard, the TCP/IP network protocol. Therefore, TRACE systems can serve as a "bridge" (or "gateway") between DECnet hosts and TCP/IP networks.

8.4 Multiflow FORTRAN

Multiflow FORTRAN provides many features for compatibility with VAX/VMS FORTRAN. These extensions to the FORTRAN 77 standard make it easy to port software to the TRACE from VAX/VMS environments. The extent of Multiflow's support for VMS FORTRAN indicates Multiflow's complete commitment to the VAX/VMS programmer.

Chapter 3 provides a more complete description of Multiflow FORTRAN. The list below summarizes the VMS FORTRAN compatibility features:

- Extended VMS character set (uppercase, lowercase, special characters) and 32–character variable names.
- TAB–formatted source and continuation lines.
- Extended–length (132 column) source lines supported with a command line option.
- INTEGER*2, INTEGER*1, BYTE, LOGICAL*1, LOGICAL*2, and DOUBLE COMPLEX data types.
- NAMELIST I/O.
- IMPLICIT NONE statement.
- ACCEPT, TYPE, ENCODE, and DECODE statements.
- VMS File organization and record type keywords supported.
- DO WHILE and ENDDO statements, extended range DO–loops.
- Direct support for Asynchronous I/O.
- Remote file access over DECnet.
- Automatic conversion of DEC floating point and integer data formats when accessing VAX/VMS files.
- Argument list built–in functions (%VAL, %REF, %DESCR, and %LOC) provided.
- Bit function intrinsics compatible with VMS FORTRAN and Military Standard FORTRAN.
- PARAMETER statement without parentheses.
- Q and $ edit descriptors in FORMAT statements.
- Comments beginning with exclamation points.
- D debugging lines.
- Data initialization in type statements.

- DATA statements may be mixed with executable statements
- Octal and Hexadecimal constants may be used in DATA statements.
- Full Hollerith support, including use of Hollerith constants as integers.
- & prefix to labels.
- FORTRAN-66 compatibility option.
- VMS system call library including mathematical functions, string manipulation, error handling, logical names, inter-process communication, file manipulation, and other features.

In addition, the TRACE provides a source-level symbolic debugger, tools for program analysis and profiling, cross-reference listings, and automatic compilation and source control systems (similar to DEC/MMS and DEC/CMS respectively).

8.5 TRACE/DECLARE Utilities

The DECLARE utility package provides tape compatibility between Multiflow TRACE systems and VAX/VMS. It allows the TRACE to read and write tapes in VAX/VMS formats, making it simple to move data, source code, tape archives, backups, and other data between systems. It simplifies tape management by letting you use a single tape format, readable by all systems, for your backup and archive tapes.

The package consists of two utilities: *vmscopy* and *backup*. *Vmscopy* handles VAX/VMS COPY format tapes, and is integrated into the TRACE/DCL COPY command. *Backup* handles VAX/VMS BACKUP tapes, and is integrated into the TRACE/DCL BACKUP command. They provide the following features:

VMSCOPY:

- Reads and writes variable-and fixed-record length tapes.
- Upon request, pads all records to a fixed record length.
- Reads tapes without ANSI headers. (These are "header-less" tapes; on a VAX/VMS system, they would be mounted FOREIGN).
- Can skip files on the tape if desired.
- Writes tapes in VMS FILES-11 format.
- Can read tapes containing SOS files. Upon request, it can either retaining or discarding the SOS line numbers.

BACKUP:

- Reads and writes VAX/VMS save sets.
- Reads and writes multiple volume save sets.
- Notifies the user when a new volume needs to be mounted.
- Saves and restores directory hierarchies; files are saved together with their position in a directory structure.
- Provides verification and file comparison operations. These operations are equivalent to the VAX/VMS /VERIFY and /COMPARE qualifiers.
- Supports special handling: operations on selected files only, exclusion of certain files from backup or restore operations, deletion of files after they have been written, and confirmation of operation on a file-by-file basis. These operations are equivalent to the VAX/VMS /SELECT, /EXCLUDE, /DELETE, and /CONFIRM qualifiers.

- Translates text files between the VAX/VMS and TRACE/UNIX formats.
- Supports VAX/VMS file and directory naming conventions; automatically converts file names to a form appropriate for the target system.

CHAPTER 9
SERVICE AND RELIABILITY

9.1 Reliability

In the two years since Multiflow's first shipment, the TRACE 200 series has proven to be an exceptionally reliable platform. The integrity of the software and hardware has produced a system that rarely experiences failure of any kind. It is common for a TRACE to run for months without any unscheduled interruption of service.

Total system MTBF (mean time between failures) in the field has been substantially over 4100 hours. This figure includes all systems shipped to date; in particular, it includes our beta-test systems for both the 200 and the 300 series. Our early 300 series customers have found it to be as reliable and stable as our earlier products. CPU-only MTBF is in excess of 15,000 hours. MTBF for each peripheral device and controller is in excess of 30,000 hours. Few products achieve these levels of reliability in their lifetime, let alone in the first years after their introduction.

9.2 Diagnostic Environment

The I/O Subsystems run a small, UNIX-like operating system called MDX (Multiflow Diagnostic Executive). MDX performs power-on testing and is responsible for booting the TRACE. It accesses a dedicated disk partition with an independent file system that contains bootstrap programs, diagnostics, and error logs.

The MDX environment simplifies running and developing diagnostic tools. Through the I/O subsystem, it has access to the TRACE Diagnostic Bus, a special diagnostic channel that probes every part of the TRACE. Diagnostic programs have direct access to every signal on the TRACE gate arrays, access to the instruction cache, most special registers, many internal buses, and other logic throughout the system.

The diagnostic bus also also performs board configuration. There are no switch settings or jumpers in the CPU or memory boards. This allows remote configuration and testing.

The MDX environment can be accessed from the console terminal or through the diagnostic modem associated with the diagnostic processor. The diagnostic modem allows Multiflow customer support engineers to access the diagnostic system remotely, run diagnostics, and diagnose problems. The ability to diagnose problems prior to making a service call substantially decreases the time to repair.

9.3 Environmental Processor

The TRACE Environmental Processor (EP) monitors operating conditions and enforces safety limits. It monitors all power supply voltages, temperature, air flow, and AC power. The EP controls AC power and power supply DC voltage settings. It can select high and low margin voltages and clock speeds for testing. The high and low margins insure that the TRACE operates safely under a wide range of abnormal conditions.

The environmental processor includes a modem that allows customer support operations to perform many operations (including power-on and power-off) remotely.

9.4 Error Detection and Correction

TRACE systems are intended to handle enormous computations with large numbers of hardware functional units. Accordingly, Multiflow has taken great care in the system design to assure computational integrity, and to eliminate undetected transient errors.

All general registers, buses, and major datapaths incorporate parity checking. Parity hardware verifies that data is correctly transmitted and stored. Byte parity is computed on-chip when functional units generate results, and checked on-chip when functional units accept operands. Parity flows throughout the system and is stored in registers, for end-to-end protection. To avoid impacting the system cycle time, parity is transmittedand checked one clock period following the data.

Main memory is protected by error detection and correction logic which automatically corrects all single-bit errors and detects all double-bit errors. Seven extra bits are stored for each 32-bit field stored in memory, providing enough redundancy to allow all single-bit errors to be transparently corrected, and all double-bit errors to be detected. A memory scrubbing protocol ensures that all of main memory is accessed and rewritten with corrected data on a regular basis. Error correction significantly improves system reliability, as it compensates for the primary source of errors in modern computer systems: soft errors (dropped bits) in dynamicRAMs.

The Instruction Cache and address translation buffers detect and correct single-bit errors. Byte parity is stored with the cache data; parity errors are treated as cache misses, and the cache entry is transparently reloaded from the backing store. This technique has a similar impact on overall system uptime; occasional static RAM errors can be tolerated.

APPENDIX A BIBLIOGRAPHY

This section gives a limited bibliography of academic publications discussing TRACE computer systems, the VLIW architecture, the Trace Scheduling compacting compiler, and TRACE/UNIX.

Clancy, Patrick, Benjamin F. Cutler, J. C. Dodd, *et. al*. "UNIX^{tm} on a VLIW." *USENIX Conference Proceedings.* June, 1987. pp. 225–241.

Clancy, Patrick. "Virtual Memory Extensions in TRACE/UNIX^{tm}." *Proceedings: USENIX Workshop on UNIX and Supercomputers."* September, 1988. pp. 137–150.

Colwell, Robert P., Robert P. Nix, John J. O'Donnell, David B. Papworth and Paul K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler." *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems [ASPLOS–II] October 5–8, 1987.* Computer Press of the IEEE, pp. 180–92, 1988.

Durocher, Lawrence, and V. Urbach. "Evaluating High–Performance Computers for Finite Element Analysis." *Computers in Mechanical Engineering*. July/August, 1988. pp. 48–55

Ellis, John R., Joseph A. Fisher, John C. Ruttenberg, *et. al.* "Parallel Processing: A Smart Compiler and a Dumb Machine." *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*. June, 1984.

Ellis, John R. *Bulldog: A Compiler for VLIW Architectures.* Cambridge: MIT Press. 1986.

Fisher, Joseph A. "The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources." *Technical Report COO–3077–161.* New York: Courant Mathematics and Computing Laboratory. October, 1979.

-----. "Trace Scheduling: A Technique for Global Microcode Compaction." *IEEE Transactions on Computers*. Vol. C–30 [July, 1981]. pp. 478–490.

-----. "Very Long Instruction Word Architectures and the ELI–512." *Proceedings of the 10th Symposium on Computer Architectures*. June, 1983. pp. 140–150.

-----. "Wide Instruction Word Architectures: Solving the Supercomputer Software Problem." *Proceedings of the INRIA International Seminar on Scientific Supercomputers.* February, 1987.

-----. "Microprogramming, Microprocessing and Supercomputing." *Microprocessing and Microprogramming* 24 (1988). pp. 17–20.

-----. "Backing the Right Technology." *Proceedings of the International Congress on Technology and Technology Exchange.* 1988. pp. 18–20.

Fisher, Joseph A. and John J. O'Donnell. "VLIW Machines: Multiprocessors We Can Actually Program." *CompCon 84 Proceedings*. 1984. pp. 299–305.

Lubeck, Olaf M. *Supercomputer Performance: The Theory, Practice, and Results.* Los Alamos National Laboratory Report LA–11204–MS UC–32. January, 1988.

MULTIFLOW

MULTIFLOW COMPUTER, INC. / 31 Business Park Drive / Branford, Connecticut 06405 / (203) 488-6090

102665799