A VLIW Architecture for a Trace Scheduling Compiler

Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, Paul K. Rodman

Multiflow Computer
175 North Main Street
Branford, CT. 06405
(203) 488-6090

1. Abstract

Very Long Instruction Word (VLIW) architectures were promised to deliver far more than the factor of two or three that current architectures achieve from overlapped execution. Using a new type of compiler which compacts ordinary sequential code into long instruction words, a VLIW machine was expected to provide from ten to thirty times the performance of a more conventional machine built of the same implementation technology.

Multiflow Computer, Inc., has now built a VLIW called the TRACE™ along with its companion Trace Scheduling™ compacting compiler. This new machine has fulfilled the performance promises that were made. Using many fast functional units in parallel, this machine extends some of the basic Reduced-Instruction-Set precepts: the architecture is load/store, the microarchitecture is exposed to the compiler, there is no microcode, and there is almost no hardware devoted to synchronization, arbitration, or interlocking of any kind (the compiler has sole responsibility for runtime resource usage).

This paper discusses the design of this machine and presents some initial performance results.

2. Background for VLIWs

The search for usable parallelism in code has been in progress for as long as there has been hardware to make use of it. But the common wisdom has always been that there is too little low-level fine-grained parallelism to worry about. In his study of the RISC-II processor, Katevenis reported[Kate85] "...We found low-level parallelism, although usually in small amounts, mainly between address and data computations. The frequent occurrence of conditional-branch instructions greatly limits its exploitation."

This result has been reported before [Tjad70, Fost72] and judging from the lack of counterexamples, seems to have been interpreted by all architects and system designers to date as a hint from Mother Nature to look elsewhere for substantial speedups from parallelism.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Researchers at Yale, however, [Fish83, Elli86] found that fine-grained parallelism could be exploited by a sufficiently clever compiler to greatly increase the execution throughput of a suitably constructed computer. The compiler exploited statistical information about program branching to allow searching beyond the obvious basic blocks in a program (e.g., past conditional branches) for operations that could be performed in parallel with other possibly-unrelated operations.[Fish79] Logical inconsistencies that were created by these motions were corrected by special compensation code inserted by the compiler.

These researchers labelled their proposed architecture "Very-Long-Instruction-Word", and suggested that a single-instruction-stream machine, using many functional units in parallel (controlled by an appropriately large number of instruction bits) would be optimal as an execution vehicle for the compiler. It was proposed that the most suitable VLIW should exhibit four basic features.

- One central controller issues a single long instruction word per cycle.
- Each long instruction simultaneously initiates many small independent operations.
- Each operation requires a small, statically predictable number of cycles to execute.
- Each operation can be pipelined.

In the same spirit as RISC efforts such as MIPS [Henn81] and the IBM 801 [Radi82] the microarchitecture is exposed to the compiler so that the compiler can make better decisions about resource usage. However, unlike those efforts, a VLIW provides many more functional units that can be used in parallel; Multiflow's Trace Scheduling compiler finds parallelism across basic blocks to keep them busy.

Multiflow Computer, Inc., has now demonstrated the fundamental soundness of both the compiler and the architecture, announcing a product based on these concepts. This paper will discuss Multiflow's TRACE architecture. Some initial experience with programming a VLIW (bringing up UNIX on the TRACE machine) will be recounted.

TRACE, Trace Scheduling, and Multiflow are trademarks of Multiflow Computer, Inc. UNIX is a registered trademark of AT&T Technologies. VAX and VMS are registered trademarks of Digital Equipment Corporation. IBM is a registered trademark of International Business Machines Inc.

© 1987 ACM 0-89791-238-1/87/1000-0180 $00.75

3. Introduction to VLIW Computer Architecture

VLIW computers are a fundamentally new class of machine characterized by

- A single stream of execution (one program counter, and one control unit).

- A very long instruction format, providing enough control bits to directly and independently control the action of every functional unit in every cycle.

- Large numbers of datapaths and functional units, the control of which is planned at compile time. There are no bus arbiters, queues, or other hardware synchronization mechanisms in the CPU.

Unlike a vector processor, no high level regularity in the user's code is required to make effective use of the hardware. And unlike a multiprocessor, there is no penalty for synchronization or communication. All functional units run completely synchronized, directly controlled in each clock cycle by the compacting compiler.

The true cost of every operation is exposed at the instruction set level, so that the compiler can optimize instruction scheduling. Pipelining allows new operations to begin on every functional unit in every instruction. This exposed concurrency in the hardware allows the hardware to always proceed at full speed, since the functional units never have to wait for each other to complete. Pipelining also speeds up the system clock rate. Given that we can find and exploit scalar parallelism, there is less temptation to try to do "too much" in a single clock period in one part of the design, and hence slow the clock rate for the entire machine. Judiciously used small scale pipelining of operations like register-to-register moves and integer multiplies, as well as the more obvious floating point calculation and memory reference pipelines, helped substantially in achieving a fast clock rate.

The absence of pipeline interlock or conflict management hardware makes the machine simple and fast. Hennessy [Henn81] has estimated that building the MIPS chip without interlocked pipeline stages allowed that machine to go 15% faster. In our VLIW, given our large number of pipelined functional units, pipeline interlocking and conflict resolution would have been almost unimplementable, and the performance degradation would have been far greater than 15%.

VLIWs exploit the same low-level, scalar parallelism that high-end scalar machines have used for decades. Execute-unit schedulers which look ahead in a conventional instruction stream and attempt to dynamically overlap execution of multiple functional units were incorporated in systems beginning with the IBM 360/91 [Toma82] and the Control Data 6600. [Thor70] These "scoreboards" perform the same scheduling task at runtime that Multiflow's Trace Scheduling compacting compiler performs at compile time. Even with such "complex and costly hardware", Acosta et al. [Acos86] report that only a factor of 2 or 3 speedup in performance is possible. This limitation, of course, is the same as previously discussed: the hardware cannot see past basic blocks in order to find usable concurrency.

Over the last two decades, the cost of computer memory has dropped much faster than the cost of logic, making the construction of a VLIW, which replaces scheduling logic with instruction-word memory, practical and attractive. In conjunction with the global optimization ability of our compiler, we find no remaining reasons to build run-time scheduling hardware. The scheduling problem is much better solved in software at compile-time. This "no control hardware" attitude permeates the design of the TRACE architecture.

An obvious potential disadvantage to the VLIW approach is that instruction code object size could grow unmanageably large, enough so that much of the performance advantage would be lost in extra memory costs and disk paging. We have addressed this problem in the design of the TRACE, and have a very satisfactory result to report in Section 9.

4. Trace Scheduling Compacting Compilation

Multiflow's Trace Scheduling compacting compiler automatically finds fine-grained parallelism throughout any application. It requires no programmer intervention, either in terms of restructuring the program so it fits the architecture or in adding directives which explicitly identify opportunities for overlapped execution. This is in sharp contrast to "coarse-grained" parallel architectures, including vector machines, shared-memory multiprocessors, and more radical structures such as hypercubes [Seit85] or massively-parallel machines. [Walt87] The process by which programs are converted into highly parallel wide-instruction-word code is transparent to the user.

To detect fine-grained parallelism, the compiler performs a thorough analysis of the source program. One subroutine or module is considered at a time. After performing a complete set of "classical" optimizations, including loop-invariant motion, common subexpression elimination, and induction variable simplification, the compiler builds a flow graph of the program, with each operation independently represented.

Using estimates of branch directions obtained automatically through heuristics or profiling, the compiler selects the most likely path, or "trace", that the code will follow during execution. This trace is then treated as if it were free of conditional branches, and handed to the code generator. The code generator schedules operations into wide instruction words, taking into account data precedence, optimal scheduling of functional units, register usage, memory accesses, and system buses; it emits these instruction words as object code. This greedy scheduling causes code motions which could cause logical inconsistencies when branches off-trace are taken. The compiler inserts special "compensation code" into the program graph on the off-trace branch edges to undo these inconsistencies, thus restoring program correctness.

This process allows the compiler to break the "conditional jump bottleneck" and find parallelism throughout long streams of code, achieving order-of-magnitude speedups due to compaction.

The process then repeats; the next-most-likely execution path is chosen as a trace and handed to the code generator. This trace may include original operations and compensation code. It is compacted; new compensation code may be generated; and the process repeats until the entire program has been compiled.

A number of conventional optimizations aid the trace selection process in finding parallelism. Automatic loop unrolling and automatic inline substitution of subroutines are both incorporated in Multiflow's compilers; the compiler heuristically determines the amount of unrolling and substitution, substantially increasing the parallelism that can be exploited.

More information about the design of the compiler will be forthcoming; interested readers may also find previous reported research enlightening. [Fish79, Elli86, Fish81, Fish84, Elli84]

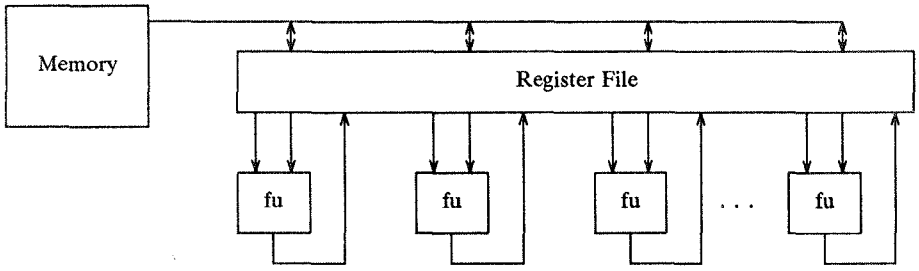

Figure 1. Block Diagram of an Ideal VLIW Execution Engine

5. The Ideal VLIW

The ideal execution vehicle for this compacting compiler would be a machine with many functional units connected to a large central register file. Each functional unit would ideally have two read ports and one write port to the register file, and the register file would have enough memory bandwidth (Very Large) to balance the operand usage rate of the functional units. A block diagram of this ideal engine is shown in Fig. 1. A centralized register file would simplify code generation; when scheduling operations, the selection of functional unit would be unimportant, and the code generator would only have to worry about when the operation was scheduled. The provision of a full set of separate read/write ports for each functional unit would guarantee the independence of each operation. As long as there were enough registers, no extraneous data movement would ever be needed.

However, any reasonably large number of functional units requires an impossibly large number of ports to the register file. Chip real estate and chip pinout limit the number of independently controlled ports which can be provided on a single register set. The only reasonable implementation compromise is to partition the register files, in some way that minimizes the additional workload on the compiler and also minimizes data traffic between the different register files.

Another problem in this "ideal VLIW" is the memory system. All modern computers have to deal with a significant speed mismatch between the logic used to build the processor and the access time of the dynamic RAMs used to build the memory. The memory architecture of our VLIW is perhaps the area where the greatest advantage is gained over other approaches; it is discussed in Section 6.4.

6. A Real VLIW

These were the goals for the TRACE processor design:

- A modular design, with an expandable number of functional units;
- Use standard, high-volume, low-cost electronics;
- Use standard DRAMs for main memory for high capacity at low cost;
- Deliver the highest possible performance for 64-bit floating point intensive computations;
- Perform well in a multi-user environment.

The processor is built of five board types: integer boards (I), floating point boards (F), a "global controller" (GC), memory controllers (MC), and I/O Processors (IOP). Each board is an 18 by 18 inch, 8 to 10 layer printed circuit, interconnected via a single 19-slot backplane. The backplane connectors provide 630 pins usable for signals, plus power and ground connections. The core of the computational engine was built in 8000 gate CMOS gate arrays with 154 signal pins. Advanced Schottky TTL was used for "glue" logic and bus transceivers.

Research at Yale in machine architecture accompanied research into compilation across basic blocks. The focus of efforts at Yale was to develop buildable, scalable, technology-independent machine models which could be used to evaluate the success of such a compiler. At Multiflow, the design team spent the first year studying alternative partitionings, functional unit mixes, and opcode repertoires, with a specific product and implementation technology in mind. This allowed us to be much more aggressive in hardware support for effective compilation, and come much closer to an "ideal VLIW" structure than any machines we considered at Yale. Architectural alternatives were evaluated using a prototype easily-retargetable compiler and simulator.[Elli86]

Given the implementation constraints, we decided that a maximum of eight 32-bit buses could traverse the edge connector. The number of buses we could support was one of the major constraints on CPU expandability, given the required balance between memory bandwidth and the rate of floating point operations to support general computations.

We partitioned the core processor into an Integer and a Floating unit (the "I" and "F" boards), and provided separate physical register files for the floating point functional units and the integer ALUs. This makes intuitive sense, since there is little need for performing integer operations on floating point operands (and vice versa), while it is often the case that a chain of floating point operations can proceed while the integer units are performing the address computations in parallel.

The unit of processor expansion is this Integer-Floating board pair. One, two, or four I-F pairs can be configured, corresponding to a 256-bit, 512-bit, or 1024-bit instruction word.

Two 32-bit buses carry data traffic between the boards of a pair (through a dedicated front-edge path, rather than the backplane). Each board carries its own register file/crossbar touching twelve 32-bit datapaths, handling four writes, four reads, and four bus-to-bus forwards in each minor cycle, plus bypassing from every write port to every read port. Sixty-four 32-bit registers are provided. This register file/crossbar is implemented in nine gate arrays; each is a 4-bit slice (byte parity is carried throughout the machine).

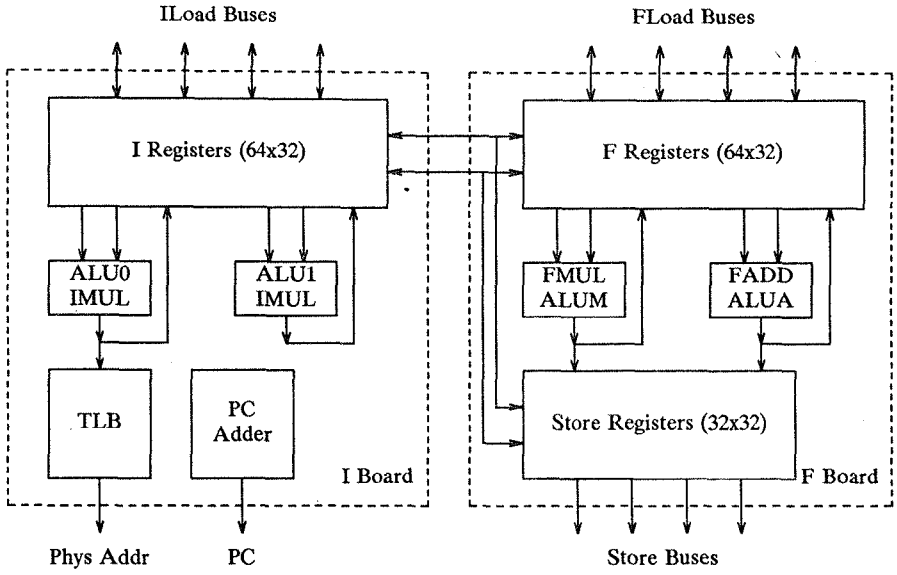

Figure 2. I and F Board Block Diagrams

6.1. Integer Operations

The integer instruction set comprises some 80-odd opcodes, including arithmetic, logical, and compare operations; high performance 16-bit primitives for 32-bit integer multiplication; and shift, bit-reverse, extract, and merge operations for bit and byte field manipulations. The integer instruction set (excluding multiplication) is implemented in a single gate array, four copies of which are included in a single I-F pair.

Every operation specifies its destination register at the time of initiation. A modifier ("dest_bank") specifies which register file contains the target register: the local general register bank, the general register bank in the paired F unit, the store file in the paired F unit, a general register bank in another I unit, or a branch bank on an I or F board. Branch banks are small 1-bit register files used to control branching; see Section 6.5.2.

Substantial support was provided for injecting immediate constants into the computation. Each ALU can get a 6-bit, 17-bit, or 32-bit immediate provided on one operand leg, under the control of the instruction word. A 32-bit immediate field is flexibly shared between ALU0, ALU1, and a 32-bit PC adder which generates branch target addresses.

Included in the I board instruction set are pipelined load and store instructions for referencing memory. Memory addresses are 32-bit byte pointers. The memory system hardware operates only on 32-bit or 64-bit quantities; access to fields of other sizes is provided via extract/merge/shift operations which are arranged to accept the same 32-bit pointer, using the low bits to specify the field position.

The I board also includes dynamic address translation hardware and supports demand-paged virtual addressing. A Translation Lookaside Buffer provides a cache of 4K virtual-to-physical address translations on 8KB page boundaries. Simple paging is used; no segmentation or other address translation is provided. Traps are taken on TLB misses; trap-handling software manages TLB refills. The TLB is process-tagged so that flushes are unnecessary at context switches. Its indexing scheme includes a process-ID hash to minimize conflicts between entries for multiple processes.

Each instruction executes in two minor cycles, or "beats". Each beat is 65ns. The I board ALUs perform unique operations, specified by new control words presented in both the early and late beats.

Figure 3 shows the format of the instruction word for a single I-F pair. This instruction word is replicated four times in a fully configured processor.

6.2. Floating Operations

The F board (floating point) was optimized for 64-bit IEEE standard floating point computation. It uses the same register file chips as the I board, providing sixty-four 32-bit registers (which are used in pairs for 64-bit quantities). The floating functional units each perform one new operation per instruction, or every other beat. The 32-bit datapaths carry 64-bit data to and from the floating point units in two beats.

A pipelined floating adder/ALU shares resources and opcodes with an integer ALU; the integer ALU has one beat latency, while the floating adder has six beat latency in 64-bit mode. A floating multiplier/divider similarly shares resources with another integer ALU. The multiplier has seven beat latency doing 64-bit multiplication, and 25 beat latency doing 64-bit division. New operations may be started on each functional unit in each instruction (except on the multiplier while division is in progress).

The integer instruction set, excluding memory references and integer multiply, is available on both ALUs on the F board. This was an implementation convenience. We found it desirable to provide "fast move" paths to allow data moves without the pipeline depth of the floating point units. We also included the integer SELECT operation, which provides the semantics of the C "?" operator without branching. It was simpler to include copies of the already-designed integer ALU than to dedicate another gate array to these more limited functions.

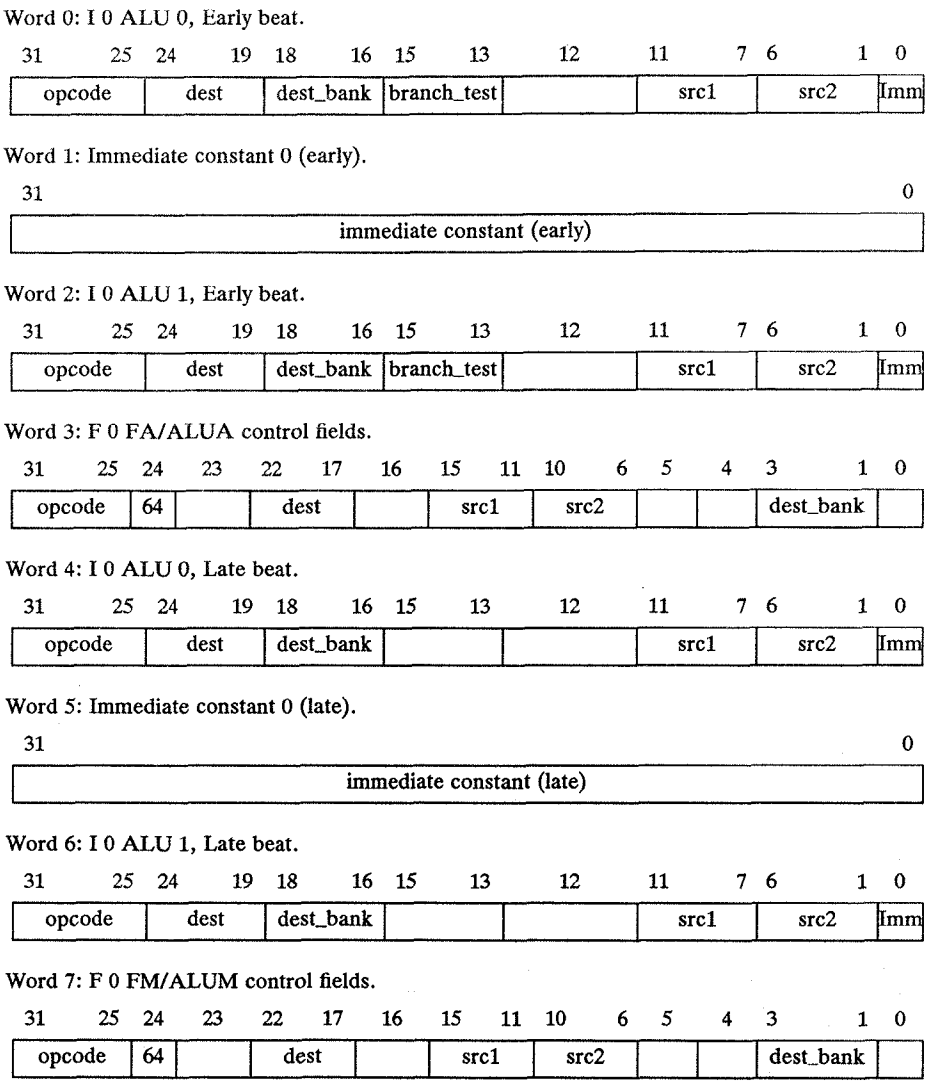

Figure 3. Instruction Word Format for one I-F pair

The floating point functional unit pipelines are "self-draining"; the destination register is specified when the operation is initiated, and a hardware control pipeline carries the destination forward, writing the target register when the operation completes. To allow interrupts to occur at any point in the program, by convention the target register of any pipelined operation is "in use" from the beat in which the operation is initiated until the beat in which it is defined to be written. If a trap or interrupt occurs while the pipelined operation is in process, the register gets written early, relative to the execution of the instructions immediately following in the program text.

The F board also carries the Store Register File. When an I board issues a Store opcode, it also issues a "Store Read Address" on a bus that all F boards monitor. Physical addresses are generated on the I boards, and data to be stored comes from the "Store Register File" on the F boards. The Store Register File, implemented using the same register chip used elsewhere, expands the number of register read ports and eliminates pipeline conflicts between memory stores and other operations.

6.3. System configuration

A fully configured TRACE processor incorporates four I-F pairs. With a 1024-bit instruction word that initiates 28 operations per instruction, it has peak performance of 215 "VLIW MIPS" and 60 MFLOPS. Figure 4 shows the top level architecture and backplane interconnect for the system. The entire CPU and its interconnect is synchronized to a single master clock.

The ILoad Buses, FLoad Buses, and Store Buses are each a set of 4 independently-managed synchronous 32-bit buses. The Load buses are multidirectional, and the Store buses are unidirectional. Each bus is independently scheduled by the compiler for each execution beat. Each Load bus carries a 10-bit control field, or "tag", specifying the destination of the data carried on the bus in that beat; the tags are derived from instruction words which specify data moves or memory references. Because the "arbitration" is handled by the compiler, the buses are fast, simple, and cheap. This is a major advantage versus the complicated interconnects of a multiprocessor, where arbitration, buffering, interlocking, and interrupts are required.[Pfis85]

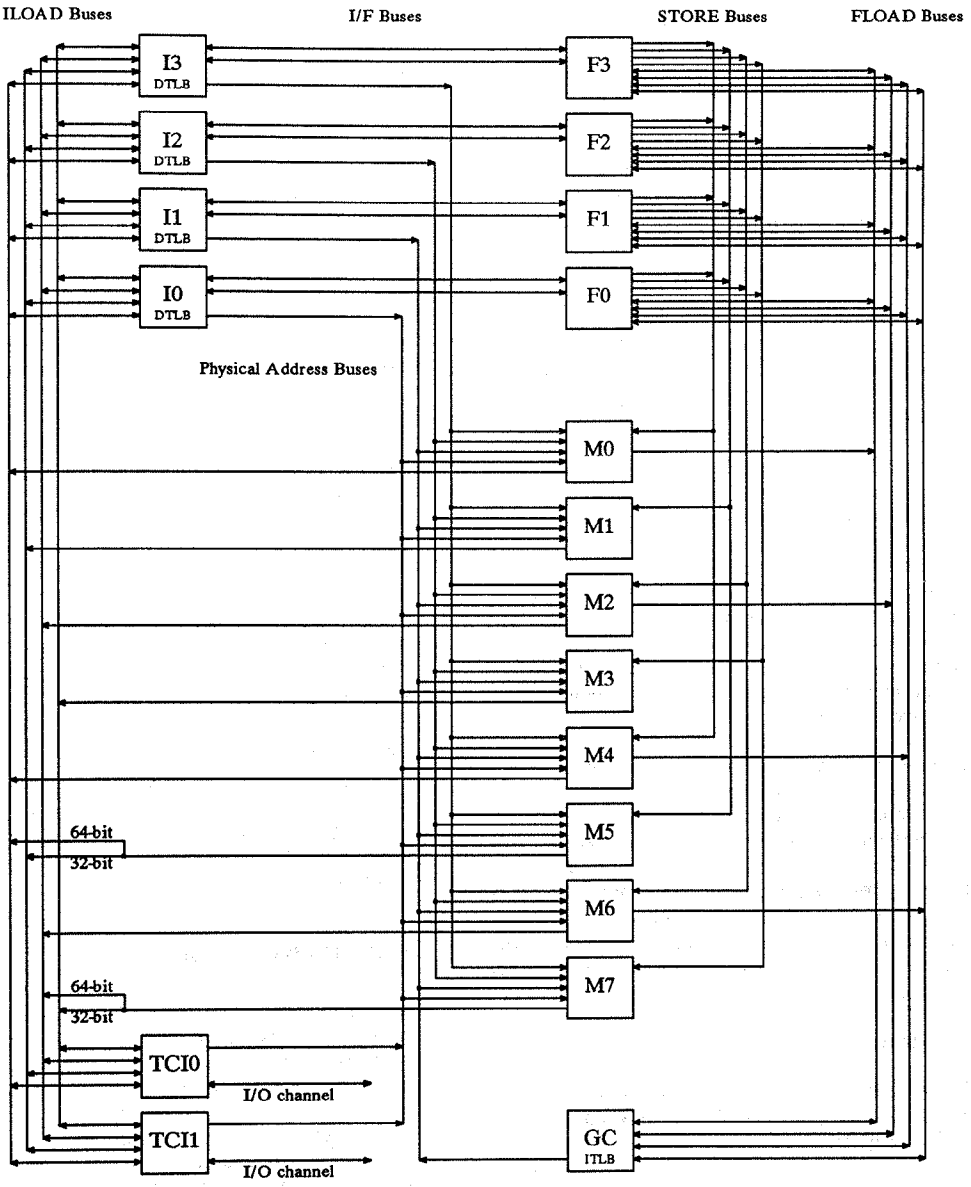

Figure 4. The TRACE Major Datapaths

Up to eight memory controllers comprise the memory system. Each controller carries up to 8 independent banks. Memory addresses are interleaved among controllers and banks. Each memory controller watches all four physical address buses for valid requests, and touches one Store bus, one I bus, and one F bus. A fully populated memory system comprises 512 MBytes of physical storage.

6.4. The Memory Subsystem

The speed of the CPU/memory interconnection in a computer system is a first order determinant of overall performance (the legendary "von Neumann bottleneck"). The designers of every modern computer system, from the IBM PC to the Cray-2, have had to deal with a substantial mismatch in speed between the cycle time of the processor and the access time of the memories. This mismatch ranges from a factor of 2 to a factor of 59. Two major approaches have been taken by the designers of these systems to handle the mismatch: caching and interleaving.

Cache memories provide lower latency than main memory when there is reasonable locality of reference in the access pattern. They can be expensive to implement, and their cost grows rapidly in systems attempting to issue more than one memory reference per cycle. While instruction caches are universally effective, data caches work poorly in many scientific applications, where very large arrays of data are repeatedly accessed; hit rates fall off rapidly, and system performance degrades to the performance of the backing store (main memory).[Smit82]

Interleaved memories exploit parallelism among memory references to address the speed mismatch in a different manner. Memory addresses are spread across multiple independent banks of RAMs. The memory system is pipelined; while one or more new references may be initiated in each cycle, it takes multiple cycles for a single reference to complete. During several of those cycles, a single RAM bank will be tied up and unable to accept new requests; achieving performance requires that addresses be spread across multiple banks. Correctness depends upon somehow managing the referencing pattern, and avoiding references to banks that are busy.

When parallelism can be found in the memory referencing pattern, an interleaved memory system will provide much higher sustained performance for large scientific applications than any cached scheme of comparable cost. However, the management complexities and exposed parallelism of the interleaved approach have prevented all but supercomputer designers from building such memory architectures.

One major problem in building an interleaved memory system is managing the status of several simultaneously outstanding references. In a traditional scalar or scalar/vector computer, a hardware bank scheduler, or "stunt box", is required to track the busy status of each bank, watch each memory reference address, and prevent conflicts (by temporarily suspending all or selected portions of execution). Details such as out-of-order data returns can complicate the picture further. The complexity of such a scheduler grows as the square of the number of pending references to be managed.

Our VLIW computer system provides enormous memory bandwidth using an interleaved memory system, without memory-reference scheduling hardware and without a data cache. This is a major advantage of the VLIW approach in building a scientific computer.

6.4.1. Memory Implementation Details

Software sees a seven beat memory reference pipeline in the TRACE. The pipeline stages look like this:

0. The program says `LD R1, R2, R3`. R1 and R2 are added to form a virtual address. R2 may be replaced by a 6-, 17-, or 32-bit immediate constant.
1. The virtual address is looked up in the TLB.
2. The physical address is sent over the buses to the memory controller.
3. The desired RAM bank starts cycling.
4. RAM access continues.
5. Data is grabbed from the RAMs on the memory controller.
6. Data is sent over the buses to the CPU; simultaneously, ECC is checked.
7. Data is written into the register file, and the CPU can use the data in R3.

Like the floating point pipelines, the memory pipelines are "self-draining"; loads specify the destination register when the operation is initiated, and a hardware control pipeline carries the destination forward, tagging a data bus with it in the cycle when the data is sent to the CPU. This simplifies interrupt handling and the generation of "compensation code" for off-trace branch cases, when compared with the "pusher/catcher" approach found in most horizontally microcoded attached processors.

In a fully configured TRACE, four memory references may be started in each beat, to four independently generated addresses (one per I board). When each of these references is a 64-bit reference, this corresponds to a memory bandwidth of 492 megabytes per second.

However, a number of restrictions must be met:

- At most one reference may be initiated on any individual controller.
- No two references may be initiated which would require the use of the same bus to return their data.
- No two references should be initiated to the same RAM bank within four beats of each other.
- The total number of ILoad, FLoad, or Store buses used must not exceed the number available.
- The available number of register file write ports must not be exceeded.

In order to satisfy some of these requirements, the compiler must know quite a lot about the memory addresses being generated by the program. For example, it must be able to guarantee that the addresses for two simultaneously-issued LOAD operations will never be equal modulo the number of memory controllers.

6.4.2. The Disambiguator

The *disambiguator* is the module of the compiler which passes judgment on the feasibility of simultaneous memory references. Memory reference disambiguation – distinguishing between references which can be to the same location and those which cannot – is required in order to find parallelism among array references. When a loop is unrolled, for example, the disambiguator is called upon to answer whether a store into `C(I)` can be moved above a reference to `C(I+J)`. The disambiguator builds derivation trees for array index expressions and attempts to solve the diophantine equations in terms of the loop induction variables.

Relatively simple extensions to the disambiguator allow the code generator, as it schedules memory references, to ask for any two references, "can these conflict, modulo the number of memory banks"? The answer can be "no", "yes", or "maybe". When the answer is "no", references can be scheduled simultaneously, at very high bandwidth, without any memory-bank management hardware. When the answer is "yes", the operations will not be scheduled simultaneously. When the answer is "maybe", as in the case of references to two arrays passed in as arguments to a subroutine (so that their base addresses are unknown), the compiler has to treat this as a conflict for certain resources, but may overlap the utilization of other resources, because the hardware provides a "bank stall" mechanism (described below).

6.4.3. Virtual Memory

Virtual memory posed a special challenge for a VLIW architecture issuing multiple memory references in every beat. Given that address translation is pipelined, TLB misses are not detected until several beats after the memory reference has been initiated. Since memory reference pipelining is exposed, this presents no problem; no computation could possibly depend upon the result, and we have several cycles in which to determine the correctness of the reference. On a TLB miss, hardware aborts the reference, and signals the processor to switch to Trap Mode to handle the failed reference. However, trap handling code cannot just load the TLB with the appropriate translation and return to the instruction; several more instructions have executed since the original failing operation, and they cannot be correctly reexecuted.

Each I board incorporates a "history queue" mechanism used by the trap handling code, which records uncompleted memory references and their virtual addresses (these memory accesses must then be handled by the trap code). These queues are read and the TLB contents are updated (or page faults are taken); the references are then replayed via special instructions which allow the queue contents to be reissued as new operations. As the queues are four entries deep, up to sixteen independent TLB misses can be pending on a single entry to the trap code.

The trap handling code is standard machine code resident at a specific physical address. It is executed with instruction stream virtual addressing disabled, but is otherwise normal; early versions were written almost entirely in C. No "microcode" is present anywhere in the processor; this is as close as we come to it. This provides great flexibility in the virtual architecture exposed to processes. (For instance, "copy-on-write" is a very simple change to the trap code, not a hardware change.)

6.4.4. Memory Summary and Comparisons to Earlier Work

Several major advances in the memory system beyond earlier research are incorporated in the Multiflow TRACE system:

• Only relative disambiguation is necessary. Unlike earlier proposed VLIW architectures, the presence of a full crossbar between address generators and memory controllers means that the disambiguator need only answer "is <exp1> ever equal <exp2> modulo N", and not "what is the value of <exp1> modulo N". This greatly improves the likelihood of successful disambiguations, particularly in subprograms where array base addresses cannot be known.

• The same datapaths are shared between intra-CPU and Memory-to-CPU traffic. This concentrates the bandwidth and better accommodates the bursty nature of computations, providing higher sustained performance without additional costs.

• A "bank-stall" mechanism was devised. When a given RAM bank is accessed, that bank goes busy for four beats. In cases when the disambiguator answers "maybe" to a bank conflict, the compiler has the option of moving references into potentially conflicting schedule positions. In this case the memory will "bank-stall" the CPU if an actual conflict occurs, until the bank busy time is satisfied. This "rolling the dice" can improve performance.

We believe that this software-managed parallel memory system is an important architectural breakthrough. It allows much higher memory performance than would otherwise be possible. Although a run-time memory reference scheduler has more perfect information than a compile-time disambiguator (it sees no "potential" conflicts, only the real ones) it can do less in the way of scheduling its way around conflicts when they arise. Compile-time scheduling, with a larger perspective on the schedule, is more able to fill conflict times with useful work than hardware schedulers, which typically can only suspend execution until the conflict is resolved. Furthermore, it is dramatically simpler and less expensive to build a highly parallel memory system when no centralized control unit is required to verify the memory reference pattern.

6.5. Instruction Fetch Considerations

Fetching and managing the execution of 1024-bit instructions in a pipelined machine posed some interesting challenges.

We implemented a physically distributed, full-width instruction cache. Bits of the instruction word are cached on the boards that they control; the processor's master sequencer (the "GC") contains the cache tag and control logic. The cache is built out of 35ns 64K static RAMs, and holds 8K instructions with a total bandwidth of 984 MB/second. In a fully configured machine, this is 1 megabyte of cache. It is virtually addressed and process tagged; flushing the cache is required only when we "run out" of hardware process tag values, not when we context switch.

Instruction virtual addresses are translated to physical addresses during cache refill through a dedicated instruction-stream TLB. This TLB has 4K entries, and is process tagged, with an "Address Space ID" (ASID) hashing scheme (like data TLBs) to improve multiple process hit rates.

Instruction fetch is fully overlapped with execution, and never stalls or restrains the processor, except on cache misses. We provided an extremely large cache to minimize the overall miss rate, and took extreme care to ensure that we could refill the cache at high speed on a miss.

6.5.1. The Instruction Encoding Format

Most programs contain sections which have lots of parallelism that our compacting compiler can find. In these parts of the code, many operations can be packed into each instruction. To maximize performance, for these parts of the program, we want a very wide instruction capable of independently expressing as many operations as possible. However, other "suburbs" program sections often have much less available parallelism, so that only a few operations will be inserted into each instruction word, and longer sequences of less filled instructions will be generated. If we have optimized the computer for the highly parallel sections, then in these suburbs we will have many functional units idle for many instruction cycles. This means that large portions of the instruction word will contain only no-ops, and will substantially increase the memory size of the program without contributing to its performance.

Machine designers have historically dealt with this dilemma by compromising: compressing their instruction encodings by preselecting those combinations of operations that they expect will be most commonly used. This then required the compiler to find and use the "patterns" that the designers had provided, if the highest performance was to be obtained. We believe these encoding schemes work out poorly in conjunction with a compiler; we pursued a quite different solution.

We place no restrictions in the instruction on what combinations of operations can be invoked simultaneously. Object code size is minimized in a different way: we use a variable-length main memory representation of the fixed-length machine instruction. That is, the instruction cache outputs a fixed-length 1024-bit instruction in each clock cycle; bits of the instruction word are directly wired to the functional units that they control. The architecture in this sense has a fixed-size instruction. However, we use a main memory instruction representation that eliminates the no-ops, affording a significant space savings.

Implementation of this variable-size memory instruction format had to satisfy a number of serious constraints. One constraint was that the instruction format not penalize execution of in-cache instructions. When the instruction cache is loaded, the control information for the functional units must be in the "right places" so that the instruction fetch pipeline length remains minimal.

A second constraint is that refilling the cache on a miss must proceed at the highest possible rate, without a huge amount of hardware dedicated solely for filling the cache. Since the TRACE system possesses massive main memory bandwidth through its use of an interleaved memory system and many buses, this means that it must be possible to control the cache refill without inspecting and interpreting each word as it comes from memory.

The instruction set must facilitate an easy-to-implement correspondence between the Program Counter, cache locations, and main memory locations, so that variable-length instructions can be "unpacked" quickly into a fixed-width cache.

Given that variable-length instructions are being fetched from a parallel, interleaved memory system, the "schedule" of what word will be on each bus in each cycle, and knowledge of which field of the instruction cache is to receive that value, must be produced by a control unit in real time as the data is returned from main memory. For a practical implementation, this requires that the schedule be precomputed by that control unit. The instruction representation must be such that this control unit is as simple and fast as possible.

We store instructions in main memory in blocks of four. Each block is preceded by four 32-bit "mask" words, which specify which 32-bit fields of the instruction are present in the block; the others are filled in the cache with zeros (no-ops).

The cache refill engine fetches and interprets the mask words. It never has to see or process actual instruction fields destined for the various functional units. This engine decides upon the schedule of buses to be used, initiates the instruction field fetches, and then tags the fields as they fly by on the ILoad buses so that they are steered to the proper functional units' cache words. The real-time overhead of this scheme is very low, since the actual cache refill proceeds at the maximum memory bandwidth and cpu bus bandwidths (the same buses are used in refill as are used for general computation).

This cache refill engine is perhaps the most complex piece of hardware in the TRACE, starting up enormous numbers of pipelined loads from memory and then directing them to the various instruction cache memories distributed throughout the machine. For sequential code, the mask interpretation is overlapped with the execution of the current block of instructions, so the operation of this cache refill engine represents a low overhead on the overall performance of the machine.

6.5.2. Branching

The architecture includes compare-predicate operations, rather than test operators and condition codes. We found it helpful to include compare instructions which could write the general registers, to allow evaluation of IF chains without branching. The architecture includes a special one-bit-wide 7-element register file, called the "branch bank", which can hold the result of compare (and other) operations, and which can be used to control branching. This allows the compiler to perform register allocation on branch bank elements, and move compare operations independently of the actual branches. A typical code sequence would look like:

```
CEQ R1, R2, BB(R3)     Write BB 3 with 1 if R1 == R2,
                       else write BB 3 with 0
BRANCH (R3)  LABEL     The "branch_test" field selects R3
```

The branch operation can be issued in the early beat of every instruction; part of the immediate field is used as the displacement for the branch. The branch is taken if the selected branch bank element is a 1. This branching structure resembles the "delayed branch" of other RISC machines, in that operations following the compare-and-branch are unconditionally executed while the branch target is fetched.

Conditional branching becomes an interesting problem as we attempt to fill wider instructions.[Fish83] Conditional branches occur every five to eight operations in typical programs; if we try to compact many more than five operations together, some mechanism will be required to pack more than one jump into a single instruction. We provided a multiway jump in the TRACE processor with multiple independent targets, with a software-controlled priority scheme.

Consider two jumps, with unique target addresses, which are initially sequential in the source program. If we want to pack them into a single execution cycle, we must establish a priority relationship between them, which defines which target address to branch to in the case that more than one of the simultaneous tests are true. The "highest priority" test whose condition is true provides the next address for execution. The priority relationship is driven by the original ordering of tests in the sequential program. The test that was originally first in the sequential program must be the highest priority; in the original sequential program, if the first test were true, then the second jump would never have been executed. Therefore, when we pack them together, we must arrange to ignore the results of the second lower-priority test if the first higher-priority test is true.

Each I unit can perform one test per instruction. A 32-bit "branch target adder" on each I board adds the current program counter to an immediate field of the instruction word. This computation yields a potential branch address. The branch arbitration mechanism determines which of four tests being performed simultaneously (on different I units) is the highest priority true test, and distributes the branch target associated with that test (defaulting to PC+1 when none of the four tests is successful).

Each I unit has nine bits of instruction word that control branching. Three bits ("branch_test") select an element from branch bank 0; another three select an element from branch bank 1. The values of the two selected bits are logically ORed to determine if "this board wants to branch". Three more bits (hidden in the immediate field) are defined by software to specify the relative priority of this test versus that of the tests being executed on the three other I units. These priority bits

show which other I boards it can preempt if its branch is true (which it does by sending "inhibit" signals to them directly). If no I board has a TRUE branch condition, then a central system controller board supplies a default PC value. Otherwise, the I board which has a TRUE branch condition and no inhibits is enabled to drive the new PC value onto the backplane.

This scheme is elegantly simple. It is fast, requiring only two gate delays (and one backplane traversal) to effect the arbitration. It is software-controlled, so that the compiler can adjust the relative priorities of the branches for each instruction. It allows the rapid selection of one of five potential next-addresses during every instruction; the fetching of the next instruction from that address is fully overlapped with the execution of the current instruction.

7. Exceptions and Optimizations

Some things you might take for granted, like traps and interrupts, have some subtle consequences when you try to rearrange execution order. We found we needed some unusual architectural features to enable more compiler code motion and optimization than a traditional approach to exception handling would have allowed.

Consider a FORTRAN loop that contains an array reference which is accessing across a row. If this loop is unrolled a number of times, and we allow the code generator to move the LOADs above the conditional branch that tests for the last iteration of the loop, several LOADs may be issued to addresses beyond the end of the program's current address space. The conditional jump will be all set to exit the loop, so that these references will be ignored (their data will never be used); but a conventional virtual memory system would terminate the program with a "Bus Error".

The architecture includes a special set of LOAD opcodes used by the compiler in the case when a LOAD moves above a conditional branch. When trap handling code sees one of these opcodes on a TLB miss, if no valid translation can be established for the reference, execution continues; the target register is loaded with a "funny number" to help catch bugs. These special opcodes are used only when necessary; we don't give up the helpful "Bus Error" traps when we don't have to, to assist in program fault isolation. This technique enables the compiler to be much more aggressive in code motions involving memory references.

A similar problem exists in floating-point exception detection and handling. Consider the fragment: `IF (A .NE. 0) C = D/A`. It's very much in the interests of performance to move divides up in the schedule; they take a long time. But if we want to detect division by zero, we must wait until the test has completed before initiating division.

Here again, we provided some assistance in the architecture. The processor has several floating exception modes, one of which is called "fast mode". In fast mode, floating exceptions cause traps only if the result is being written to the store file, being used in a compare, or being converted to integer form. Otherwise, a NaN ("Not-A-Number") or infinity will result from the offending computation, but no exception will be generated. As NaNs and infinities tend to propagate, any computation using the offending result will eventually cause a fault (by writing something to memory, for example). The trap will not occur at the most perspicuous point, but overall execution speed will be higher. (Note that floating underflows escape our notice in "fast mode", in that they are flushed to zero. We find this not to be a problem for many programs, and provide lower performance modes in which exceptions are detectable immediately.)

8. UNIX on the TRACE

A VLIW may appear to be an odd sort of CPU to make into a virtual memory timesharing system. Indeed, the original designers of the ELI-512 expected their machine to be useful only as a number-crunching back-end processor.[Fish83] The problems associated with making this heavily pipelined parallel machine capable of servicing interrupts seemed daunting enough, let alone all the rest: supporting virtual memory on a CPU without microcode, the incredible number of registers that would have to be context switched, extending the architecture and compiler to support systems code in addition to its numerical chores, not to mention that long instruction words might make all the utility programs consume gigabytes of disk space.

We've figured out ways around all of these problems, but it is natural to wonder why we built the TRACE to run 4.3BSD UNIX in the first place. The reason is simple: modern numerical applications programs do much more than perform floating point calculations. They make the usual demands of a system for disk, graphic, and terminal I/O, but they can make these demands at rates far exceeding those of "I/O intensive" systems programs. And scientific applications programmers have the same desires for reasonable and friendly programming environments that system programmers do. Fulfilling all these demands, particularly for performance, with a smoothly integrated front-end/back-end processor seemed difficult and unnecessary, so we built the operating system to run directly on the CPU.

8.1. Support for a Multiple Process Environment

We have already described many of the architectural features needed to support an operating system: the instruction and data TLBs needed for virtual memory; mechanisms and constraints for dealing with exceptions; and the desire for interruptability that led to our pipeline handling philosophy.

The TRACE supports its multiuser operating system in the usual way. Appropriate protection modes and privileged instructions are provided so that the user process environment is maintained. All accesses to mapping hardware, I/O stimulus instructions, and the PSW are carefully protected. A limited set of traps to system mode are provided for system calls and breakpoints.

We were concerned about the effects of running multiple processes, and the overall impact that context switching would have on performance. Our goal was to support about as many users as would be comfortable on a large supermini but to support order-of-magnitude larger computations than current superminis could support.

Context switching is often considered to be simply the cost of saving and restoring registers. But the actual cost of a context switch also includes the interrupt time, scheduling overhead, and any penalty for cache purging and cold-start. [Clar85] On many machines, the cost of purging the virtual address translation and instruction caches dominates register saving. The TRACE provides very large instruction and translation caches (see Sections 6.4 and 6.5), which are process tagged with an 8-bit "Address Space ID", or ASID. No purging of the instruction cache or translation buffers is necessary on a context switch; caches must be purged only every 255 address space mapping changes, when the set of ASIDs overflows.

Updating the ASID registers is cheap, so the high available memory bandwidth in the system permits a complete context switch in 15 microseconds. This figure holds in any machine configuration, because usable memory bandwidth increases as the number of registers. This performance is comparable to other machines that are trying to support our number of users.

8.2. Interrupts

Interrupt handling is almost entirely conventional. There is a priority interrupt system, with maskable interrupts from each device. When an enabled interrupt request arrives, execution suspends, the processor changes state, and execution resumes at a "trap" address. Since the pipelines are self-draining, after the maximum pipe depth time, all of the state of the processor is either in general registers or in main memory. A few hand-coded instructions begin saving registers while the pipelines drain; after several instruction times we enter C code to process the event.

8.3. Input/Output

Given an exposed architecture where the compiler knows about the machine resources being used throughout the system, it's difficult to allow I/O to "cycle steal" or otherwise share hardware resources on a fine-grained basis with program execution.

A memory-mapped I/O scheme would have required the CPU's memory interface to deal with devices with two distinct speeds: fast (to memory) and slow (to I/O devices). We chose not to implement our I/O this way. Instead, the CPU interacts with its devices through a surrogate called the I/O Processor (IOP). The IOP is based on an MC68010 with a multiported high bandwidth buffer memory and a "DMA engine" which can read and write blocks of main memory at half of peak memory bandwidth. The IOP interfaces to a VMEbus, a standard 32-bit asynchronous bus where the device controllers reside.

When the DMA engine wants to read or write main memory, it signals the GC. The GC suspends processor execution and allows pipelines to drain. The DMA engine then talks directly to memory at high speed; for example, 10 MB/s of I/O consumes only 4% of the machine's cycles in the largest CPU configuration. Execution resumes as soon as a burst of data has been transferred.

The I/O processor talks to the CPU using a bidirectional interrupt and a channel command protocol in main memory. Device drivers run on the I/O processor, a scheme which minimizes interrupts and CPU involvement in I/O operations. The IOP is also responsible for bringing the system up. A small operating system on the IOP supports execution of diagnostic and bootstrap programs.

We have devised a generic set of drivers on the TRACE side for each class of device on the IOP (disk, tape, ethernet, and terminal) which are very small, and which interface to device-specific drivers on the IOP. We have also implemented an I/O configuration system where all possible drivers are present (at a small cost in memory) and system device configuration is changed by editing a file on the diagnostic file system before booting UNIX.

8.4. Systems Code on a VLIW

The hundreds of thousands of lines of code which make up the UNIX kernel and utilities do not know they're running on a VLIW. One of our compilers is for the C language. Nearly all of the UNIX utilities, and a large chunk of the kernel, are written in portable C. (By actual count: 300 lines of assembly and 64K lines of C in the kernel; 1100 lines of assembly and 700 lines of C in the trap handlers.) The fact that our compiler performs exotic optimizations like inter-block compaction and transforms the code into a parallel form is irrelevant. We compile these programs and they do what they're supposed to do; *grep* doesn't know it's stretching the frontiers of technology, it just greps along at a terrific rate.

Trace Scheduling compacting compilation was originally conceived for numerical applications; we expected to run into problems handling systems code. The systems code in UNIX differs in several respects from numerical code. Systems code makes pervasive use of pointers, which leads to more difficult compiler optimization problems. The code tends to have even smaller basic blocks than numerical code. And most important, systems code has proportionately many more procedure calls than numerical code.

Pointers and small basic blocks have not been a problem. In fact, procedure call overhead seems to be the only issue that has required special attention. Performance on systems code is quite good (the C and Fortran compilers share a common back end).

The TRACE provides no special architectural support for procedure calls (other than the large memory bandwidth already built in). During the design, we considered several hardware mechanisms intended to minimize procedure call/return overhead, but none of them was both a clear performance win and clearly feasible. We decided to rely on the compiler to be clever with its use of registers and procedure inlining, and to develop a global register allocating linker, which builds a global call graph and minimizes register saves (currently in the works).[Wall86] We expect this work to be complete by the time of the conference presentation, and will report on it there.

When we initially debugged UNIX on the TRACE, we restricted traces to basic blocks, and disabled loop unrolling; compiler heuristics for how much unrolling to perform had not yet been installed, and code grew unmanageably. Those heuristics are now in place, and their performance is remarkably good. The full compacting compiler optimizations work well for a wide variety of systems code, including the kernel itself, without undue code growth.

This result surprised us somewhat; we hadn't anticipated as much improvement on systems code as we got. Good performance on systems code is very desirable, as it restrains the proportionate growth of operating system overhead that is usually encountered on a parallel machine. Unlike "coarse-grained" architectures where systems code runs on a single scalar unit (and can become a substantial bottleneck), we retain the same OS-to-user balance found on more traditional systems.

9. Code Size: Initial Results

The"no-op" fields of an instruction are not represented in main memory, so the object code size of a program is directly proportional to the number of operations in the compiled program. There are thus three components to consider when comparing VLIW code density to that of other architectures:

- the number of bits required within the instruction set to express a given operation;

- the succinctness, or lack thereof, with which common high-level operations (like procedure call) can be expressed in the instruction set; and

- the number of new operations introduced through compiler optimizations and loop unrolling.

The VLIW encoding of each operation is roughly on par with other RISC machines. It is a three address architecture, all loads and stores are explicit, and there is minimal instruction encoding. The code expansion per operation is probably around 30 – 50% when compared to a tightly encoded machine like the VAX or Motorola 68000. The variable-length main memory instruction encoding has an associated overhead of a few bits per operation, which coupled with main memory alignment constraints adds roughly an additional 5 – 10%.

Operations that cannot be initiated in a single instruction cycle are broken down into constituent sub-operations. These constituents are usually substituted inline, although certain operations such as the block register save and restore associated with procedure call are implemented via special subroutines. The overall code expansion due to this, as compared to a machine like the VAX that has an extensive library of microcoded "subroutines", is difficult to quantify, but is probably in the neighborhood of 10 – 20%.

The compiler performs an enormous number of optimizations, most of which reduce the number of operations in the program, but some of which increase the number of operations with the goal of increasing parallel execution. The three most notorious code-expanders are inter-block trace selection (which can produce compensation code), loop unrolling, and inline procedure substitution. All three of these are currently automatic and have been tuned to avoid undue code growth. These optimizations can increase the size of some small fragments of code by a large factor, but their overall effect seems to be to increase code size by a factor of around 30 – 60%, although the user can increase or decrease these factors arbitrarily through the use of compiler switches.

Several large (100K – 300K lines) FORTRAN programs have been built on the TRACE. After unrolling and trace selection, the code size is approximately 3 times larger than VAX object code (compiled with the VAX/VMS FORTRAN compiler).

The concern about code size led us to implement a shared-libraries facility very early in our UNIX development. This has substantially reduced the size of the UNIX utilities images. The UNIX utilities consume approximately 20MB of disk space on a VAX, and approximately 60MB on our VLIW using shared libraries.

UNIX has been running on the TRACE and supporting its own development for some time. The principal advantage of Multiflow's parallel processing technology is that it is transparent to its clients. Thus, most of the challenging problems in developing an operating system and programming environment for the TRACE come not from its VLIW nature but from our intention to make the system into a first rate environment for high performance engineering and scientific computation. A thorough discussion of our approach is beyond the scope of this paper.

10. Summary and Future Work

This paper has introduced the Multiflow TRACE Very-Long-Instruction-Word architecture. Before this machine was built, some designers and researchers predicted that the negative side-effects of the VLIW/compacting compiler approach (object code size, compensation code, context swap time, and procedure call/return overhead) would likely swamp the machine's performance gains. These predictions were wrong: we slew some of these dragons with cleverness, tamed a few, and couldn't even find the rest.

It is too early to be able to separate out all the different contributions to performance in the TRACE. Our future work will concentrate on quantifying the speedups due to trace scheduling vs. those achieved by more universal compiler optimizations. We will also be examining the efficacy of memory-bank disambiguation, speed/size tradeoffs of the fixed and variable instruction encoding schemes, and instruction cache usage statistics.

Our conclusion should be unsurprising: given an implementation technology, the best way to use it is to build a VLIW. If you build a standard scalar machine instead, you pass up significantly higher performance at only slightly higher cost; the extra functional units are cheap compared to the overhead of building the computer in the first place (memory, control, etc.). If you build a vector machine instead, the parallel hardware you build "turns on" only occasionally, and the speed of some vector code is all that will be improved. And if you build a multiprocessor instead, you pay the full overhead of instruction execution and run-time synchronization per functional unit, without getting the fine-grained speedups a VLIW can offer.

11. Acknowledgements

Thanks go to Chris Genly and Ben Cutler for help with the diagrams in this paper, and to Helen Spontak for easing scheduling constraints.

References

Kate85.
Manolis Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, MIT Press, Cambridge, Mass., 1985.

Tjad70.
G.S. Tjaden and M.J. Flynn, "Detection and parallel execution of independent instructions," *Transactions on Computers*, vol. C-19, no. 10, pp. 889-895, IEEE, October 1970.

Fost72.
C.C. Foster and E.M. Riseman, "Percolation of code to enhance parallel dispatching and execution," *Transactions on Computers*, vol. C-21, no. 12, pp. 1411-1415, IEEE, December 1972.

Fish83.
Joseph A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," *Proceedings of the 10th Symposium on Computer Architectures*, pp. 140-150, IEEE, June, 1983.

Elli86.John R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, Mass., 1986.

Fish79.
Joseph A. Fisher, "The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of

Processor Scheduling with Resources," *Technical Report COO-3077-161*, Courant Mathematics and Computing Laboratory, New York University, October 1979.

Henn81.
John L. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "MIPS: A VLSI processor architecture," *Proceedings of the CMU Conference on VLSI Systems and Computations*, pp. 337-346, Computer Science Press, October 1981.

Radi82.
George Radin, "The 801 Minicomputer," *Proceedings SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 39-47, ACM, March 1982.

Toma82.
Robert M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *Computer Structures: Principles and Examples*, pp. 293-305, McGraw-Hill, 1982.

Thor70.
James E. Thornton, *Design of a Computer: The Control Data 6600*, Scott, Foresman & Company, Glenview, Illinois, 1970.

Acos86.
R.D. Acosta, J. Kjelstrup, and H.C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Transactions on Computers*, vol. C-35, no. 9, pp. 815-828, September, 1986.

Seit85.
Charles Seitz, "The cosmic cube," *Communications of the ACM*, vol. 28, no. 1, pp. 22-33, ACM, January 1985.

Walt87.
David L. Waltz, "Applications of the Connection Machine," *Computer*, vol. 20, no. 1, pp. 85-97, IEEE, January 1987.

Fish81.
Joseph A. Fisher, "Trace Scheduling: A technique for global microcode compaction," *Transactions on Computers*, vol. C-30, pp. 478-490, IEEE, July, 1981.

Fish84.
Joseph A. Fisher and John J. O'Donnell, "VLIW Machines: Multiprocessors We Can Actually Program," *CompCon 84 Proceedings*, pp. 299-305, IEEE, 1984.

Elli84.John R. Ellis, Joseph A. Fisher, John C. Ruttenberg, and Alexandru Nicolau, "Parallel Processing: A Smart Compiler and a Dumb Machine," *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, ACM SIGPLAN Notices, June 1984.

Pfis85.Gregory F. Pfister and V. Alan Norton, "Hot-Spot Contention and Combining in Multistage Interconnection Networks," *Transactions on Computers*, vol. C-34, pp. 943-948, IEEE, October 1985.

Smit82.
Alan Jay Smith, "Cache Memories," *ACM Computing Surveys*, ACM, September 1982.

Clar85.
Douglas W. Clark and Joel S. Emer, "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 31-62, February 1985.

Wall86.
David W. Wall, "Global Register Allocation at Link Time," *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, ACM SIGPLAN Notices, July 1986.