A PROCESSOR ARCHITECTURE FOR HORIZON

Mark R. Thistle
Institute for Defense Analyses
Supercomputing Research Center
Lanham, Maryland 20706
thistle@super.org

Burton J. Smith †
Tera Computer Company
P. O. Box 25418
Washington, DC 20007-8418
burton@lll-crg.llul.gov

ABSTRACT

Horizon is a scalable shared-memory Multiple Instruction stream - Multiple Data stream (MIMD) computer architecture independently under study at the Supercomputing Research Center (SRC) and Tera Computer Company. It is composed of a few hundred identical scalar processors and a comparable number of memories, sparsely embedded in a three-dimensional nearest-neighbor network. Each processor has a horizontal instruction set that can issue up to three floating point operations per cycle without resorting to vector operations. Processors will each be capable of performing several hundred Million Floating Point Operations Per Second (FLOPS) in order to achieve an overall system performance target of 100 Billion (10^{11}) FLOPS.

This paper describes the architecture of the processor in the Horizon system. In the fashion of the Denelcor HEP, the processor maintains a variable number of Single Instruction stream - Single Data stream (SISD) processes, which are called instruction streams. Memory latency introduced by the large shared memory is hidden by switching context (instruction stream) each machine cycle. The processor functional units are pipelined to achieve high computational throughput rates; however, pipeline dependences are hidden from user code. Hardware mechanisms manage the resources to guarantee anonymity and independence of instruction streams.

1. Introduction

Over the last decade, supercomputer performance has risen by an order of magnitude. Machine cycle times of under 10 nanoseconds have become commonplace. However, this performances comes with a cost. Maximum performance can only be achieved with heavily vectorized codes. In addition, these codes must be tuned precisely to the machine on which they will run in order to achieve this maximum performance, thereby thwarting any effort to generate portable programs. Although non-vectorizable problems may benefit from the scalar performance of supercomputers over that of the more general purpose minisupercomputers, they still do not utilize the full potential of the machine. Consequently, scalar speed is low. A fundamental limit for scalar performance is memory latency for large memories. The problem is exacerbated in multiple processor systems by the physical and logical nonlocality of memory with respect to a processor. Cacheing techniques have been developed to reduce effective memory latency; however, in shared memory systems, cache coherence is a significant problem [1].

The Horizon supercomputer is a shared memory, Multiple Instruction stream, Multiple Data stream (MIMD) system that bridges the gap between scalar and vector performance. Each processor in the multiple processor system uses internal multistream parallelism to compensate for the latencies introduced by the large shared memory and the pipeline. The architecture of the processor in Horizon bears a strong resemblance to that of its predecessor: the Denelcor HEP. At the time of this writing, the SRC is completing a one-year study to demonstrate the feasibility of the Horizon architecture, its implementation, and its software approach. This paper describes the architecture of the Horizon processor and highlights many of the features which support the high performance targeted for the machine. Section 2 introduces the programmer's model of the Horizon system and of the individual processors. The details of the architecture are discussed in Section 3, with some of the major unresolved issues in Section 4.

2. Programmer Model

2.1 Global Machine Model

A Horizon machine consists of P processors, where P is currently defined in the range 256 to 1024, that share a memory with an address space of 2^{48} bits. Memory is organized into 64-bits words and is addressed through 48-bit virtual bit addresses, the most significant 42 bits of which form the word address. Loads and stores fetch or store the 64-bit word that contains the bit that was addressed. Associated with each memory word is a six-bit access state composed of a full/empty bit, an indirect bit, and four trap bits numbered 0 to 3. These access state bits can modify the behavior of memory references to the associated location. All communication among processes in the machine is through the shared memory [2]. describes the options available for memory access.

Processors and memory modules are spatially and logically distributed throughout a multidimensional, nearest-neighbor, packet-switched interconnection network. Each processor is individually connected to one node in the network and sends data to or receives data from a memory module by via this node. The current design calls for a ratio of the number of processors to memory modules from 1:1 to 1:2. The response time of the memory request is a function of the addressed module's distance from the processor and the traffic in the network. Simulation indicates that average response times of 50-80 cycles are attainable in a system containing 256 processors and 512 memory modules, with nearly all responses available within 128 cycles. Each machine cycle, the network is capable of receiving a message from every processor and memory module, and delivering a message to every processor and memory module. [3] describes the interconnection network for Horizon.

2.2 Processor Model

The Horizon processor manages a variable number of processes, as in the HEP architecture [4], called instruction streams (i-streams), each of which is an autonomous virtual processor with its own register set, program counter, and associated processor state. Each cycle, the processor issues the next instruction from an i-stream chosen from the set of active i-streams. Instructions from any single i-stream are executed in sequential order. Pipelining of instruction

† Work performed at the Supercomputing Research Center.

execution is hidden from the user to avoid the complexity of managing exposed pipelines. The maximum number of i-streams per processor is 128. There is no sharing of registers or other processor state between i-streams; the processor manages each i-stream's state independently.

The processor has three major execution units: the Memory unit (M-unit), Arithmetic unit (A-unit), and the Control unit (C-unit). The architecture is horizontal; a 64-bit instruction can initiate three operations, an M-unit operation (M-op), an A-unit operation (A-op), and a C-unit operation (C-op), through three independent fields in the instruction. Also part of each instruction is a lookahead field that is used to control instruction overlap. The user-specified lookahead value in an instruction indicates the number of subsequent instructions that may be issued without waiting for the completion of that instruction.

M-ops include loads and stores. A load/store architecture was selected to maximize performance with high-bandwidth register operations. Shorter instructions, less hardware complexity, and better reuse of values are additional benefits of a load/store architecture. Arithmetic, logical, control, test, and miscellaneous operations are performed in the A-unit, C-unit, or both. [5] specifies the complete Horizon Instruction Set.

Synchronization between instruction streams is performed through memory operations that use the full/empty bit in each memory cell. An instruction stream can suspend its execution or can be placed into execution by a single instruction. The allocation of instruction streams to processors is handled jointly by user code and the operating system. The operating system allocates instruction streams in response to a system call. Such a system call requires a significant amount of operating system code to be executed and is therefore used relatively infrequently. Thereafter, the user program activates the allocated instruction streams as they are needed. The activation of an i-stream is accomplished by a single instruction.

Each of the M-, A-, and C- fields consists of an operation code (opcode) and the register operands to be used. An instruction may specify up to 7 register sources and 3 register destinations, or 8 sources and 2 destinations for the execution unit operations. These registers are referred to symbolically as q, r, s, (M unit operation), t, u, v, w, (A unit operation), x, y, z, (C unit operation). Register q is either a source or a destination register for the M-unit operation, t is the destination register for the A-unit operation, and x is the destination register for the C-unit operation. The other registers are sources. The number of register accesses for each instruction is an important parameter in the design and will be discussed later. The several dozen program fragments that have been written for Horizon confirm that the horizontal aspect of the instruction set is well balanced and not overly aggressive. From 1-1/2 to 3 floating point operations per instruction have been achieved for the various test programs. A compiler for a small FORTRAN-like language was written to provide data on the suitability of the Horizon instruction set for code generation; this compiler has been successful in achieving high utilization of all three operation fields [6].

The data formats supported include integer, floating point, bit vector, target, and pointer. Integers are signed, 64-bit two's complement values. Floating point numbers are also 64-bit quantities with a non-standard format, chosen for its dynamic range, compatibility with the integer format for common test and compare operations, and recoverability of results from operations that produce overflowed or underflowed results. The definition of the floating point format and the rules for floating point addition, multiplication, and other operations are given in [7]. Pointers are 64-bit values containing a base address and several bits that control the synchronization, indirect addressing, and trapping properties of memory references.

For each of the 128 possible i-streams in the processor, there are 32 64-bit general-purpose registers. The general-purpose registers can each hold integers, floating-point numbers, portions of bit vectors, targets, or pointers. There are four 64-bit target registers and one Instruction-stream Status Word (ISW). Target registers are used to specify potential branch target addresses. The ISW specifies the current Program Counter (PC) as well as the trap mask and condition vector. There is also a 1024-word processor constant table, a read-only table for commonly-used constants such as pi, e, and bit masks, that is shared among all i-streams in the processor.

3. Processor Architecture

3.1 Introduction

Each Horizon processor has a pipelined architecture capable of sustaining a performance of 400 MFLOPS. The target machine cycle time is 4 nanoseconds. To achieve the targeted performance, Horizon embodies four levels of parallelism. At the top level is the MIMD model with several hundred processors. The next three levels are supported within the processor and include: pipelined MIMD instruction execution, that is concurrent execution of multiple instruction streams in a round-robin fashion; overlapped and pipelined execution of instructions within each stream; and horizontal instructions each performing multiple functions. Compilers have traditionally assumed the burden of managing the last two levels of parallelism. The targeted high performance is, in general, accomplished by effectively managing the parallelism, i.e. rapidly switching context in order to hide the memory latencies introduced by a large shared memory, including distance as well as contention and synchronization delays.

The remainder of this paper describes the architecture of the Horizon processor and examines how the various levels of parallelism are implemented. An overview of the functional block level design will be presented, followed by detailed examinations of the important components of the processor architecture, namely, the register organization, instruction issue sequence, i-stream selection policy and mechanism, and memory interface.

3.2 Functional Unit Model

Each processor has seven main functional components: the three execution units - M, A, and C units; the Instruction Fetch and Issue unit, the I-stream Selection unit, the Instruction Cache and Prefetch unit, and the register set. Figure 1 shows the internal organization of a Horizon processor. The instruction execution logic is completely pipelined so that it can begin executing a new instruction every cycle. Each of the function units is pipelined and requires multiple cycles to complete each instruction. Pipeline lengths will be fixed and will appear identical for all instructions, regardless of their complexity. Fixed-length pipelines allow a simple scheme to be used for register scheduling. A variable pipeline length that depends on the complexity of the instruction being performed is also possible. This scheme adds complexity to i-stream scheduling and creates register bandwidth problems for simultaneous completion of instructions but reduces the number of i-streams needed to keep the machine busy. Implications of these alternatives will be discussed later.

Every cycle, the i-stream selection logic selects an i-stream from which to issue the next instruction. The processor hardware and the compiler together must guarantee that an i-stream is eligible for selection only when all the flow and output dependences on previously issued instructions have been satisfied (there are no control or antidependence problems with previously issued instructions), and the instruction has been prefetched into the instruction buffer. To simplify the instruction issue decision, it is convenient to arrange

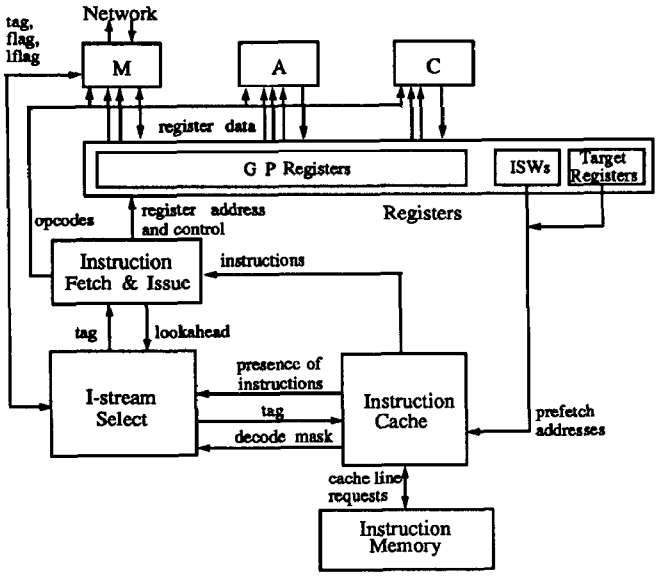

Figure 1. Horizon Processor Functional Architecture

pipeline lengths so that the earliest possible memory response (to the M-unit) is guaranteed to be later than responses from the other function units (A- and C-units). The completion of an instruction is thus made equivalent to the receipt of a response from memory. If no memory operation is specified in the instruction, instruction completion is signaled after the A- and C-units have completed.

After an i-stream has been selected for issue, its next instruction is issued to the execution units. Register operands are delivered to the execution units and execution begins. After a fixed delay, results from the A- and C- units are written back to the registers. Memory responses are essentially asynchronous events. Writing these results back to the registers will be discussed in the next section.

Every cycle the machine context is switched to another i-stream. This technique hides latencies due to memory and network response times and full/empty synchronization delays. As long as there are enough i-streams in the system to fill the pipeline in each processor, the system utilization will approach unity. The result is that the pipeline becomes totally invisible to the programmer. Instructions in a single i-stream seem to complete prior to the issue of the next instruction. The programmer need not be concerned with the number of pipeline segments in each function unit. Thus, pipelined MIMD instruction execution uses parallelism to compensate for lack of locality, and improves the programmability of horizontal (as well as conventional) processors by concealing the pipeline.

With 32 general-purpose registers, 4 target registers, 4 condition codes, the i-stream status word, and other miscellaneous state, the amount of state associated with each i-stream is large. Rather than saving and restoring the state at each instruction cycle, the processor switches context to the appropriate i-stream state. This context switching technique, taken directly from the Denelcor HEP, first appeared in the Control Data 6600 Peripheral Processor unit as the "barrel" and also in the Texas Instruments ASC I/O processor. One set of control logic is time-multiplexed among multiple i-streams. Each i-stream is served at a rate commensurate with its memory latency (or some other limiting function) while the control circuitry and arithmetic pipelines operate at a much higher rate. The maximum number of i-streams per processor 128. Technology considerations prevent it from being more than twice as large as this and 64 is too few to ensure that enough i-streams are ready to execute, given the expected network latency.

Because of the large number of i-streams in a processor, memory traffic generated by instruction fetch is considerable. Therefore access to instruction memory is made separate from data memory which is accessed by the processor through the interconnection network. Distinct processor ports for data and instruction memory operations are provided. An additional benefit of a separate instruction memory, aside from the increased bandwidth available, is that instruction security (ability to execute but not read or write instructions) may be enhanced. Depending on the speed of instruction access, an instruction cache may or may not be needed. Instruction cache issues are addressed in Section 4.

3.3 Register Organization

Each processor has one set of 32 general-purpose registers allocated to each of its i-streams. Because the number of registers per i-stream multiplied by the number of i-streams is large, it is impractical to implement the registers with individual latches or flip-flops. Instead, memory circuits must be used, thereby limiting the cycle time of the processor to some integral multiple of the register memory cycle time. This multiple depends on the number of register accesses in an instruction. Although it is possible to exchange space for time to some extent by implementing several copies of the registers behind the scenes, this approach is costly for horizontal instructions with many register references in each instruction.

The processor can achieve one instruction issue per register memory cycle while allowing a large number of register accesses per instruction by partitioning the register memory into banks. All of the registers for a subset of the processor's i-streams are resident in a single bank. In this scheme, an instruction reads and writes its registers one after another, using several cycles of the register memory bank. Other instructions that were started in the immediate past are meanwhile reading and writing their own registers which are located in other register memory banks. An i-stream becomes a candidate for execution of its next instruction based not only on whether prior instructions have finished but also on whether its register bank is available.

This register banking idea permits the implementation of a single register address space for each instruction rather than the fragmented register address spaces found in conventional horizontal processor designs. In conjunction with pipelined MIMD instruction execution register banking thus solves a problem traditionally associated with code generation for horizontal processors, namely the problem of deciding which registers should hold which values. This problem, together with the difficulty of managing an exposed pipeline with mixed function unit latencies, is largely responsible for much of the bad reputation horizontal machines have as targets for compilers.

The number of registers is a compromise stemming from the number of functional units, the number of source and destination registers needed by each function unit, the length of the instruction, and the register read/write bandwidth available. Each instruction contains up to ten references to registers requiring 50 bits of the 64-bit instruction to specify register names. The remaining 14 bits are used for the M-, A-, and C-unit opcodes and for the lookahead specification. Fewer registers, 16 for example, is not well-balanced with the number used in each instruction and would have led to under-utilized function units or worse -- register spills to memory. More registers, 64 for example, would have required too many bits (assuming fixed size instructions). Given a larger instruction width, 64 registers might be desirable.

Because every Horizon instruction requires either eight register reads and two register writes or seven register reads and three register writes, the sequential register read/write schedule would cause a long instruction cycle. To ease this problem, an identical copy of each register bank is maintained. This allows simultaneous reads to each copy, resulting in two register reads per cycle. A register write places identical data in both bank copies in one cycle. Trading space for time in this way reduces the requirement from eight to four cycles of register reads per instruction. Up to three cycles of register writes for each instruction are required.

With at most seven register cycles used by an instruction, at most seven register banks will be busy every cycle. There should be enough banks compared to the number of register references in an instruction so that there are always i-streams available to execute. If there were on the average only one or two free register banks at any cycle, the number of i-streams needed to keep the processor busy would have to be fairly large compared to the number of banks. To ensure there will always be enough free banks with ready i-streams, the number of register banks currently being considered is 16. Each bank contains the general-purpose registers for 8 i-streams.

This register organization requires a connection-intensive multiplexing scheme to route register accesses from the execution units to the correct register banks. A good implementation of this scheme is critical in order to achieve reasonably short register access latency.

3.4 Instruction Issue

The instruction issue sequence in the Horizon processor requires register bank schedule management similar to the pipeline reservation tables described by [8]. Future register bank cycles are reserved by issuing instructions for reading and writing operands, thereby affecting issue eligibility of other i-streams with registers in the same register bank. This section describes how this scheduling is performed.

When an instruction issues, four consecutive bank access cycles will be consumed reading up to eight source registers. During this time, no other instruction may issue from any i-stream whose registers are resident in that same bank. As the execution units receive their operands, their processing begins. There will be a fixed pipeline delay due to the processing time of the A- and C-units. As the results from each of the A- and C-units emerge from the pipe, they are written to the destination registers. Every cycle one instruction will issue from an i-stream in a different register bank . Instruction issue from a bank is allowed if the next four consecutive bank cycles are free (not previously reserved for register reads or writes). The pipeline schedule for a single instruction issue and execution sequence is illustrated in Figure 2.

The successful completion of a memory reference, unlike operations in the A- and C-units, is an asynchronous event. Memory references must traverse the interconnection network to reach the target address and then return to the source processor. In addition, some memory references may not be satisfied by the first access. For example, in a "wait-for-full-and-set-empty" load, unsuccessful attempts to read a full memory location require that the operation to be retried. Consequently, the register write resulting from a memory load cannot be scheduled at instruction issue time as are those for the results of the A- and C-units.

As memory references are completed, the information about that memory reference (i-stream number, destination register, 64-bit datum, etc.) is enqueued in the M-unit. The M-unit must then steal a cycle from the appropriate register bank to write the datum to the destination register. A bank is considered free for an M-unit result register write if there is no activity scheduled for that cycle. Free bank cycles arise in one of two ways: (1) Future reserved cycles could have prohibited instruction issue from the bank leaving the next several cycles free. If the bank was eligible for instruction issue but was not selected, there would be at least one free cycle for a register write; (2) An issued instruction might not require all its register read and write slots, thereby leaving free bank cycles.

Experience with the HEP [4] and initial simulation results indicate that a sufficient number of register write cycles are available to avoid starvation. However, to utilize all unused register slots, some decoding of the instruction must be done early. The register read and write cycles that are not needed must be known early enough to steal the free cycles for register writes. Timing is critical only for the first few register read cycles, since more than enough time is available to determine the need for the A- and C-unit write cycles.

3.5 Instruction Stream Selection

The i-stream selection mechanism selects and issues an instruction every clock cycle, as long as there are i-streams ready to issue. This context switching mechanism is performed every cycle and is fundamental for the processor to achieve high performance in a shared memory environment. An i-stream is ready to issue if there are no dependences on previously-issued, unfinished instructions. These dependences are indicated in the lookahead field of the assembly language instruction. The other factors which determine an i-stream's issue eligibility are (1) whether the i-stream's register bank is free for an instruction issue (described in the previous section), and (2) whether the next instruction is available. Every clock cycle, the selection mechanism chooses from among the instruction streams that are candidates for issue. A "fair" selection algorithm will help to ensure that issue starvation does not occur. Other selection mechanisms are being considered that impose issue priorities on streams; however, as yet, none have been formally defined.

The mechanism that manages a stream's dependence information is the lookahead logic. The lookahead value associated with an instruction indicates the number of subsequent instructions that may be issued before it must complete. Instruction completion is defined to be the completion of the memory operation; an instruction with no memory operation completes after the fixed delay of the pipeline. The lookahead logic allows instructions to overlap. Consider an instruction (i) containing a memory reference and a lookahead value of (L). Before instruction (i)'s memory reference is complete, instructions (i+1) through (i+L) may issue. The lookahead field in an instruction is 3 bits wide; therefore, the maximum lookahead value is 7. Hence an i-stream may have a maximum of 8 instructions simultaneously executing.

The maximum lookahead value is related to the instruction word size, the number of destination register references in an instruction and to the size of an i-stream's register set. With 32 registers and 3 destination registers per instruction, 24 registers would be reserved for 8 outstanding memory references (maximum lookahead value of 7). Since a nominal number of registers, say 8, will be live and unusable, a maximum lookahead of 7 seems reasonable. A maximum lookahead of 15, however, would allow all registers to be reserved for memory reference returns. This change would have a

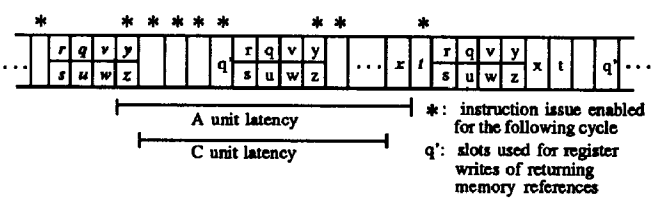

Figure 2. Instruction Execution Sequence

positive effect of reducing the number of i-streams required to keep the processor busy, since a greater memory latency could be tolerated by each i-stream. However, it has the negative effect of doubling the amount of state that must be kept by the M-unit for outstanding memory references, as well as increasing the amount of logic required to manage the additional lookahead and increasing the instruction width. Perhaps more importantly, a greater maximum lookahead also tends to increase the statically scheduled parallelism needed within each instruction stream. Experience with simulation of application codes has not yet provided insight into the realizable benefit from an extended lookahead.

The determination of whether an i-stream is a candidate for issue is performed by the lookahead logic. A functional block diagram of the lookahead mechanism and ready logic is shown in Figure 3. There are 8 locks per i-stream, each corresponding to one of 8 possible instructions awaiting completion. Since there are 128 i-streams per processor, there are a total of 1024 locks in the processor. This defines the maximum number of outstanding memory references allowed from the processor. The value of lock i indicates for how many previous instructions instruction i is waiting. An instruction may not issue until its lock is zero. The current instruction flag counter, flag, points to the lock of the instruction to issue next.

When an instruction issues, the lock associated with the future dependent instruction is incremented. The lock number, lflag, is stored in the M-unit along with the rest of the state describing the memory reference (M-unit) part of that instruction. Lflag indicates which lock is to be decremented when the memory reference completes. Each time an instruction is issued from an i-stream, flag is incremented mod 8. The i-stream is "ready" to issue if the lock addressed by flag is zero.

The inputs to the i-stream selection mechanism are the bank-free signals, the i-stream ready signals described above, and the instruction-present signals. With its current definition, the selection mechanism can be implemented entirely in combinatorial logic, as a two-level, inverted binary tree. On one side, a register bank from among the 16 possible banks is selected to issue an instruction. A register bank is "ready", i.e., is a candidate for this selection, only if there is at least one i-stream in the bank that is ready to issue an instruction and the bank is free. The bank-free bits for each register bank are generated in parallel from the bank schedules. Similarly, the ready bits for all the i-streams are produced in parallel by the lookahead logic. Bank readiness is determined for all free banks in parallel by OR-ing the i-stream ready bits in each bank. Bank selection is based on a "roving" round-robin algorithm, in which the highest priority rotates among the banks. The round robin selector determines the bank with the highest priority for issue each cycle. With this scheme, the maximum possible delay between instruction issues from a bank with ready i-streams is 16 cycles.

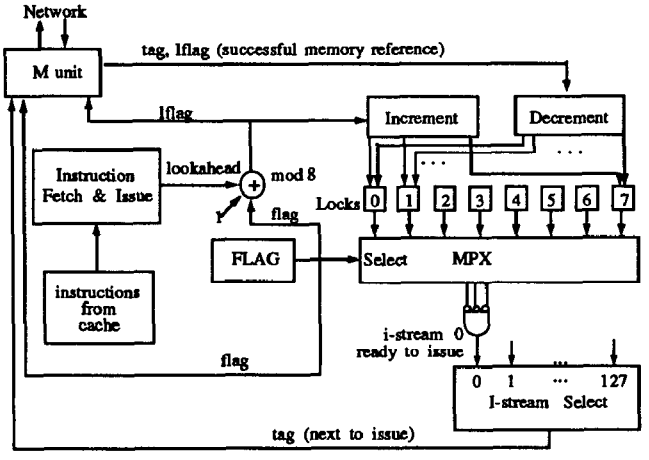

Figure 3. I-stream Lookahead Logic

The other side of the tree is the selection of an i-stream from a bank. The same round-robin mechanism as in bank selection is used. The choice of which i-stream will be issued next from each bank is generated in parallel for each bank. The worst-case delay between issues from a particular i-stream (assuming it is ready) is 8 bank issue cycles; that is, 8 cycles in which some i-stream is selected from the particular i-stream's bank. The two sides of the tree are joined by multiplexing down the tag of the chosen i-stream in the chosen bank. This selection algorithm guarantees uniform treatment of banks and of i-streams within a bank. Given the register banking constraints, i-streams in lightly loaded banks will be favored for execution over those in heavily loaded banks. This natural priority may be useful for run time scheduling of multiple priority tasks within a processor.

3.6 Memory Interface

The M-unit directs all memory references from the processor into and out of the interconnection network. Memory requests are generated from instructions, processed after returning successfully from the network, reissued if unsuccessful, and rerouted if destined for another node. (Instruction cache lines are not fetched through the M-unit, since instructions do not travel through the network.) When a load or store instruction is issued, the M-unit receives the tag and flag (Figure 1) of the instruction from the I-stream Selection unit. This information is saved in the M-unit state table, along with the information about the request from the instruction.

The M-unit state table holds the following information: the operation to be performed (opcode), the virtual memory address, the tag and flag of instruction, destination register number, the 64-bit datum, a timestamp, and a retry value. The flag of the instruction is the address of the lock to decrement when the instruction completes. Because the arrival of a successful response from memory is an asynchronous event, some of the responses must be retained until the datum can be written to the registers during a free register bank cycle. Figure 4 shows a functional block diagram of the M-unit components. The 16 queues, one per register bank, are used to buffer completed memory requests until a register write can be performed. When the register write occurs, the datum is written to the destination register of the appropriate i-stream register set (in the case of a load). For either a load or a store, the lock addressed by lflag is decremented, indicating completion of the instruction. Assuming

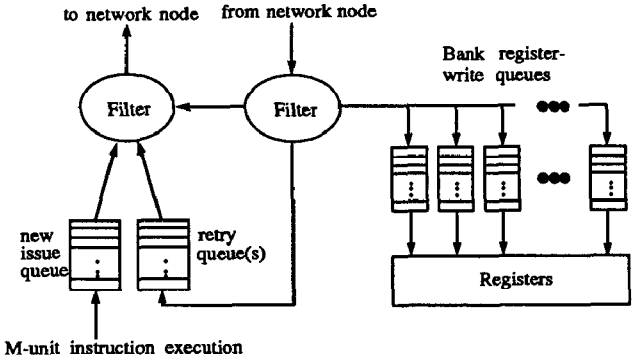

Figure 4. Memory Unit Network Interface

that each M-unit result register write queue is provided a unique data path to the registers and the i-stream selection logic can support parallel access to the locks, up to 16 M-unit result register writes could be performed in any cycle.

If the instruction contains a synchronizing load or store (e.g. wait-for-full), the M-unit must schedule a retry in the event of an unsuccessful request. Simply re-trying every unsuccessful request immediately might quickly flood the interconnection network with requests that would likely be unsuccessful again. (Flow control would limit the extent to which flooding will occur [3].) In addition retries would cause interference with issue of newly-generated requests from issued instructions since there is only a single port to the network. Hence, some back-off policy will be adopted for retry scheduling. Each successive unsuccessful attempt at a memory access will result in a greater delay before another re-try is issued. The back-off policy is further complicated by the possibility that a message may be returned without ever having gained access to its destination memory cell. Such is the case at "hot-spots" where bank conflicts overflow queues at the memory banks. Randomization policies reduce the probability of memory bank conflicts; however, hot spots will inevitably occur, whether because of poor programming practices, algorithmic constraints, or "bad" statistical epochs. Consequently, several classes of retry messages may be required.

The last responsibility of the M-unit is re-routing messages that are not addressed to it. A message may be delivered to a processor it is not intended for if there are no other links available for that message at the node [3]. Such a message is called a misrouted message. The M-unit must then turn the message around and re-issue it to the network. The filters in Figure 4 perform this function.

As indicated in Figure 4, there are essentially two queues from which messages are inserted into the network. One queue contains messages generated by newly issued instructions and the other contains messages to be retried. An alternating scheduling mechanism of memory references delivered into the network prevents starvation of either queue. Misrouted messages must have priority over either queue for injection into the network, since the processor has no knowledge about nor any reserved storage for such messages. The size of the issue queue is 1024, the maximum number of outstanding memory references allowed from the processor. To avoid deadlock, the retry queue must also be length 1024.

4. Unresolved Issues

4.1 Introduction

As the processor architecture evolves, many new (not yet resolved) issues are identified. Some of these issues are implementation details which become more important in later stages of development. However, there are important architectural topics which have not been addressed in this paper but are noteworthy, nonetheless. Instruction stream creation, traps and exceptions, and instruction memory hierarchy are three such topics. The latter two are addressed below.

4.2 Traps and Exceptions

It is an objective of the trap mechanism in the Horizon processor is to be as lightweight as possible, requiring no operating system intervention. This means that enough information about the machine state at the time of the trap should be accessible to the user to allow identification of the cause and appropriate recovery. Moreover, the integrity of the system must not be compromised by giving unprivileged programs this level of access.

The proposed traps are listed in Table 1. Two levels of masking are provided to the user for maskable traps. Summary trap mask bits are provided in the ISW for the countdown trap, branch trap, all the floating point traps and access state violation traps so that they

Table 1. Traps

CLASS[1]	TRAP	DETECTING UNIT[2]
m	Countdown	Iselect
u	Hardware Error	All
m	Single Memory Error	IF, M
u	Double Memory Error	IF, M
u	Memory Protection Violation	IF, M
u	Unimplemented Memory Address	IF, M
m	Access State Violation	M
m	Floating Point Overflow	A, C
m	Floating Point Underflow	A, C
m	Floating Point Indefinite	A, C
m	Floating Point Loss of Significance	A, C
m	Branch	C
u	Privileged Instruction Trap	IF, C
u	Unimplemented Instruction Trap	IF, A, C, M
u	Create Fault Trap	M

[1] m : a user-maskable trap
u : an unmaskable trap

[2] A : A-unit
C : C-unit
M : M-unit
IF: Instruction Fetch unit

can be manipulated easily inside target registers. For masking individual types of floating point traps, access state violations, and other traps listed in Table 1, separate mask registers are accessible as part of the processor state.

Because of the pipelined nature of the machine, all instructions, once issued, flow completely through the functional units. When a trap occurs, further instructions from the same i-stream may be prevented from being issued. Some traps are detected very early, even before the offending instruction enters the pipeline. In this case, the instruction(s) already in the pipeline may be completed correctly. However, some traps are not detected until the offending instruction has nearly completed and hence not in time to prevent one or more a subsequent instruction(s) from being issued (if allowed by lookahead). Depending on the nature of the trap, the later instruction(s) are either completed correctly or else their side-effects are inhibited.

Two scenarios were considered for generating traps: trap on generate and trap on use. In the latter scenario, if an error occurred in an operation, the associated destination register would be poisoned for future references, i.e., given a value that when encountered as an operand in a subsequent instruction would cause a trap. This approach has the property that traps are put off as long as possible and in fact may never occur if the offending value is not re-used. Such a scheme is useful for handling unsafe loads, where fetching an operand at the end of a loop for the next iteration causes an invalid address for the last iteration of the loop. Such an error could be ignored if the destination register is not subsequently read before being written again with valid data. However, this mechanism is not sufficient for operations that have no destination register. Specifically, a memory protection violation occurring on a store operation has no destination register value to poison. Another argument in favor of a trap on generate scheme is that a "fatal" trap may not be evidenced until much more (wasted) computation has been done. The potentially large "distance" from the condition which caused the trap to its detection makes debugging more difficult. Consequently, the trap on generate scenario was chosen.

4.3 Instruction Cache

One possible design for an instruction cache for the Horizon processor is simply an instruction prefetch buffer. The buffer should keep at least 6 lines for each active i-stream: one for the PC, one for the PC incremented, and one for each of the 4 targets. When the PC increments or a branch instruction is processed, the next instruction is always ready in the buffer. When an event occurs that changes the set of lines needed for a particular i-stream, the relevant lock is incremented in the lookahead logic for that i-stream. The lock is then decremented when the line is brought into the instruction buffer. With this instruction prefetch mechanism, instruction issue latency is only affected if the instruction prefetch time exceeds the minimum time between successive issues from a single i-stream, which is determined by the dependence on previous instructions and the number of i-streams contending for issue slots.

5. Conclusions

The processor architecture described in this paper combines three levels of parallelism, multiple instruction streams, instruction lookahead, and horizontal instructions to achieve vector-like performance on scalar code. The processor removes the burden of managing resources, hiding the pipeline and register read/write timing, from the compiler, thereby greatly simplifying code generation. Latencies caused by contention or synchronization through the shared memory, creating serious performance degradation in most machines, are effectively hidden by the processor. Each instruction stream is served at at rate commensurate with average operation latency, while the processor sustains a utilization (instructions issued per cycle) approaching unity. There are, however, many unresolved issues to be addressed. From early simulation studies and programming experience on the Horizon simulator, it appears that the underlying processor architecture is well balanced and is capable of achieving the targeted performance.

6. Acknowledgements

The authors gratefully acknowledge the contributions of the architecture team at the Supercomputing Research Center, all of whom share primary roles in the conceptual development and design of Horizon: Paul B. Schneck, Chief Architect of Horizon at the SRC, and James T. Kuehn, both of whom contributed to and provided consultation for this writing, William E. Holmes, Daniel J. Kopetzky, Fred A. More, and David L. Smitley. Steven Melvin is recognized for the origination of the register banking scheme developed originally for the HEP-3. The authors also recognize all those in industry and academia who contributed in no small part to the genesis and evolution of the Horizon processor architecture.

7. References

[1] M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, Vol. 21, No. 2, February 1988, pp 9-21.

[2] J. T. Kuehn and B. J. Smith, "The Horizon Supercomputing System: Architecture and Software," *Supercomputing '88*, submitted for publication, 1988.

[3] F. M. Pittelli and D. L. Smitley, "Analysis of a 3-D Toroidal Network for a Shared Memory Architecture," *Supercomputing '88*, submitted for publication, 1988.

[4] B. J. Smith, "A Pipelined Shared Resource MIMD Computer," *1978 International Conference on Parallel Processing*, 1978.

[5] B. J. Smith, "The Horizon Instruction Set," unpublished note, Supercomputing Research Center, 1987.

[6] J. Draper, "Compiling on Horizon," *Supercomputing '88*, submitted for publication, 1988.

[7] M. R. Thistle, "Floating Point Arithmetic on Horizon," Internal Research Note, Supercomputing Research Center, 1988.

[8] P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill, NY, 1981.