An HPS Implementation of VAX; Initial Design and Analysis

Wen-mei Hwu
Steve Melvin
Mike Shebanow
Chien Chen
Jia-juin Wei
Yale Patt

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

HPS (High Performance Substrate) is a new microarchitecture targeted for implementing high performance computing engines. Like the other aspects of Aquarius, it attempts to achieve high performance, in part, by exploiting concurrency. The HPS execution model is a restriction on fine granularity data flow. This paper reports on the first experiment using HPS, a preliminary design and simulation of the VAX architecture. We briefly describe the execution model of HPS and the rationale for using it to implement the VAX. We then discuss in detail our design approach with respect to a number of the fundamental aspects of a VAX implementation. Finally, we present some encouraging preliminary measurements of a partial simulation of the VAX architecture.

1. Introduction.

HPS is a new microarchitecture specifically targeted for implementing high performance computing engines. It is part of the Aquarius project, and its thrust is consistent with the objectives of the Aquarius project. Aquarius as a whole is an attempt to obtain enormous improvements in computer performance, in part by exploiting concurrency at all levels of transformation, from the design of the algorithms to the use of parallel circuits. HPS represents our approach to dealing with concurrency at the microarchitecture level.

This paper describes our first experiment with the HPS microarchitecture, the design and simulation of a partial implementation of the VAX architecture. The VAX architecture is a very complete ISP architecture. It supports, among other things, more than 300 opcodes, more than 20 addressing modes, a large number of data types, precise interrupt handling, and a virtual memory mapping system. We chose the VAX for our first undertaking partly because of this rich complexity. We felt that experiences gained in implementing this machine couldn’t help but enhance our ability to implement machines of lesser complexity.

The paper is organized in five sections. Section 2 introduces the HPS microarchitecture, explains why it is an inviting approach to implementing the VAX, and discusses some of the fundamental caveats associated with such an implementation. Section 3 delineates the details of the implementation. Section 4 describes the measurements and analyzes the results of those measurements. Section 5 discusses the work we are currently doing with respect to HPS and the VAX architecture.

2. Background Information.

2.1. The HPS Execution Model.

The HPS execution model is a restriction on classical fine granularity data flow; we are calling it "restricted data flow." There are, however, substantial differences between our model and the classical fine granularity data flow engines of Dennis [1], Arvind [2], and others. One important difference is our notion of an "active window," which we define as the set of ISP instructions whose corresponding data flow nodes are currently part of the data flow graph which is resident in the microengine. Consequently, in our model, only a small subset of the entire program is present in our microengine at any instant of time. As the active window moves through the dynamic instruction stream, HPS executes the entire program.

An abstract view of HPS is shown in figure 1. The figure implies that static instructions are prefetched, according to some branch predictor, into a dynamic instruction stream, which is then decoded and merged into the data flow graph corresponding to the active window. In the case of the VAX implementation, this is, of course, the case. In general, however, this is not a necessary part of the HPS specification. Indeed, we are investigating having HPS directly process multinode words (i.e., the nodes of a directed graph) which would be produced as the target code of a (for example) C compiler.
What is essential to our HPS definition is that, for each instruction, the output of the decoder which is presented to the Merger for handling by HPS is a data flow graph.

The Merger takes the data flow graph corresponding to each ISP instruction and, using a generalized Tomasulo algorithm to resolve any existing data dependencies, merges it into the entire data flow graph for the active window. Each node of the data flow graph is shipped to one of the node tables where it remains until it is ready to fire.

When all operands for a data flow node are ready, the data flow node fires by transmitting the node to the appropriate functional unit. The functional unit (an ALU, memory, or I/O device) executes the node and distributes the result, if any, to those locations where it is needed for subsequent processing: the node tables, the Merger (for resolving subsequent dependencies) and the Fetch Control Unit (for bringing new instructions into the active window). When all the data flow nodes for a particular instruction have been executed, the instruction is said to have executed. An instruction is retired from the active window when it has executed and all the instructions before it have retired. All side effects to memory are taken care of when an instruction retires from the active window. This is essential for the correct handling of precise interrupts.

The instruction fetching and decoding units maintain the degree of parallelism in the node tables by bringing new instructions into the active window, which results in new data flow nodes being merged into the data flow node tables.

2.2. Why a VAX Implementation Makes Sense.

Several facts suggest that an HPS implementation of the VAX architecture is a viable alternative for high performance. First, it is consistent with our three-tier model of computing discussed in [4]; that is, irregular parallelism in a program can be best exploited by software restructuring of the global parallelism, as in CEDAR [5], and hardware execution of the local parallelism, as in HPS, with a sequential control flow ISP architecture in the middle, such as the VAX.

Second, an examination of the VAX instruction format and addressing modes argues that a good deal of potential concurrency exists in each VAX instruction. Third, our studies of a number of large VAX instruction traces indicate that most of the parallelism in a VAX program is local, and in fact can be captured with a moderately sized active window.

Finally, the VAX is a memory oriented architecture and as such, has the property that concurrent execution of memory accesses with functional unit computations can smooth out the memory access time if the active window is sufficiently large.

2.3. Caveats.

There are many good reasons to attempt an HPS implementation of the VAX. There are also many caveats. Several are enumerated here. For example, the VAX architecture has a substantial number of subtle details that are irregular in the way they manipulate data items, and therefore do not lend themselves easily to an HPS implementation. A byte destination write into a general purpose register does not touch the high byte. This means either each register must be implemented in four pieces, or some mechanism must be invented to keep track of which parts of a register are to be updated as a result of a destination write. (And we parenthetically note that the rules governing destination writes to registers and the rules governing destination writes to memory are not at all the same.) Also, for example, the setting of condition codes (which ones, and the algorithm for setting them) varies with the opcode.

In addition, VAX allows unaligned memory accesses. This creates problems in at least two ways. (1) HPS must be able to access the byte stored at location A and the high byte of the word stored at location A+1 as the same byte. (2) HPS must be able to manipulate data elements of various sizes which involve at times a single element and at times multiple elements.

Also, since the characteristics of the data flow graph of some VAX instructions depend, in part, on one or more of the operands in the particular instance of the instruction, the HPS decoder can not generate the data
3. Implementation Details.

3.1. Decoding and the Node Cache

A node cache is added in this implementation to match the bandwidth of the decoder to that required by the performance goal. In order to achieve high performance, more than one node per cycle should be merged into the node tables. However, the variable instruction format and some sequential semantics in the instruction set specifications make it difficult to go far beyond supplying one node per cycle from the decoder.

The idea of a node cache is to save the nodes generated for VAX instructions in a much more parallel format than what can be easily generated by the decoder in one cycle. When a particular VAX instruction is decoded the first time, the nodes generated in each cycle go to the merger as well as to the node cache filler. The nodes are compressed into multinode words within which dependencies between nodes are specified explicitly. More specifically, the register dependencies are resolved by a modified Tomasulo algorithm for all nodes in one multinode word. When the instruction is executed the next time due to a loop, the output from the decoder will use these cached multinode words instead of the sequential decoding result.

3.1.1. Node Cache Organization.

The node cache is a direct-mapped cache accessed by the program counter of the instructions. Instructions generating different nodes due to different operand values cause problems for node caching. These instructions can generate different nodes in different execution instances. Complexity involved in checking such caching hazards may greatly reduce the merit of a node cache in terms of performance and effort. As a result, tentatively only the nodes of those instructions whose node generation is invariant among operand values are cached.

The other problem with node caching for a VAX implementation is that the number of nodes generated for each instruction can potentially vary significantly. The simplest instructions can be implemented with only one node. However, the most complicated instructions (e.g., CALLS) can result in more than thirty nodes. The question is whether we should fix the size of each node cache entry or not. If we do, the next question is what should be the size of each entry. The preliminary design has a fixed size of four multinode words per instruction. For those instructions generating more than four multinode words, only the first four multinode words are cached. These four cached multinode words provide some buffering time for the decoder to generate the rest of the nodes.

In a later version, we may partition the node cache into two sections. The first section caches those instructions which result in very few nodes (one or two multinode words). The second section deals with complicated instructions which generate more nodes than can fit in the first section. From past experience, we expect that the first section should have a lot more entries than the second section.

Each node cache entry has two major parts: ISP level instruction information and HPS nodes. The ISP level part is roughly an instruction cache whose content is decoded instruction information instead of fetched instructions. The HPS node part stores the nodes generated for the instructions.

The important logical fields of the ISP level part are described as follows.

- **tag**: This field contains part of the program counter for the cached VAX instruction. It serves as the matching tag to access the cache entries.
- **opcode**: The field contains the opcode of the cached instruction. This field will be shipped to the Active Instruction Table for bookkeeping and retirement purposes.
- **next PC**: This field contains information necessary to obtain the next default instruction counter. In the case where an instruction is found in the node cache, this field efficiently provides the next default program counter. Thus the cache access for the next default instruction can be done even before the current one finishes merging. The preliminary design of the field is an eight bit quantity which will be added to the program counter of the currently decoded instruction to form the next default program counter.
- **target PC**: This field contains information required to obtain the alternative instruction program counter from the default one. When the target address of JMP, JSR, CALLS and CALLG uses absolute or relative mode, the target address will be cached in this field. The target program counter of branch instructions are always saved in this field.

The node part of the cache contains multinode words generated by the corresponding instructions. Each multinode word contains several ALU operation nodes, memory access nodes, literal constants and a branch confirmation node. The multinode words are very large in size. Even for a very small scale HPS engine where a multinode word has only one ALU node, one memory node, one literal constant/branch confirmation node(multiplexed), the size can be as large as seventy-
two bits. However, this is not so much a problem considering the multinode word is stored only in the node cache and transferred to the merger within the processor.

Each ALU node in the cache has the following important logical fields: opcode, two input operand fields, one output register number and a condition code setting field. Opcode field encodes the ALU operation to be performed, data size and exception enabling status. The input operand field specifier specifies the way to get the input operand value, the possible sources are general purpose registers, processor registers, literal values and result of nodes in the same multinode word.

The important logical fields of each memory node are opcode, privileged mode, address operand/register number, data operand and a condition code setting field. Opcode field encodes the memory system operations to be performed and data size if operation is memory access. Memory system operations in the preliminary design include memory accesses, translation buffer management, address translation register manipulation and access protection probing. The rest of the fields are similar to those in the ALU nodes.

Branch confirmation nodes are used to report whether branch prediction made was correct or not. The opcodes of such nodes specifies the way conditions are combined to decide the actual branch direction. The input operands are sources of the condition bits used by this particular branch: condition code bits in PSL or nodes in the same multinode word.

3.2. Register Alias Table Organization.

The key components of the merging mechanism are register alias tables. These tables keeps the dynamic information of busy/ready status and the node which produces the most up-to-date value for most VAX registers in the machine. We use the term alias to refer to the fact that the actual value of a register will be directly forwarded as the result of some other operation rather than fetch from the register itself. The content of the register, in this case, gives the direction of establishing the forwarding link instead of the actual value. In this way, we alias the register to an operation which is going to generate the most up-to-date value of that register.

There are actually three register alias tables in the preliminary VAX implementation. These three tables are the general purpose register alias table, processor register alias table and condition code alias table.

3.2.1. General Purpose Register Alias Table.

There is a one-to-one mapping between the general purpose registers and the general purpose register alias table entries. This part has sixteen entries. Each general register alias table is designed to be a current slot and several backup slots. This design provides fast repair from branch prediction misses. With the restriction on the branch prediction depth, each general purpose register alias table entry can be designed as two back-to-back parts: current and backup.

Each part of a GPR alias table entry has three fields similar to the original Tomasulo algorithm: ready, tag and value. Ready bit indicates whether the value field holds valid data for nodes currently merged. Tag field contains a six-bit tag of the node to supply the register value. Value field contains the 32-bit register value when the ready bit is set. We briefly describe the operations defined on the general purpose register alias table as follows.

Merging. All merging accesses are to the current slot. More specifically, read access for setting up input operand field of a node table entry gets the information from the current slot. When a node writes into a register, it writes the new alias information into the current slot.

phase 0 There is one port for each input operand of every ALU and every memory-access prenode. When an input operand is of GPR mode, the general purpose register entry indexed by the num field is copied to the corresponding node table entry.

phase 1 There is one write port for each ALU prenode and one for each memory-access prenode. When an output operand is of GPR mode, the general purpose register entry indexed by the num field is written. First, the ready bit is cleared. Second, the tag assigned to the node generated from the prenode is written into the tag field. A clear restriction is that, within the same multinode word, no two writes can access the same register. This will be enforced by the node cache filler.

Distribution. The result values are distributed to both the current and the backup slots. This keeps the backup part up-to-date in the ready and value field. The point is to keep the backup part the way it would be if none of the instructions after the branch has entered the machine.

There are two cases for the backup slots. First, when there is no branch prediction performed in the current cycle, comparison is made between distributed tags and stored tag. If there is match, the value accompanying the matching distributed tag is gated into the value field and the ready bit is set.

Second, when there is a branch prediction and current slot is transferred to a backup slot, comparison is made between the distributed tags and tag from the current part. If there is match, the value accompanying the matching distributed tag is gated into the value field and the ready bit is set.

There are three cases for the current part. First, when the entry is not written by either of the currently merged nodes and there is no repair signaled this cycle, comparison is made between the distributed
tags and the stored tag. If there is a match, the value accompanying the matching distributed tag is gated into the value field and the ready bit is set.

Second, when there is a repair signaled in the current cycle, comparison is made between the distributed tags and the tag from the selected backup slot. If there is a match, the value accompanying the matching distributed tag is gated into the value field and the ready bit is set.

Third, when there is no repair signaled but the entry is written by a node currently being merged, nothing happens from the distribution. The write from the merger takes priority.

Prediction. When a branch prediction is done, the current slot of each general purpose register is copied to a backup slot. The register alias table can be designed so that this is single-cycle operation. This happens in phase one and the distribution access described above has already included the operation.

Repair. When a branch prediction miss is signaled, the backup copy is copied to the current part. Again, this can be a single-cycle operation. This happens in clock phase one and the distribution access described above has already included this operation.

3.2.2. Process/Processor Register Alias Table.

The process/processor register (PPR) alias table contains entries for Kernel Stack Pointer (entry 0), Executive Stack Pointer (entry 1), Supervisor Stack Pointer (entry 2), User Stack Pointer (entry 3), Interrupt Stack Pointer (entry 4), Process Control Block Base (entry 5) and System Control Block Base (entry 6). Each entry is designed the same as is the GPR alias table. The access operation is the same as that of the GPR alias table.

3.2.3. Condition Code Aliases Table.

The condition code (CC) alias table handles N (entry 0), Z (entry 1), V (entry 2) and C (entry 3). The design is almost the same as the GPR alias table. The difference comes from the fact that each node can affect more than one CC alias table entry. Thus all access except for merging are the same as that of the GPR alias table.

Merging. Merging is very similar to that of the GPR alias table. However, each node can modify more than one CC alias table entry. This is very different from the write-access characteristics of the other register alias table entries. During phase one, all entries specified by the set_cond field receive the same tag and have their ready bit cleared.

3.3. Node Table Organization.

There are three major types of node tables in the preliminary design: ALU node table(s), memory system node table and branch confirmation node table. Each node table feeds one function unit. The goal is to supply one node from each node table to its corresponding function unit in each cycle.

3.3.1. ALU Node Table Organization.

Each ALU node table has the following important logical fields.

- **Opcode**: The control to the fixed point function unit. It specifies the type of operation to be performed (ADD, SUB, etc.), the data size and the exception enabling status. This field is directly transferred from the proper node cache entry.

- **Tag**: This field is the tag to be distributed with the output value. It will be used to distribute into nodes and the register alias table entries awaiting the result.

- **ID**: This field is the index to the active instruction table entry of the corresponding VAX instruction. It is used to access the repair information when an ALU exception occurs and also used by the retirement mechanism.

- **Inputs**: There are two input operand fields within each fixed node table entry. The format of each input operand field is as follows:

- **Ready**: Whether the value is already in the value field or not.

- **Tag**: The tag of the node, if any, which supplies the value to the input operand.

- **Value**: This field holds the input operand value when the ready bit is set.

Merging. Insertion to the node table operates in a circular queue style. Merging always adds the incoming node to the tail of the node table. The fixed point node table is organized into two sections. The node table has a valid bit for each entry. The stalling condition for the merger is as follows. When both sections have valid entries and tail pointer is at the end of either of the two sections, the merger has to stall.

Phase 1: There is one write port for the merger. The entry written by this write port is indexed by the internally maintained counter which gets incremented whenever a new entry is inserted. All fields of the selected tail entry are affected.

Scheduling. If the tail pointer points to the first section, the oldest ready node in the other section is fired. If there is no firable node in the second section, the oldest ready node in the first section is then fired. The case where tail pointer points to the second section is the symmetry of above. The restriction in the merging part eliminates the need for a rotator finding the oldest entry in a circular list.

Distribution. The operands with the tag field matching the distributed tag on either of the distribution bus is set ready and the result value on that bus is gated into the value filed.
3.3.2. Memory Node Table Organization.

The memory node table handles dependencies of the memory system related nodes. This includes the memory access (READ and WRITE), managing address translation processor registers (LOAD and fetch), and access control probing (PROBE). Memory node table is more complicated than the ALU node table. The reason for the complication is that memory operations have more sequentiaity which can not be easily resolved in the instruction issuing time. Thus the memory operations may not be able to fire even when all input operands are ready. For example, if one memory node invalidates a address translation register or invalidates the TB, then the following memory access nodes can not fire until this node is done. Also, memory reads may depend on previous memory writes but the dependency can not be resolved by merger because the addresses may not be available in merging time. Readers are referred to[6] for more details.

Each memory node table entry has the following format:

- **opcode** This field specifies the operation to be performed by the memory unit. For memory accesses, this field also specifies the data size.

- **mode** This field specifies the privilege mode based on which the operation is performed. This field will be checked against the protection field to see if the access is allowed. This field is distributed with the output data.

- **ID** This field is the same as that in ALU node table entries.

- **inputs** There are two input operand fields in each node entry: address operand and data operand fields. The data operand field of memory read nodes are not used. The input operand format is the same as that in the ALU node table entries. For load and fetch nodes, the data value field gives the memory management processor register number P0BR(0), P0LR(1), P1BR(2), P1LR(3), SBR(4), SLR(5), TIA(6), TBIS(7) and MAPEN(8).

The memory node table is also organized to be a circular queue. The access rule for the memory node table is as follows:

- **Merging**. Merging always adds the incoming node to the tail of the node table. The memory access node table is organized into two sections. The node table has a valid bit for each entry. The stalling condition for the merger is as follows. When both sections have valid entries and tail pointer is at the end of either of the two sections.

- **Scheduling**. The minimum requirement to fire a memory node is that all relevant operands are ready. We state the additional restriction in the preliminary design below. Firing a ready READ node requires that there is no LOAD nodes or (potentially conflicting) WRITE nodes older than it in the table. Firing a WRITE node requires that there is no LOAD nodes older than it in the table. To fire a LOAD node requires that there is no node older than it in the table. To fire a FETCH node requires that there is no LOAD node older than it in the table. A PROBE node is ready when there is no LOAD node older than it in the table. The oldest fireable node is fired in each cycle. This is not the optimum firing rule in terms of concurrency. However, this is feasible in hardware design in our current research stage.

3.4. Branch Confirmation Node Table Organization.

Each branch confirmation node table has the following format.

- **opcodeR** This field specifies the way the input conditions are evaluated to get the actual branch direction.

- **ID** This field is the same as that in the ALU node table.

- **condition** There are two input operand fields in each branch node. Each operand has a mask, a tag, a ready bit and the condition value. The mask indicates which of the condition bits in the operand color will be used. The ready bit indicates whether the condition field is valid or not. The tag holds the tag of the node, if any, which supplies the value to the operand. The condition value is used to evaluate the branch confirmation.

The merging and distributing accesses to the branch node table is the same as those to the ALU node table.

3.5. Exception Handling

3.5.1. Exception Detection.

The preliminary design handles most of the traps and faults defined in the microVAX II specification. There are four major groups of exceptions detected: arithmetic traps/faults, memory management faults, operand reference faults and instruction execution faults.

- **Arithmetic Traps**. The arithmetic traps/faults handled are integer overflow trap, integer divide by zero trap, floating overflow fault, floating divide by zero fault and floating underflow fault. The opcodes of ALU operations specify the exception enable status. The result of the ALU operation has a attached subfield of flags for the exception(s) occurred during the operation. When results with exception flags on are distributed, the exception logic brings the engine into the exception handling mode and tries to execute up to the instruction causing the exception.
Memory Management Faults. The memory management faults handled are access control violation fault and translation not valid fault. The memory write is done, as far as the data path is concerned, when the address translation is done. The memory write in preliminary design distributes a value and an exception flag. When a translation not valid occurs, the value contains the address of the invalid page. Right after the address translation is done, the write node is put into memory write buffer and the exception address/flag are shipped to exception monitoring logic.

The memory read nodes are done when the memory data comes back. There are also value/flags distributed. In case of a successful access, the distributed value is a zero extended long word with the addressed byte in the least significant byte. If the translation not valid fault occurs, this field contains an address within the invalid page.

Operand Reference Faults. The operand reference faults handled are reserved addressing mode fault and reserved operand fault. The reserved addressing mode fault is detected in the decoding time.

The reserved operand mode fault is implemented by setting a reserved operand fault enable bit in the ALU node which checks the range of the operand value. It is handled in the same way as is the arithmetic faults.

Instruction Execution Faults. The instruction execution faults handled are reserved/privileged instruction fault, extended function fault and breakpoint fault. This fault is detected in decoder.

3.5.2. Sequential Retirement.

In order to achieve precise exception handling, instructions are retired from the active window sequentially. An instruction is done when all its nodes finishes execution. This is done by a counter array monitoring the distribution bus. In order to retire a instruction, all instructions before it have to be retired.

When an exception is signaled, the retirement point is locked to the exception causing instruction. The engine tries to retire instructions older than the exception-causing If new exception arises during the process, the retirement point is locked to the oldest exception causing instruction.

Within each active instruction table entry, there are tail pointers of each major data structure before the instruction enters the machine. When an older exception is detected, the distributed ID is used to access these tail pointers to repair the engine to the state before the exception-causing instruction.

3.5.3. Memory and Register Buffers.

There are two major data structures to recover the register and memory state in the face of an exception: register save buffer and memory write buffer. Memory write buffer is also used to forward memory data when a subsequent memory access the same location as does a memory write.

3.5.3.1. Register Save Buffer Structure.

The register save buffer is used only when the exception occurs to recover the register state to where every instruction before the exception-causing instruction are executed and none of the instructions form the exception causing instruction on are executed.

Merging. The original content of the modified register alias table entry is copied to the tail of the register save buffer. All the three fields of that entry are copied.

Distribution. The entries matching the distributed tag on either of the distribution bus are set to be ready; the value on the bus is also gated in.

Retirement. The entries between the internal head pointer and the tail pointer from the active instruction table is invalidate. The head pointer is replaced by the tail pointer from the active instruction table.

Exception. When an exception happens, merging process stalls and all instructions before the exception-causing one are retired first. Active instruction table supplies the pointer which points to the starting entry of backing up in the register save buffer. All entries issued after that point are copied, youngest first, back to the register alias table. At the end of this step, the register alias table gives a consistent state where all register values can be found in the value field of the entries.

3.5.3.2. Memory Write Buffer Organization.

Memory write buffer holds the memory writes until the corresponding instruction is retired. The memory write buffer is organized into a circular queue. Each memory write buffer entry has the following basic fields:

ID The active instruction table index of the instruction issuing the memory write.
address Address for the first byte of the memory write.
sizes The number of bytes to be written.
data The data to be written.

There are also fields in each entry for memory conflict detection and forwarding. The following scheme works only when there is no unaligned accesses to memory. Slight modifications have to be done to incorporate unaligned accesses in this scheme. In the following scheme, the memory write buffer has four columns, one for each byte of a long memory word. Column zero always contains the conflict detection information for location with two least significant address bits 00. Column one serves location with two least significant address bits 01, column two for 10 and column three for 11. Each entry has a valid bit and a data byte.

Merging. Insertion into the memory write buffer operates in a circular queue style. Merging always adds the incoming node to the tail of the node table. The memory write buffer is organized into two sections. When both section have valid entries and the tail pointer of memory write buffer is at the end of either section, the merger has to stall.
Retirement. All memory writes older than the current retirement point are allowed to be released to the memory system. Please recall that the memory writes are considered to be done after the address translation is done. There is no guarantee that the exception caused by memory hardware preventing actual writes cause precise exception.

Scheduling When a memory read node fires from the node table, there are three things happening. First, a bit mask is generated to mask out all memory write younger than the read node from conflict detection.

Second, the upper 30 bits of read address is used to compare against the upper 30 bits of stored write addresses. The match signal is ANDed with the valid bit of each column entry to form the hit signal of each column entry. Each hit signal is masked by the bit mask generated above and priority-encoded within each section. Third, if no hit for the column, the corresponding byte has no data dependency on any of the active memory write buffer entry. A memory request has to be issued to fetch the data. If there is a hit signal, the corresponding entry gives the up-to-date byte data.

When a memory write is released from the node table, there are four things to do. First, an entry pointed by the index field of that node is selected. Second, the address value is stored into the address value field. Third, the two least significant address bits and the size are used to set the valid bit and gate the data into appropriate column. Fourth, the ID form the node is gated into the ID field of the selected entry.

3.6. Memory System

The virtual address translation, access control probing, memory read accesses and memory-related processor register manipulation are all done in the memory function unit. Please see the memory node table and memory write buffer subsection for memory access conflict detection and data forwarding.

4. Preliminary Measurements

In this section we describe the simulation model, report the measurements taken and comment on the results obtained. In particular, we describe the extent to which the VAX architecture was simulated, and the parameters associated with the HPS implementation.

It is important to emphasize at the outset, lest anyone take these results out of context, that this was a partial simulation, and that a number of key aspects of the VAX architecture were not handled. Most important, we note that the simulation of an architecture is at best a sanitized model of a piece of hardware, and that any performance figures seriously claimed on the basis of counting cycles in a simulation should be held in abeyance until working hardware, with all its attendant "gotchas," validate the simulated results. We include the (optimistic) data nonetheless because it gives a first cut at how the HPS can be used to implement a very complex ISP architecture. We are currently continuing this work to include those features of the VAX architecture not present as of this writing.

4.1. The Simulation Model.

The major characteristics of the simulation model are the following: the maximum active instruction window, the cache, the branch prediction scheme, the node merging rate, and the penalty of branch prediction failure. The model is established to catch the parallelism in the VAX instruction streams which can be potentially exploited by HPS.

All the VAX addressing modes were implemented. Given a VAX instruction, data flow nodes are generated to evaluate the address of input/output operands and to fetch input operands if necessary. Writing output operand values to registers or memory is part of the opcode execution process and is done only when the opcode is implemented in the simulator.

As of the date of the measurements reported in this paper, we had implemented the 115 most frequently used opcodes from the VAX instruction set. If a VAX opcode were implemented, all the input operand address evaluation, input operand fetch, output operand address evaluation, opcode execution, and output operand write were carried out by data flow nodes. Otherwise, only input operand address evaluation, input operand fetch, and output operand address evaluation were taken care of by data flow nodes. Opcode execution and output operand writes were not.

Since the operand processing represents a significant part of the VAX instruction execution, the execution time for those instructions with unimplemented opcodes was substantial. Furthermore, we did not count those instructions with unimplemented opcodes when we calculated performance in terms of MIPS. This makes the performance figures not overly optimistic. The fact that some of the unimplemented opcodes are more complicated than most of the implemented ones should not significantly reduce the performance when the implementation is complete because only the execution phase and destination write are still required for each of them.

For example, in Table 1, 10,000 instructions in benchmark #1 were visited and 3,460 of them have implemented opcodes. When we calculated performance,

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{Simulator status} & 1 & 2 & 3 & 4 & 5 \\
\hline
\text{Number of instruction examined} & 10000 & 10000 & 10000 & 10000 & 10000 \\
\hline
\text{Number of instructions whose operand(s) were processed} & 10000 & 10000 & 10000 & 10000 & 10000 \\
\hline
\text{Number of opcodes executed} & 9460 & 8900 & 9635 & 9406 & 8033 \\
\hline
\text{Number of distinct opcodes recognized} & 45 & 51 & 29 & 42 & 52 \\
\hline
\end{array}
\]

\textbf{Table 1. VAX Opcodes Processed}
9,460 instead of 10,000 is divided by the time used by the simulated machine. The time used by the simulated machine includes the time to partially execute those 540 instructions with unimplemented opcodes. Therefore, even though some of the unimplemented opcodes may take several data flow nodes to implement, the negative impact on performance should be small when the implementation is completed.

The instruction window size was set sufficiently high (32) that it should not be a performance limitation for any node merging rate that we experimented with. We expect that for low node merging rates this upper limit of 32 will never be achieved and the window size can be lowered without affecting performance.

4.2. The Measurements.

This section describes the potential performance which might be achieved for various instruction decoding rates. Two cases are considered, a microengine having the capability of merging eight nodes per cycle, and a microengine having the capability of merging only one node per cycle. The eight node per cycle decoder provides some idea of what is feasible if a substantial investment in decoding hardware is specified or if a node cache is used in a computing environment where a loop intensive program is executed. The one node per cycle decoder gives some indication of the potential of HPS even with a very limited hardware implementation.

In Table 2.a, the microengine has the capability of merging eight nodes per cycle into the HPS data path. The potential performance obtained for this case was between 6.16 and 7.27 MIPS for the five benchmarks. The hardware requirements were measured in terms of two critical hardware resources: node table size and distribution network bandwidth. The size of the node table supports our claim on the low synchronization overhead. That is, we can design the node table with fast and expensive logic because the size is so small.

The second measurement projects that a decoding rate of one node per cycle can deliver 1.5 MIPS (see Table 2.b), while requiring a node table of only three entries and a result distribution network capacity of only two results per cycle.


There are two major efforts going on regarding implementing the VAX architecture with HPS. One is a parallelism analyzer which is designed to study the local concurrency behavior in VAX instruction streams. This tool allows us to gather data on the available local parallelism given a set of user specified hardware resources. The analysis will help characterize the local parallelism based on which HPS model is constructed. It will also help us to specify the hardware required for exploiting a desired amount of local parallelism.

The other VAX implementation effort is a register transfer level simulation of a limited-resources HPS implementing the microVAX architecture. This small scale HPS engine is targeted to a chip set implementation. Eventually we will do a large scale HPS engine to study the high end of the HPS/VAX.

We are also studying the effectiveness of HPS as a high performance microarchitecture for other applications. These implementation targets under study are a generic multinode-word instruction HPS machine, a C machine and a Prolog processor. From these experiments, we will investigate the limits of the HPS microarchitecture, both from the standpoint of a minimal implementation and from the standpoint of a Cadillac version.

Acknowledgements.

The authors wish to acknowledge first the Digital Equipment Corporation for their generous support of our research. Linda Wright, formerly Head of Digital's Eastern Research Lab in Hudson, Massachusetts, provided an environment during the summer of 1984 where our ideas could flourish; Bill Kania, formerly with Digital's Laboratory Data Products Group, was instrumental in DEC's providing major capital equipment grants that have greatly supported our ability to do research. Digital's External Research Grants Program, also provided major capital equipment; and

<table>
<thead>
<tr>
<th>Measurements</th>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance (MIPS, 200ns clock)</td>
<td>6.25 6.16 7.27 6.50 6.37</td>
</tr>
<tr>
<td>result distribution network bandwidth</td>
<td>6 7 7 6</td>
</tr>
<tr>
<td>node table size</td>
<td>35 35 35 35 35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Measurements</th>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance (MIPS, 200ns clock)</td>
<td>1.39 1.50 1.34 1.37 1.50</td>
</tr>
<tr>
<td>result distribution network bandwidth</td>
<td>2 2 2 2 2</td>
</tr>
<tr>
<td>node table size</td>
<td>3 3 3 3 3</td>
</tr>
</tbody>
</table>
Fernando Colon Osorio, head of Advanced Development with Digital's High Performance Systems/Clusters Group, provided funding of part of this work and first-rate technical interaction with his group on the tough problems. We also acknowledge that part of this work was sponsored by Defense Advance Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089. Finally, we wish to acknowledge our colleagues in the Aquarius Research Group at Berkeley, Al Despain, presiding, for the stimulating interaction which characterizes our daily activity at Berkeley.

6. References.