Run-Time Generation of HPS Microinstructions From a VAX Instruction Stream

Yale N. Patt, Stephen W. Melvin, Wen-mei Hwu, Michael C. Shebanow, Chien Chen, Jiajuin We,

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

ABSTRACT

The VAX architecture is a popular ISP architecture that has been implemented in several different technologies targeted to a wide range of performance specifications. However, it has been argued that the VAX has specific characteristics which preclude a very high performance implementation. We have developed a microarchitecture (HPS) which is specifically intended for implementing very high performance computing engines. Our model of execution is a restriction on fine granularity data flow. In this paper, we concentrate on one particular aspect of an HPS implementation of the VAX architecture: the generation of HPS microinstructions (i.e. data flow nodes) from a VAX instruction stream.

1. Introduction

The VAX architecture has been implemented in several different technologies using a wide range of microarchitectural features. At the high end, the heavily pipelined VAX 8600 requires on the order of 6 cycles to execute each VAX instruction. Critics of the architecture suggest that there are specific characteristics of the VAX which preclude a very high performance implementation. As one step toward addressing this criticism, we will examine the process of decoding a VAX instruction stream into HPS microinstructions (i.e. fine grain data flow nodes).

HPS (for High Performance Substrate) is a microarchitecture we have developed which is specifically geared for implementing high performance computing engines. Like most other proposals for high performance, it exploits concurrency, but in our case, by a variation on the classical fine grain data flow scheme. Our model allows a control flow architecture to be implemented by a data-flow-oriented microarchitecture. We call our approach "restricted data flow" because at any one time, the only data flow nodes (i.e. microinstructions) present in the microengine are those corresponding to a restricted part of the dynamic instruction stream. An introduction to this model of execution and the rationale for why we believe it has potential for implementing high performance engines is contained in [2]. A discussion of the general issues involved in designing an HPS machine is contained in [3].

We are currently investigating the viability of HPS for implementing several sequential control flow ISP architectures, including the VAX. This paper examines the problems that we have observed in the decoding portion of an HPS implementation of the VAX architecture. In some cases, there is a fundamental mismatch between the VAX and HPS. In many cases, however, the problem can be solved with additional hardware. As we will see, frequently, the problem can be handled by optimizing for the typical case, while providing non-efficient support for the seldom occurring general case.

The paper is divided into seven sections. Section 2 contains an overview of the HPS microarchitecture. Section 3 describes how the VAX architected registers are handled and in particular the partition into "static" registers and "dynamic" registers. Section 4 describes the individual problems involved in the generation of HPS nodes for each VAX instruction. Section 5 describes the problems associated with branches and section 6 discusses the caching of nodes. Finally, in Section 7, we offer a few concluding remarks.

2. The HPS Execution Model

An abstract view of HPS is shown in figure 1. Instructions are fetched and decoded from a sequential instruction stream, shown at the top of the figure. From this instruction stream, the decoder produces a data flow sub-graph for each instruction. The decoder is made up of the static registers, the node generator, the branch predictor and the node cache.

The static registers are internal processor registers which are changed infrequently or must be known by the decoder. They are discussed in more detail in section 3. The node generator provides nodes (or microinstructions) including branch nodes to the branch predictor which then supplies a dynamic node stream to the merger. This scheme is in contrast to other HPS implementations in

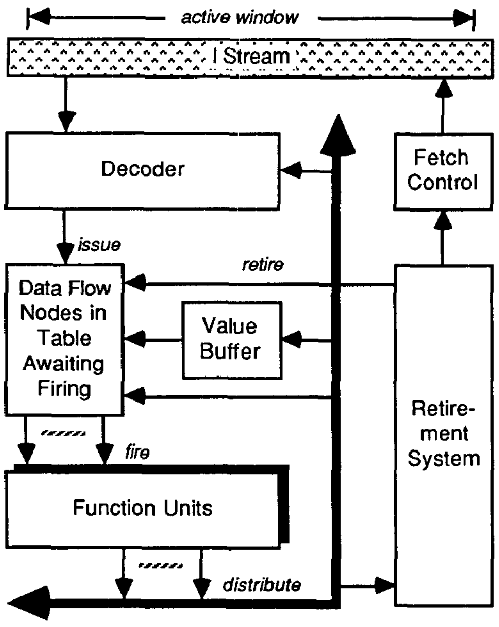

Figure 1.
Astract View of HPS

which the branch predictor supplies a dynamic instruction stream to the node generator [2](see figure 2). The node cache stores the decoded instructions, so that the nodes for instructions re-executed within a short period of time aren't regenerated.

The merger takes the data flow graph corresponding to each instruction and, using a generalized Tomasulo algorithm [4] to resolve any existing data dependencies among the dynamic registers (i.e. the general purpose registers and the condition codes), merges it into the data flow graph present in the machine. Each node of the data flow graph is sent 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 node "fires" (i.e., it is transmitted to the appropriate function unit). The function unit (an ALU, or the memory-I/O unit) 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 decoder (for confirming branch predictions). When all the data flow nodes for a particular instruction have been executed, the instruction itself is said to have executed. An instruction is retired when it has been executed and all of the instructions before it have retired. All side effects to memory are held until the associated instruction retires. This is essential for the correct handling of precise interrupts [1].

An important part of the specification of HPS is the notion of the active instruction window. Unlike classical data flow machines, it is not the case that the data flow graph for the entire program is in the machine at any one time. We define the active window as the set of ISP instructions whose corresponding data flow nodes are currently being worked on in the data flow microengine. Parallelism which exists within the window can be fully exploited by the microengine. The decoder maintains the degree of parallelism by bringing new instructions into the active window as old ones are retired, which results in new data flow nodes being merged into the node tables.

3. Static and Dynamic Registers

We have partitioned the VAX architected registers into two sets, static and dynamic. Static registers are those which are changed infrequently or must be known by the decoder while dynamic registers are those which are changed frequently and don't have to be known by the decoder. When an instruction is decoded which modifies a dynamic register, the new value may not be known. It may depend on other instructions which have not yet executed. But this is not a problem, the nodes that calculate the new value are merged into the node tables where they

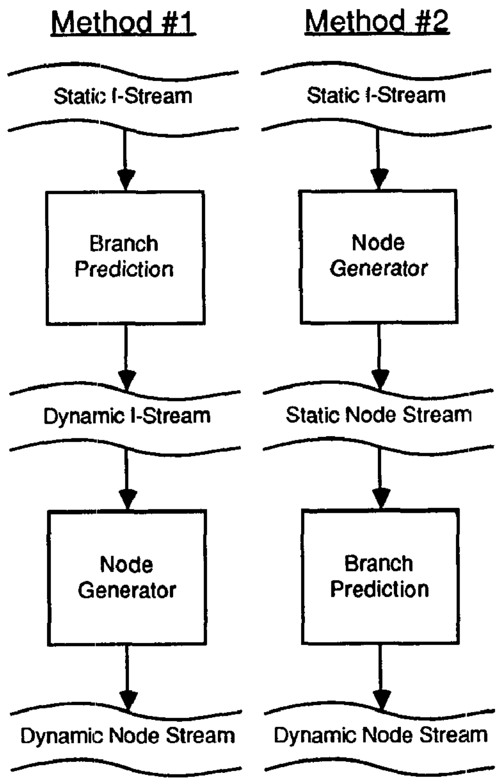

Figure 2.
Two Different Decoder Designs

will wait for the value they depend upon to be computed. If, on the other hand, an instruction is decoded which modifies a static register, decoding must in general halt until the new value is known and the node cache must be flushed.

3.1. Static Registers

All bits in the PSL other than the condition code bits as well as all VAX internal processor registers are static registers. We will describe the mechanisms necessary to implement these registers below.

First consider the T, TP, IV, DV and FU bits. The T and TP bits must be manipulated directly by the decoder. At the beginning of each instruction (other than BPT), T is copied to TP and if TP is set at the end of an instruction, control is transferred to the trace trap handler. The three trap enable bits (IV, DV and FU), must be attached to the relevant arithmetic nodes in order for the function units to know whether or not an exception should be requested. For example, if the FU bit is set, then a floating point underflow should cause the function unit to notify the exception handler. Otherwise, a floating point zero is produced and no exception occurs. If the T bit or the trap enable bits are changed by the BISPSW or BISCPSW instructions, the new value must be known by the decoder. Since the mask operand may not be an instruction stream literal, the node generator may not know how these bits are being changed which would force the decoder to wait until the new values becomes known.

The Current Mode field is a two bit field that specifies the current privilege level of the machine. The decoder must know the value of the Current Mode field for three reasons. First, it must be attached to all memory nodes so that the privilege assigned to the page in memory can be compared to the current privilege level. Second, some instructions behave differently depending on the privilege mode so the decoder must know how to generate the nodes for the instruction as well as where to transfer control. Finally, the Current Mode must be used to interpret accesses to R14 (the stack pointer). The IS field is also needed for this. When an instruction refers to R14 (e.g., a PUSHL), the current mode field and the IS field must be used to decide which of the five stack pointers the instruction actually refers to. The Previous Mode field must also be known, since it has to be attached to PROBER and PROBEW instructions.

Fortunately, these fields are changed in predictable ways. The Current Mode and Previous Mode fields are changed by the CHMK, CHME, CHMS and CHMU instructions, so the opcode reveals the new value. The only problem occurs with the execution of an REI instruction, which restores the fields from information on the stack. In this case, the decoder must wait until the memory read returns. This is a problem analogous to the problems encountered for RET and RSB that will be discussed below.

The FPD bit also must be known at decode time since it will determine which nodes are merged. For example, if a MOVC3 is decoded with FPD set, nodes must be merged which use the general purpose registers to continue the move where it left off. If, on the other hand, FPD isn't set, the nodes that are generated must start from the beginning. Other considerations concerning non-atomic instructions are discussed below.

The VAX internal processor registers, which are also static registers, are accessed through the MTPR and MFPR instructions. Although it is possible to specify a MTPR instruction in which the register being written is not known, the only instances of MTPR in our trace study had literal register number operands (more details about our trace study are discussed below). If an MTPR instruction refers to one of the five stack pointers, we need only to generate the nodes to change the register. All five stack pointers are dynamic registers, so they can be changed at any time by merging nodes and updating the register alias table.

If, on the other hand, a MTPR instruction writes to any other register, the decoding process must wait until the new value is known. Consider the registers that control the virtual address translation mechanism. If a process writes into one of these registers, then every virtual address access after this point could be affected (including opcode fetches). Since the memory-I/O unit can't handle some nodes that have old virtual addresses and some nodes that have new virtual addresses, the only solution is to halt the decoder and allow the node tables to drain when this occurs. Changes to these registers, however, are infrequent.

3.2. Dynamic Registers

The general purpose registers (i.e. R0 — R13 and the five stack pointers) and the condition codes form the dynamic registers. The condition codes were included in the set of dynamic registers due to the fact that most instructions write to them and a non-trivial number read them. The instructions that read them are: ADWC, SBWC, MOVPSL, the B (branch) instructions, SVPCTX, BICPSW, BISPSW, CALLS and CALLG. Each condition code bit is maintained independently since some instructions only modify a subset of the four bits. Whenever the condition code bits need to be gathered into a word, as when the PSW is pushed onto the stack, special nodes are merged that combine them if necessary. It is unlikely that the condition code bits will have to come from more than one node, and very unlikely that they will have to come from more than two nodes but they may have to come from four different nodes.

In order to confirm branch predictions for the B instructions, the alias that is current for the relevant condition code bit(s) must be recorded. Then, when the node or nodes are distributed, the prediction can be tested.

Clearly, the condition code mechanism adds a significant level of complication to the machine.

4. Node Generation

First let's address the problem of converting instructions into nodes on an instruction by instruction basis, without considering where the instructions come from or where the nodes go after they are generated. We will consider problems that can prevent the generation of nodes for a particular instruction. First, we will look at specific instructions for which this is the case, and the frequency of occurrence. Then, questions concerning cost/performance tradeoffs in the node generator will be addressed. Finally, special considerations necessary to handle non-atomic instructions will be examined.

4.1. Node generation stalls

The process of generating nodes is fairly simple for most instructions. The information required is present in the instruction stream, so it is possible to generate the nodes without relying on any other information. However, for some instructions the information required may not be present in the instruction stream, so a stall may be necessary. Here we use the term *stall* to refer to the situation where the node generator must wait for information that is not part of the instruction itself before it can generate the complete data flow sub-graph for that instruction.

Node generation stalls fall into two categories, depending on what the node generator needs to complete its job. An *operand stall* occurs if the node generator needs the value of an operand in order to generate the nodes, and the needed value is not an instruction stream literal. A *non-operand stall* occurs if information other than operand values is needed. The reason for this distinction will become clear momentarily. First let us look at some examples of stalls.

Some VAX instructions which cause operand stalls are: **PUSHR**, **POPR**, **BISPSW**, **BICPSW** and the four **POLY** instructions. The first four require register masks before they can generate their nodes. PUSHR and POPR push and pop respectively exactly those registers specified by the mask. BISPSW and BICPSW set and clear respectively bits in the PSW corresponding to bits specified by the mask (see section 6). In each of the above cases, the nodes cannot be generated until the mask value is read, and the mask may be an instruction stream literal, the contents of a register, or the contents of a memory location. In the worst case, it could depend on the previous instruction, which could cause the node tables to drain while waiting for the mask. However, in our trace study of over 7 million instructions, of which about 26,000 were PUSHR, 34,000 were POPR and 200 were BISPSW, we didn't find a single instance of an instruction without a literal register mask (see figure 3). This suggests that being able to specify these masks so generally is not a very important architectural feature and simply stalling the node generator when this happens is a perfectly acceptable solution.

The POLY instructions take three operands (an ar-

Branch Destination Stall Status	Node Destination Stall Status: no stall			operand stall			non-operand stall		
no stall	(all others)		92.618%	PUSHR	0.344%	0.000%			
				POPR	0.440%	0.000%			
				BISPSW	0.003%	0.000%			
				BICPSW	---	---			
				POLY	---	---			
operand stall	JMP	0.020%	0.013%				CALLS	1.288%	0.934%
	JSB	0.917%	0.895%				CALLG	0.001%	0.000%
	CASEB	0.381%	0.000%						
	CASEW	0.171%	0.000%						
	CASEL	0.393%	0.000%						
non-operand stall	RSB	2.013%	2.013%				RET	1.349%	1.349%
	REI	0.063%	0.063%						

Summary

Dynamic frequency of node generation stalls:	2.2829%
Dynamic frequency of branch destination stalls:	5.2667%
Dynamic frequency of any stall:	5.2667%

Figure 3.
Trace Study Results -- Stall Analysis

gument, a degree and an address of a table of coefficients) and compute a polynomial. If the degree operand is an instruction stream literal, the nodes to compute the polynomial can immediately be generated. However, the node generator must stall if the degree is stored elsewhere. As with PUSHR, etc. the access type for the degree operand is "read" which means that the operand can come from a register, memory, etc. and in the worst case could drain the node tables. We found no instances of POLY instructions in any of our trace files, so we have no hard statistics. However, we know that in the UNIX 4.3 BSD distribution of library math functions, the only instances of POLY have literal degree operands. It seems likely that this is the case for virtually all instances of POLY. So, as before, this problem can be solved by just having the node generator stall without a significant performance penalty.

Three VAX instructions that cause non-operand stalls are: **CALLS**, **CALLG** and **RET**. The first two are the general procedure entry instructions that create stack frames for the called procedures. The problem that the node generator has with these instructions is that the register save mask, which specifies which registers to save, is stored in the first two bytes of the destination address. Thus, the node generator must stall until it gets this value from memory.

RET is the formal procedure return instruction. It deletes the stack frame and restores the registers. The register save mask is pushed onto the stack by the CALL instructions, so the node generator must pop it from the stack and use it to generate the nodes that will restore the registers. Note that the situation for RET is even worse than that for the CALL instructions because the register save mask is in the data stream, not the instruction stream. Since the masks are part of the instruction stream, they can be read directly by the prefetcher for CALLS and CALLG. For RET, however, the memory read must go through the main data path since the desired information is on the stack and could be in the memory write buffer.

4.2. Cost / Performance Tradeoffs

Once the node generator has all the information needed to generate the data flow sub-graph for a VAX instruction, the next issue is the optimality of the sub-graph generated. Data flow graphs can be generated only one instruction at a time (i.e., no optimization is possible across instruction boundaries), so the question is how good is the set of nodes generated for a particular instruction.

First, suppose we could devote unlimited resources to the node generator. We could generate a graph of minimum height that maximizes the nodes at the lower levels. We could generate a graph with the minimum number of nodes (which might be taller than the minimum height). Or, we could have some strategy in between. The characteristics of the microengine would influence this decision of the preferred node generation strategy. If the function units are always busy, and the potential latency that could occur is not a problem, then a greater speed up would be obtained by always generating the minimum number of nodes, even if they are all sequential.

Next is the question of how important it is to generate the preferred node pattern. Because the VAX architecture is complex, situations arise where it is much harder to generate nodes optimally than to generate them slightly sub-optimally. The frequency of these cases and the degree of sub-optimality must be weighed against the additional cost that would be required. For example, operand specifiers are defined in the architecture to be interpreted sequentially. So, if an operand specifier has a side effect, as with auto-increment and auto-decrement, for example, the side effects must semantically occur in order. However, all operations must be interpreted together in order to generate nodes optimally. For example, we would probably want the instruction

ADDL2 (R0)+, (R0)+

to generate a node that adds 4 to register 0 and a node that adds 8 to register 0 instead of two sequential nodes that add 4. Detecting cases like this that are probably rare, rather than generating the slightly sub-optimal graph is probably poor utilization of hardware.

However, there are cases where it is much more important to optimize. Some VAX instructions are so general that if some of the operands are literals, the node pattern is much simpler. For example, the FFS instruction requires four operands; if none are known to the node generator, 15 nodes are needed to implement the instruction. If, however, ⟨startpos⟩ is a literal and ⟨size⟩ = 32, only 3 nodes are required. Furthermore, in our trace study, every single instance of this instruction had literals for these two operands, and 54.63% of the time the second operand was equal to 32. This case is therefore much more important to optimize.

4.3. Non-atomic Instructions

The VAX architecture contains 29 instructions which are not treated as atomic units. That is, they may be interrupted after having completed only part of what the instructions specifies, and later resumed where execution left off. These instructions fall into four categories: the 11 character string instructions, the 16 decimal string instructions, the **EDITPC** instruction and the **CRC** instruction. The most important single characteristic of all of these is that they must be decomposed into many instructions before they can be merged into the HPS microengine. We must be able to retire part of the instruction before the entire instruction has been completely merged, since the node tables would not be large enough to hold all the nodes corresponding to a string instruction. In this case, what the HPS main data path sees as an instruction may not correspond exactly to a VAX instruction. This is not a problem, however, since the VAX first part done (FPD) bit precisely addresses this issue. When the instruction is interrupted, the state is maintained in the general purpose registers, the FPD bit is set and control is passed to the interrupt or exception handler.

Consider the issue of converting these instructions into the loops that they specify. Typically the inner loops of these instructions involve a read from memory, an operation (which may involve another memory read), a write to memory and a test for the terminating condition. The node generator will generate a group of nodes that implements the loop as well as the startup and termination portions and the branch predictor will supply a dynamic node stream to the merger. This allows several iterations of the loop to be executed simultaneously, assuming that there are no data dependencies.

However, most string instructions do not have a single inner loop. For example, in MOVC5 if the source length is less than the destination length, but greater than zero, then there will be both a *move* inner loop and a *fill* inner loop. Also, some string instructions (e.g., MOVC3, MOVC5, MOVTC) are specified such that if source and destination strings overlap, the results will still be correct. This means that the operation must be able to proceed from high to low addresses or from low to high addresses. So, depending on the operands, two different inner loops are needed.

5. Branches

Redirection of the instruction stream also introduces problems. A *branch* is any change in the flow of control of the instruction stream. Problems occur either if we cannot determine the destination of a branch, or if we know the destination, but can't predict the outcome of the condition on which the branch depends.

5.1. Branch destination stalls

If the destination of a branch instruction is not known, the node generator must stall as described above, but for a different reason. We classify branch destination stalls in the same way node generation stalls were classified, into *operand stalls* and *non-operand stalls*. In the first case, the branch destination is determined by an operand which may or may not be an instruction stream literal. In the second case, the destination is determined by some other information.

Most instructions don't stall for branch destinations. This includes all instructions which don't change the flow of control, as well as **B**, **BR**, **BB**, **BLB**, **ACB**, **AOB**, **SOB** and **BSB**. These instructions have branch destinations that are always computable from information contained in the instruction stream.

Instructions which may cause operand stalls are: **JMP**, **JSB**, **CALLS**, **CALLG** and **CASE**. The first four have destinations that are computable from their operand specifiers, but the computation often requires the contents of a register or memory location. There will be no stall, therefore, if the operand has absolute or any PC relative addressing mode. We have found that a stall actually occurs in these instructions a significant percentage of time, unlike the node generation operand stalls discussed above that virtually never stall. Thus, it seems that being able to specify branch destinations that aren't static is a widely used architectural feature.

The CASE instruction is in this category for a different reason. All of its destinations are literals, but the (limit) operand may not be a literal, so we can't even interpret the instruction stream until we know its value. The number of branch destinations that follow the instruction depends on the limit operand. In our trace study we have found not a single instance of a CASE instruction without a literal or immediate limit operand. Thus, this is another example where the generality of the architecture isn't used, so the HPS implementation need not handle it efficiently.

The instructions **RSB**, **RET** and **REI** cause non-operand stalls. These return instructions get their destinations from the stack, so they will cause stalls while the destination is being read from memory. These instructions illustrate a problem in adapting a classical stack based call-return architecture to HPS. In order to keep the node generation process from stalling, the return address is needed as soon as the return instruction is decoded. However, the saved PC is on the stack in data memory, and we may not even know where, since the stack pointer may depend on nodes currently pending execution. Combining control information with data in this way is a problem for HPS. A separate stack for return addresses that cannot be changed by non-control instructions would be easier to implement. But given the architectural constraints of the VAX, not much can be done in the general case.

However, if we consider how most call-return stacks are used in the VAX, we can optimize on the typical case. In the CALLS, CALLG, RET mechanism, the saved PC is saved in the stack frame, which shouldn't change from within a procedure. Thus, the frame pointer should be ready when a RET instruction is encountered, so the PC read could be performed. Further optimization would be possible by caching PC writes in the machine and allowing the node generator access to them. Then, in order to preserve the VAX architecture, we would have to have a mechanism that checked for writes into the stack frame.

A summary of the stall classes presented above is presented in figure 3. There are two categories of stalls (node generation and branch destination), and three types of stalls within each category (no stall, operand stall and non-operand stall). Since the two categories are independent, there are nine different stall classes (of these, only six contain instructions). Figure 3 shows the VAX instructions in each stall class, and the frequency of occurrence in our trace studies. Note that the operand stall classes may not stall all of the time, so for these classes we also provide the frequency of an actual stall.

6. Caching Nodes

It is difficult to generate nodes very fast from a VAX instruction stream due to the sequential decoding problem. The length of an instruction depends on all of its operand specifiers and the length of an operand specifier can vary greatly. Therefore, it is even difficult to decode two sequential instructions in parallel. Furthermore, since the main data path is capable of executing several nodes per cycle, and given that each VAX instruction produces on the average approximately 4 nodes, a decoding rate of at least one instruction per cycle is required to keep the function units busy. We conclude that a node cache is required to hold the decoded instructions. However, we may not be able to cache the nodes of every instruction. An instruction that causes a node generation stall probably cannot be cached. This is because an instruction may stall for a different value each time it is encountered.

Stalling instructions are not always non-cachable, however. Consider the possibility of caching the nodes of CALL or RET instructions. If we assume no self-modifying code, a particular instance of CALLS or CALLG will always use the same register mask as long as the destination is static. A particular instance of RET *should* always use the same register mask, but the VAX architecture doesn't prevent a program from altering the register mask on the stack. The simplest solution is to not cache the nodes in this case, but they could be cached as long as a test were included to guarantee that if the mask had changed it would be detected.

7. Conclusions

The highest performance implementation of the VAX architecture currently available requires about 6 cycles on the average to execute each VAX instruction. A reasonable question to ask is whether there are specific characteristics of the VAX that rule out an implementation that, say, could execute one VAX instruction per cycle.

We have developed a microarchitecture (HPS) which is specifically intended for implementing very high performance computing engines and we are investigating its suitability for implementing the VAX architecture. In this paper, we have examined one aspect of an HPS implementation of the VAX: the generation of HPS microinstructions from a VAX instruction stream.

We have sketched the basic design of such a decoder and have examined the places where performance bottlenecks would arise. Some, but not all, of these bottlenecks can be overcome with additional hardware and by careful consideration of the frequency of certain occurrences. However, there are definitely characteristics of the VAX architecture that make an HPS implementation more difficult.

A property that is ideal for an HPS decoder is the ability to generate nodes and determine branch destinations from information present only in the instruction stream. We saw that most VAX instructions have this property and many that don't in general, do in the typical case. However, an exception that cannot be ignored is the case of the return instructions. In order to handle VAX return instructions efficiently in HPS, substantial hardware may be required. A modified VAX architecture would be much easier to implement. By separating control information and data, the HPS microengine could operate more effectively.

8. Acknowledgement

The authors wish to acknowledge the Digital Equipment Corporation for their generous support of our research, in particular Bill Kania for providing us with the VAX 8600 in order to enhance our ability to do research in microarchitecture and microprogramming; also, Fernando Colon Osorio, Mario Troiani, Nii Quaynor, Steve Ching and Harold Hubschman, from DEC's High Performance Systems and Clusters Group in Marlboro. Our work in microarchitecture is part of a larger architectural research effort at Berkeley, the Aquarius Project. We acknowledge our colleagues in the Aquarius group for the stimulating environment they provide. Finally, we acknowledge that part of this work was sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089.

REFERENCES

[1] Anderson, D. W., Sparacio, F. J., Tomasulo, R. M., "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM Journal of Research and Development*, vol. 11, January, 1967, pp. 8-24.

[2] Patt, Yale N., Hwu, Wen-mei, and Shebanow, Michael C., "HPS, a New Microarchitecture: Rationale and Introduction," *The 18th International Microprogramming Workshop*, Asilomar, CA, December 1985.

[3] Patt, Yale N., Melvin, Stephen W., Hwu, Wen-mei, Shebanow, Michael C., "Critical Issues Regarding HPS, A High Performance Microarchitecture," *The 18th International Microprogramming Workshop*, Asilomar, CA, December 1985.

[4] Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, January, 1967, pp. 25-33.

[5] *VAX Architecture Handbook*, Digital Equipment Corporation, 1981.