A characterization of processor performance in the VAX-11/780

Conference Paper *in* ACM SIGARCH Computer Architecture News · June 1984

DOI: 10.1145/773453.808199

2 authors, including:

Joel S. Emer
NVIDIA
142 PUBLICATIONS **9,389** CITATIONS

A Characterization of Processor Performance in the VAX-11/780

Joel S. Emer

Digital Equipment Corp.
77 Reed Road
Hudson, MA 01749

Douglas W. Clark

Digital Equipment Corp.
295 Foster Street
Littleton, MA 01460

ABSTRACT

This paper reports the results of a study of VAX-11/780 processor performance using a novel hardware monitoring technique. A micro-PC histogram monitor was built for these measurements. It keeps a count of the number of microcode cycles executed at each microcode location. Measurement experiments were performed on live timesharing workloads as well as on synthetic workloads of several types. The histogram counts allow the calculation of the frequency of various architectural events, such as the frequency of different types of opcodes and operand specifiers, as well as the frequency of some implementation-specific events, such as translation buffer misses. The measurement technique also yields the amount of processing time spent in various activities, such as ordinary microcode computation, memory management, and processor stalls of different kinds. This paper reports in detail the amount of time the "average" VAX instruction spends in these activities.

1. INTRODUCTION

Processor performance is often assessed by benchmark speed, and sometimes by trace-driven studies of instruction execution; neither method can give the details of instruction timing, and neither can be applied to operating systems or to multiprocessing workloads. From the hardware designer's or the computer architect's point of view, these are serious limitations. A lack of detailed timing information impairs efforts to improve processor performance, and a dependence on user program behavior ignores the substantial contribution to system performance made by operating systems and by multi-processing effects.

In this paper we use a novel method to characterize VAX-11/780 processor performance under real timesharing workloads [13]. Our main goal is to attribute the time spent in instruction execution to the various activities a VAX instruction may engage in, such as operand fetching, waiting for cache and translation buffer misses, and unimpeded microcode execution. Another goal is to establish the frequency of occurrence of events important to performance, such as cache misses, branch instruction success, and memory operations. Throughout this paper we will report most results in frequency or time *per VAX instruction*. This provides a good characterization of the overall performance effect of many architectural and implementation features.

Prior related work includes studies of opcode frequency and other features of instruction-processing [10, 11, 15, 16]; some studies report timing information as well [1, 4, 12].

After describing our methods and workloads in Section 2, we will report the frequencies of various processor events in Sections 3 and 4. Section 5 presents the complete, detailed timing results, and Section 6 concludes the paper.

2. DEFINITIONS AND METHODS

2.1 VAX-11/780 Structure

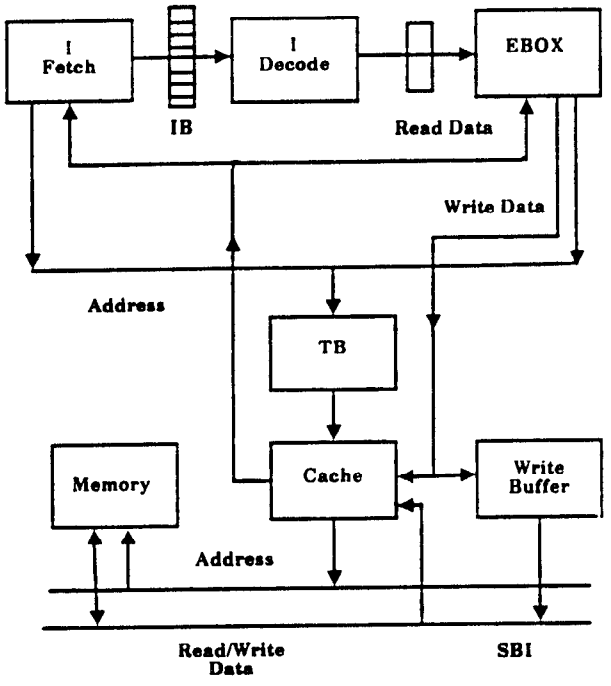
The 11/780 processor is composed of two major subsystems: the CPU pipeline, and the memory subsystem. These subsystems and their constituent components are illustrated in Figure 1. The CPU pipeline is responsible for most of the actual instruction execution, and as is shown, consists of three stages. The operation of the CPU pipeline may be most easily understood by noting that VAX instructions are composed of an opcode followed by zero to six *operand specifiers*, which describe the data operands required by the instruction. The 11/780 implementation of the VAX architecture breaks the execution of an instruction into a sequence of operations that correspond to the accessing of the data operands of the instruction and then its execution. In general these operations correspond to the tasks that flow down the CPU pipeline.

The individual stages of the CPU pipeline are: the *I-Fetch* stage, which sequentially fetches the instruction stream into the Instruction Buffer or IB; the *I-Decode* stage, which takes instruction bytes from the IB and decodes an opcode and/or specifier, determines a microcode dispatch address for the EBOX, and extracts additional specifier information that is used by the EBOX; and the *EBOX* stage, which is a microcoded function unit that does most of the actual work associated with fetching operands and executing instructions. In fact, the EBOX and the I-Decode stages are very tightly coupled, so that I-Decode operations only take place under specific control of the EBOX. The first I-Decode for an instruction cannot occur until the previous instruction has been competed, so the EBOX

0194-7111/84/0000/0301$01.00©1984 IEEE

FIGURE 1

VAX-11/780 Block Diagram



experiences a single non-overlapped I-Decode operation cycle for each instruction.

The EBOX can perform a number of autonomous operations, such as arithmetic and boolean computations; it can command the I-Fetch unit to start fetching at the target of a branch instruction; it can command reads and writes of memory data; and as a stage of the CPU pipeline, it can branch to a microinstruction location determined by the I-Decode stage. In this final instance it may have to wait as a result of a pipeline delay if the I-Decode stage has not yet been able to compute the desired location. We will call this delay an *IB stall.*

As the EBOX contains the microcode and does the majority of the instruction computation, we will be focusing mainly on its activity. We use the EBOX microinstruction time of 200 nanoseconds as the definition of a *cycle.*

In the process of instruction execution by the CPU pipeline, both the I-Fetch and EBOX stages may make references to memory. In order to support the virtual memory of the VAX these references must first pass through a *translation buffer*, or *TB*, where the virtual address generated by the CPU is translated into a physical address. A successful translation is called a TB *hit*, and conversly a failed translation is called a TB *miss.* In the event of a TB miss for an EBOX reference, a microcode interrupt is asserted and a microcode routine is invoked which inserts the desired translation into the TB. In the event of a TB miss for an I-Fetch reference, a flag is set; when the EBOX finds insufficient data bytes in the IB to do a desired decode, it recognizes that the flag is set and again goes about the task of putting the appropriate translation into the TB.

After successful translation by the TB, the physical address that was generated is used to access the *data cache.* Just as with the TB, we can have cache hits and misses. In the case of a read hit, data is simply passed back to the requesting unit. In the case of a read miss, a reference is made over the backplane bus, called the *SBI* for *Synchronous Backplane Interconnect*, to fetch the data from memory into the cache and to forward it to the requesting unit. During the time the data is being read from memory on behalf of an EBOX request the EBOX itself is *read stalled* waiting for the data, while during I-Fetch requests the EBOX is free to run unimpeded unless it too needs data from memory. A read operation which results in a hit in both the TB and cache consumes one cycle.

Only the EBOX is capable of doing data writes, and the 11/780 implements a *write-through* memory scheme in which all data writes are passed through to the memory via the SBI. Just as with reads, the TB is used to generate a physical address for the reference. In order to avoid waiting for the write to complete in memory the 11/780 provides a 4-byte *write buffer.* Thus it takes one cycle for the EBOX to initiate a write and then it continues microcode execution, which will be held up in the future only if another write request is made before the last one completed. The delay caused when a write encounters another write in progress is called a *write stall.* In addition, during a data write, the cache is accessed to update its contents with the data being written. Note, however, that if the write access misses, the cache is not updated.

2.2 Methods: Micro-PC Histogram Technique

Our measurements were collected with a special purpose hardware monitor that enabled us to create histograms of microcode execution in the 11/780 processor. This uPC monitor consists of a general purpose histogram count board, which has 16,000 addressable count locations (or histogram buckets), and is capable of incrementing the count in a selected location at the microcode execution rate of the 780. A processor-specific interface board was also required. It provided the address of a histogram count bucket and control lines to signal when a count should be made. For these experiments the interface board addressed a distinct histogram bucket for each microcode location in the processor's control store, and a count was taken for each microinstruction executed.

The histogram collection board was designed as a Unibus device, and Unibus commands can be used to start and stop data collection, as well as to clear and read the histogram count buckets. Coincidentally, since the 11/780 has a Unibus, the histogram collection monitor could be installed directly on the system being measured, obviating the cost and nuisance of using a second machine for the hardware monitor. This was a further convenience as the data

collected was immediately available on a machine of sufficient capacity to do the data reduction. Note, however, that while actually monitoring microcode execution, the data collection hardware is totally passive, causing no Unibus activity and having no effect on the execution of programs on the system. Thus this technique yields measurements of *all system activity at full speed.*

The capacity of the counters on the histogram collection board were sufficient to collect data for 1 to 2 hours of heavy processing on the CPU.

Since much of the activity in the 11/780 processor is under the direct command of microcode functions, the frequency of many events can be determined through examination of the relative execution counts of various microinstructions.The uPC histogram data is especially useful, since it forms a general resource from which the answers to many questions concerning the operation of the 11/780 running the same workload can be obtained simply by doing additional interpretation of the raw histogram data.

One disadvantage of this method of hardware monitoring lies in the fact that certain hardware events are not visible to the microcode. For example, the counts of instruction stream memory references are not available, because they are made by a distinct portion of the processor not under direct control of the microcode. Another is that to save microcode space, the microprogrammers frequently shared microinstructions; in such cases we cannot usually distinguish the sharers. A third disadvantage of this measurement technique is that the analysis produces only average behavior characterizations of the processor over the measurement interval, since no measures of the variation of the statistics during the measurement are collected.

The uPC histogram measurements were taken in two different experimental settings: live timesharing, and synthetic workloads. The live timesharing measurements were taken from two different machines within Digital engineering. The first machine belonged to the research group, and was used for general timesharing and some performance data analysis. Its workload consisted of such things as text-editing, program development, and electronic mail. It was relatively lightly loaded during the measurement interval, with approximately 15 users logged in.

The second timesharing measurements were taken from a machine being used by a group in the initial stages of development of a VAX CPU. The load on this machine consisted of the same type of general purpose timesharing as in the first experiment, with the addition of some circuit simulation and microcode development. This machine had a heavier load with approximately 30 users logged in during the measurement interval.

Although realistic, these live timesharing workloads are difficult to characterize and are not repeatable, since the computational load varies greatly over time. A second experimental setting addressed this problem. In it, a Remote Terminal Emulator or RTE [7, 14] provided a real-time simulation of a number of timesharing users connected to the VAX. The RTE is a PDP-11 with many asynchronous terminal interfaces; output characters generated by the RTE from canned user scripts are seen as terminal input characters by the VAX, and vice versa. Three RTE-generated workloads were measured: an educational environment, with 40 simulated users doing program development in various languages and some file manipulation; a scientific/engineering environment, with 40 simulated users doing scientific computation and program development; and a commercial transaction-processing environment, with 32 simulated users doing transactional database inquiries and updates.

All five experiments lasted about one hour. In this paper we will report results for the *composite* of all five, that is, the sum of the five uPC histograms.

The VMS operating system (version 2) [5, 9] was used in all our experiments. The VMS Null process, which runs when the system is idle, was excluded from measurement because its trivial code structure (branch to self, awaiting an interrupt) would bias all per-instruction statistics in proportion to the idleness of the system.

All of the VAXes had Floating Point Accelerators, and all had 8 Megabytes of memory.

3. ARCHITECTURAL EVENTS

An *architectural event* is an event that would occur in any implementation of the VAX architecture; an *implementation event* is one whose occurrence depends on the particular implementation of that architecture. Thus, for example, a data-stream memory read is usually an architectural event, but a consequent cache miss is an implementation event. We discuss the former here, and the latter in Section 4.

We will need to make certain assumptions about all VAX implementations for this distinction to be valid. We assume, for the purposes of our discussion, that:

- All VAX implementations have 32-bit data paths to the closest level of the memory hierarchy (usually the cache). Since the VAX is a 32-bit architecture, this is a very minor restriction. This allows us to count architectural memory references by measuring hardware references in the 11/780 implementation.
- All VAX implementations experience the same rate of operating system events. This allows us to treat instruction frequency as an architectural concern, ignoring the fact that an increased rate of, say, page faults would increase the frequency of instructions in the page fault routine.

3.1 Opcodes

VAX opcode frequency has been reported and discussed in other papers [4, 15]. The uPC method

cannot distinguish all opcodes in the 11/780. The predominant reason for this is that hardware is used for the implementation of some opcode-specific functions. For example, integer add and subtract instructions use the same microcode, with the ALU control field determined by hardware that looks at the opcode.

We can, however, report the frequency of *groups* of opcodes. Table 1 shows this for our composite workload. The following observation about this table is by now almost a cliché: moves, branches, and simple instructions account for most instruction executions. It will turn out, however, that some of the rarer, more complex instructions are responsible for a great deal of the memory references and processing time; this point has also been made before [12]. Note that VAX *subroutine* linkage is quite simple, involving only a push or pop of the PC together with a jump; *procedure* linkage is more complex, involving considerable state saving and restoring on the stack [6, 13].

A particularly interesting opcode-oriented performance measure is the frequency of PC-changing instructions and the proportion of conditional branches that actually do branch. In Table 2 below we show these figures for the composite workload. The upper section of the table consists of members of the SIMPLE group of Table 1. Because of microcode-sharing, two unconditional branches (BRB and BRW) are grouped with simple conditional branches. We believe from other measurements that these are about 2 percent of all instructions, leaving about 17 percent due to true conditional branches. The remaining rows are the PC-changing instructions from the FIELD, CALL/RET and SYSTEM instruction groups.

PC-changing instructions are quite common, accounting for almost 40 percent of all instructions executed in the composite workload. Furthermore, the proportion of these that actually change the PC is also quite high. Both properties are in line with other measurements of such instructions, both in the VAX and other architectures. Note that about 9 out of 10 loop branches actually branched. Therefore the average number of iterations of all loops that used these instructions was about 10.

3.2 Operand Specifiers

VAX instructions specify the location of their data through one or more encoded *operand specifiers* that follow the opcode in the I-stream. These indicate, for example, whether a read operand is to be found in a register, or in memory addressed by a register, or with a variety of other addressing modes [6, 13]. The *data type* (byte, longword, floating-point, etc.) and *access mode* (read, modify, write, etc.) of an operand specifier are defined by the instruction that uses it. Branch displacements are considered separately.

In the 11/780 microcode, all access to scalar data, and to the addresses of non-scalar data, are done by specifier microcode. We thus consider the reading and writing of scalar data, and the address

TABLE 1

Opcode Group Frequency

Group name	Constituents	Frequency (Percent)
SIMPLE	Move instructions Simple arith. operations Boolean operations Simple and loop branches Subroutine call and return	83.60
FIELD	Bit field operations	6.92
FLOAT	Floating point Integer multiply/divide	3.62
CALL/RET	Procedure call and return Multi-register push and pop	3.22
SYSTEM	Privileged operations Context switch instructions Sys. serv. requests and return Queue manipulation Protection probe instructions	2.11
CHARACTER	Char. string instructions	0.43
DECIMAL	Decimal instructions	0.03

TABLE 2

PC-Changing Instructions

Branch Type	Percent of Inst.	Percent that branch	Act. branch as percent of all inst.
Simple cond., plus BRB, BRW	19.3	56	10.9
Loop branches	4.1	91	3.7
Low-bit tests	2.0	41	0.8
Subroutine call and return	4.5	100	4.5
Unconditional (JMP)	0.3	100	0.3
Case branch (CASEx)	0.9	100	0.9
Bit branches	4.3	44	1.9
Procedure call and return	2.4	100	2.4
System branches (CHMx, REI)	0.4	100	0.4
TOTAL	38.5	67	25.7

calculation of non-scalar data, to be associated with operand specifier processing and not with the instruction itself. A simple integer Move, for example, is accomplished entirely by specifier microcode: first a read, then a write.

The 11/780 specifier-processing microcode allows us to distinguish first specifiers, called *SPEC1* (those that directly follow the opcode) from all other specifiers, called *SPEC2-6*. It also lets us count PC-relative *branch displacements*, which appear in the last specifier position of certain PC-changing instructions. Not all PC-changing instructions use branch displacements: some determine their targets with ordinary operand specifiers (e.g., JMP, CALLS), while others determine their targets implicitly (e.g., RSB, RET, REI).

Table 3 shows the number of specifiers and branch displacements per average VAX instruction.

Table 4 shows the frequency of operand specifier types. Because of microcode-sharing, we are able to report the individual frequencies of the various types of memory-referencing specifiers only in the total column. Memory-referencing specifiers can optionally be indexed: the percentage of all specifiers that are indexed is shown in the bottom line of the table.

Register mode is the most common addressing mode, especially in specifiers after the first. Since the last specifier is generally the destination of the instruction's result (if not a branch), this probably reflects a tendency to store results in registers. The encoded short literal, in which a single byte is expanded to one of a small number of values whose data type is instruction-dependent, is also quite common, particularly as the first specifier. We note the scarcity of *immediate* data ((PC)+), the other method of supplying I-stream constants to the instruction. Short literals apparently do this job fairly well.

The most common memory specifier is displacement off a register. Other results [15] suggest that the displacement is most often a byte, less often a 4-byte longword, and least often a word. Index mode is surprisingly common; 6.3 percent of all specifiers were indexed.

The average number of specifiers per instruction in the composite workload is 1.48 (remember that this does not include branch displacements).

3.3 Memory Operations

3.3.1 Data

Operand-specifier processing accounts for a majority of the D-stream memory operations performed on the VAX. Most other reads and writes are due to the manipulation of non-scalar data such as character strings and stack frames. Table 5 reports the frequency of read and write operations per average instruction, broken down by the source of the operation. After specifiers, procedure call and return instructions, which push and pop registers on and off the stack, account for the greatest portion of reads and writes.

Because the results are in terms of events *per average instruction*, the number of reads reported for the CALL/RET group, for example, is *not* the average number of reads executed by the average CALL/RET instruction. Rather, it is the number of CALL/RET reads averaged over *all* instruction executions. Put another way, it is the number of CALL/RET reads weighted by the frequency of occurence of instructions in the CALL/RET group. This way of looking at the data directly measures the contribution of the various instruction groups to overall performance.

Overall, the ratio of reads to writes is about two to one. Some of these references are to 32-bit longwords that are *unaligned* with respect to the physical organization of the cache, and that therefore require two physical references. The frequency of

TABLE 3

Specifiers and Branch Displacements per Average Instruction

First specifiers	0.726
Other specifiers	0.758
Branch displacements	0.312

TABLE 4

Operand specifier distribution (percent)

		SPEC1	SPEC2-6	Total
Register	R	28.7	52.6	41.0
Short Literal	#n	21.1	10.8	15.8
Immediate	(PC)+	3.2	1.7	2.4
Displacement	D(R)			25.0
Reg. Deferred	(R)			9.2
Auto-inc.	(R)+			2.1
Disp. Deferred	@D(R)	47.0	34.9	2.7
Absolute	@(PC)+			0.6
Auto-inc.def.	@(R)+			0.3
Auto-dec.	-(R)			0.9
Percent Indexed	[R]	8.5	4.2	6.3

TABLE 5

D-stream Reads and Writes per Average Instruction

	Reads	Writes
Spec1	.306	.000
Spec2-6	.148	.161
Simple	.029	.033
Field	.049	.007
Float	.000	.008
Call/Ret	.133	.130
System	.015	.014
Character	.039	.046
Decimal	.002	.001
Other	.062	.008
TOTAL	.783	.409

unaligned D-stream references is very low: 0.016 per instruction in the composite workload.

3.3.2 Instructions

Many memory reads are due to instruction fetching, but it is difficult to characterize this in a strictly architectural way. Different organizations of the I-stream prefetching hardware can have very different streams of references to memory. The only truly architectural feature of the I-stream references is the *size* of the instructions. The average size of an operand specifier can be calculated from Table 3, together with displacement figures (byte, word, longword) from [15], and is 1.68 bytes. The average instruction has one byte of opcode, some number of specifiers, and some fractional number of branch displacements. Table 6 puts all of this together to show that the average size of a VAX instruction in our workload was 3.8 bytes.

3.4 Other Events

Two other interesting architectural events are interrupts and context switches. The latter are accomplished by the save-process-context and load-process-context instructions (SVPCTX and LDPCTX). In VMS these are used only for a switch from one user process to another; interrupts, in particular, do not cause context switches. The frequency of these events is shown in Table 7. For ease of understanding we invert our usual metric and report these in terms of the average instruction *headway* between events. VMS sometimes services hardware interrupts by chaining together several successively lower-priority software interrupts. Table 7 includes the headway between requests for software interrupts.

The context-switch figure is useful in setting the "flush" interval in cache and translation buffer

TABLE 6

Estimated Size of Average Instruction

Object	Number per inst	Est. Size	Est. Size per inst.
Opcode	1.00	1.00	1.00
Specifiers	1.48	1.68	2.49
Branch disp.	0.31	1.00	0.31
TOTAL			3.8

TABLE 7

Interrupt and Context-Switch Headway

Event	Instruction headway
Software Interrupt Requests	2539
Hardware and Software Interrupts	637
Context Switches	6418

simulations. The impact of context switching on VAX Translation Buffer performance is discussed in [3].

4. IMPLEMENTATION EVENTS

By an implementation event we mean an event whose occurrence depends on the particular implementation of the VAX architecture. A cache miss is an example; whether a memory reference hits or misses in the cache depends on the size and configuration--indeed, even the *presence*--of the cache in a particular implementation of the architecture.

4.1 I-stream References

The 11/780's Instruction Buffer or IB makes its I-stream referencing behavior implementation-specific. The 8-byte IB makes a cache reference whenever one or more bytes are empty. When the requested longword arrives possibly much later, if there was a cache miss the IB accepts as many bytes as it has room for then. Thus the IB can make repeated references (as many as four) to the same longword, but this is clearly not a requirement of the architecture.

Because the IB is controlled by hardware, the uPC histogram technique cannot count IB references. But in our earlier cache study [2] we found that the average number of cache references by the IB per VAX instruction was around 2.2, for three day-long measurements of live timesharing workloads.

Since the average VAX instruction is 3.8 bytes long (Table 6), we conclude that those 2.2 references yielded on average 3.8 bytes, for an average delivery *per reference* of 1.7 bytes.

4.2 Cache And Translation Buffer Misses

The 11/780 cache is controlled by hardware, so the frequency of cache misses is not measurable with the uPC technique. Our earlier cache study, however, found that in live timesharing workloads the number of cache read misses per instruction was 0.28, with 0.18 due to the I-stream and 0.10 due to the D-stream. The performance cost of these misses is microcode *stalls*, which are discussed below.

The virtual-to-physical address Translation Buffer, on the other hand, is controlled by microcode, and is therefore directly visible with the uPC technique. A TB miss results in a microcode trap to a miss service micro-routine. Entries to this routine indicate occurrences of TB misses, and a count of all cycles within the routine yields the time spent handling TB misses.

The TB miss rate for the composite workload was 0.029 misses per instruction, 0.020 from the D-stream and 0.009 from the I-stream. The average number of cycles used to service a miss was 21.6, of which 3.5 were read stalls due to the requested page-table entry not being in the cache. See [3] for more information on the performance of the VAX-11/780 TB.

4.3 Stalls

A *stall* occurs when a microcode request cannot yet be satisfied by the hardware. The result is one or more cycles of suspended execution until the reason for the stall goes away. As described in Section 2.1, there are three types of stall in the VAX-11/780: read stall, write stall, and IB stall.

A read stall occurs when there is a cache miss on a D-stream read. The requesting microinstruction simply waits for the data to arrive. In the simplest case (no concurrent memory activity of other types) this takes 6 cycles on the 11/780. Cache hits cause no stalls.

A write will stall if attempted less than 6 cycles after the previous write (in the simplest case). VAX instructions that do many writes, such as character-string moves, are sometimes microprogrammed to reduce write stalls by writing only in every sixth cycle.

The last type of stall, IB stall, occurs when the IB does not contain enough bytes to satisfy the microcode's request. This can occur at any point in I-stream processing, including the initial decode of the opcode, specifier decodes, and requests for literal or immediate data. Note that IB stall does not occur in direct response to an IB cache miss; only when the empty byte is actually needed by the microcode can stall occur, and by then the cache miss may have finished.

The occurrence and duration of all three types of stalls are implementation-specific characteristics of the VAX-11/780. The duration, but not the frequency of occurrence of all three can be measured with the uPC technique. The histogram board actually contains two sets of counts, one for non-stalled microinstructions, and one for read- or write-stalled microinstructions. If the microinstruction at address X does a cache read, then the non-stalled count at location X will contain the actual number of successful reads done by that microinstruction, while the stalled count at location X will contain the total number of cycles in which that microinstruction was stalled. Write stalls and read stalls are differentiated by whether the microinstruction does a read or a write (it cannot do both).

IB stalls are handled in a slightly different way. Requests for bytes from the IB result in microcode dispatches; decoding hardware maps the IB contents into various dispatch microaddresses, one of which indicates that there were insufficient bytes in the IB. The number of executions of the microinstruction at that microaddress is the number of cycles with IB stall.

5. TIME: CYCLES PER INSTRUCTION

The great strength of the uPC histogram technique is its ability to classify every processor cycle and thus to establish the durations of processor events. Table 8 shows the number of cycles per average instruction, arranged in two orthogonal dimensions. The first dimension (rows) represents the stages of an instruction's execution: its initial Decode; then its operand specifier and branch displacement processing; then its execute phase; and finally several *overhead* activities.

Instruction decode, as discussed in Section 2.1 above, takes exactly one EBOX cycle, but may stall if there are insufficient bytes in the IB.

Operand specifier processing consists of address calculation for memory specifiers, and the actual read and/or write of data for both memory and register specifiers, provided the data is scalar. Branch displacement processing consists of the calculation of the branch target address, which requires one cycle. An additional cycle is consumed in the execute phase of the instruction to redirect the IB to fetch down the target stream.

The execute phase of an instruction consists of those microcycles associated with an instruction's actual computation. Table 8 reports these results by opcode group as defined in Table 1.

The overhead activities are not associated with any particular instruction. They include interrupts and exceptions (Int/Except), memory management and alignment microcode (Mem Mgmt), and abort cycles (one for each microcode trap and one for each microcode patch).

The second dimension of Table 8 (columns) classifies microinstruction execution into one of six

TABLE 8

Average VAX Instruction Timing (Cycles per Instruction)

	Compute	Read	R-Stall	Write	W-Stall	IB-Stall	Total
Decode	1.000					0.613	1.613
Spec1	0.895	0.306	0.364				1.565
Spec2-6	1.052	0.148	0.116	0.161	0.192	0.102	1.771
B-Disp	0.221					0.005	0.226
Simple	0.870	0.029	0.017	0.033	0.027		0.977
Field	0.482	0.049	0.058	0.007	0.002		0.600
Float	0.292	0.000	0.000	0.008	0.001		0.302
Call/Ret	0.937	0.133	0.074	0.130	0.184		1.458
System	0.434	0.015	0.031	0.014	0.028		0.522
Character	0.318	0.039	0.099	0.046	0.004		0.506
Decimal	0.026	0.002	0.000	0.001	0.002		0.031
Int/Except	0.055	0.002	0.005	0.004	0.006		0.071
Mem Mngmt	0.555	0.061	0.200	0.004	0.003		0.824
Abort	0.127						0.127
TOTAL	7.267	0.783	0.964	0.409	0.450	0.720	10.593

categories. The "Compute" category represents autonomous EBOX operations, that is, microinstructions that do no memory references. The other categories are memory references and the various types of stall. On the 11/780 the six categories are mutually exclusive, so times in the individual categories can be summed, yielding the TOTAL column of Table 8.

With some minor exceptions† every microcycle in 11/780 execution falls into exactly one row and exactly one column. The numbers reported in Table 8 are the numbers of cycles spent at each row/column intersection, divided by the number of VAX instructions executed. They are therefore the numbers of cycles per average instruction for each category. The row and column totals allow analysis of a single dimension: for example, in the average instruction of 10.6 cycles, a (column) total of 0.96 cycles were lost in read stall, and a (row) total of 0.30 cycles were spent in floating-point execution.

Table 8 shows where 11/780 performance may be improved, and where it may not be improved. For example, saving the non-overlapped Decode cycle could save one cycle on each non-PC-changing instruction. (The later VAX model 11/750 did this.) On the other hand, optimizing FIELD memory writes will have a payoff of at most 0.007 cycles per instruction, or only about 0.07 percent of total performance.

A number of other observations can be made based on Table 8:

- The average VAX instruction in this composite workload takes a little more than 10 cycles. This makes the numbers in Table 8 easily intepretable as percentages of the total time per instruction.
- The TOTAL column shows that almost half of all the time went into decode and specifier processing, counting their stalls.
- The opcode group with the greatest contribution is the CALL/RET group, despite its low frequency (see Table 1).
- The execution phase of the SIMPLE instructions, which constitute 84 percent of all instruction executions (Table 1), accounts for only about 10 percent of the time in the composite workload.
- System and Character instructions, though rare (Table 1), also make noticeable contributions to performance.
- Most IB stalls occur on the initial specifier decode, rather than on subsequent specifier decodes. Although there are more bytes in the initial decode then the subsequent decodes, we interpret this to mean that most IB stall is incurred on cache misses at the target reference of a branch.
- We note that there are fewer cycles of compute in B-DISP than there are branch displacements (see Table 3), because the branch displacement need

†Two remarks on the operand-specifier portion of Table 8 are necessary. First, the 11/780 has special hardware to optimize the execution of certain instructions with literal or register operands. In these cases the first cycle of execution is combined with the last cycle of specifier processing. We report such cycles in the specifier rows of Table 8; they amounted to 0.15 cycles per instruction for the SIMPLE group and 0.01 cycles per instruction for the FIELD group. The second remark concerns the treatment of first specifiers that are indexed. Microcode sharing forces use to report the calculation of the base address in the SPEC2-6 category. We extimate that this causes about 0.06 cycles per instruction belonging to SPEC1 to be reported in SPEC2-6.

TABLE 9

Cycles per instruction Within Each Group

	Compute	Read	R-Stall	Write	W-Stall	Total
Simple	1.04	0.03	0.02	0.04	0.03	1.17
Field	6.97	0.71	0.85	0.11	0.04	8.67
Float	8.07	0.00	0.00	0.23	0.03	8.33
Call/Ret	29.08	4.14	2.29	4.03	5.71	45.25
System	20.59	0.71	1.47	0.67	1.30	24.74
Character	73.51	8.97	22.83	10.76	0.97	117.04
Decimal	84.37	5.64	1.59	3.94	5.24	100.77

not be computed when the instruction does not branch.

A comparison of the Read and Read-Stall columns of Table 8 yields another set of observations:

- Stalled cycles are half the number of operation cycles in the CALL/RET group, but more than twice the number of operation cycles in the Character group. This is presumably due to the good cache locality of the stack and the relatively poor locality of character strings.
- Memory management has more than 3 times as many read-stalled cycles as reads. This largely reflects the tendency of references to Page Table Entries to miss in the cache.

Comparing Write and Write-stall columns yields several more observations:

- The CALL/RET group generates a large amount of write stalls. This is due to the write-through cache and the one-longword write butter, which force the CALL instruction to stall while pushing the caller's state onto the stack.
- Character instructions have little write stall, because as mentioned earlier, the microcode was explictly written to avoid write stalls.

Table 9 shows the number of cycles per average instruction *within each group*, exclusive of specifier decode and processing, and *not* weighted by frequency of occurence. For example, the average instruction in the Decimal group did 84 cycles of Compute and took 101 cycles overall.

Table 9 illustrates a number of interesting properties:

- The computation associated with the average simple instruction is quite simple: a little over one cycle is all that it needs.
- However, the range of cycle time requirements of average representatives of these groups covers two orders of magnitude.
- With around 4 reads and writes per average CALL/RET or PUSHR/POPR instruction we conclude that about 8 registers are being pushed and popped.
- The average character instruction reads and writes 9 to 11 longwords, so the average size of a character string is 36-44 characters.

6. CONCLUSION

We have presented detailed instruction timing results for the VAX-11/780, evaluated under a timesharing workload. These results are, of course, dependent on the characteristics of that workload.

The uPC histogram method has provided a great deal of useful data, showing precisely the impact of architectural and implementation characteristics on average processor performance. The generation of a uPC histogram provides the analyst with a database from which many performance characteristics can be determined. These analyses are particularly useful because they are all derived from the same workload.

ACKNOWLEDGMENTS

We would like to thank Garth Wiebe and Jean Hsiao for their assistance with the uPC histogram monitor development.

REFERENCES

[1] Alpert, D. Carberry, D., Yamamura, M.,Chow, Y., and Mak, P32-bit Processor Chip Integrates Major System Functions. *Electronics 56*, 14 (July 14, 1983), pp. 113-119.

[2] Clark, D.W. Cache Performance in the VAX-11/780. *ACM TOCS 1*, 1 (Feb. 1983), pp. 24-37.

[3] Clark, D.W. and Emer, J.S. Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement. Submitted for publication, Nov. 1983.

[4] Clark, D.W. and Levy, H.M., Measurement and Analysis of Instruction Use in the VAX-11/780. *Proc. 9th Annual Symp. on Comp. Arch.*, Austin, April 1982, pp. 9-17.

[5] Digital Equipment Corp. VAX/VMS Internals and Data Structures. Document No. AA-K785A-TE, Digital Equipment Corp., Maynard, MA.

[6] Digital Equipment Corp. *VAX-11 Architecture Reference Manual.* Document No. EK-VAXAR-RM-001, Digital Euipment Corp., Maynard, MA, May 1982.

[7] Greenbaum, H.J. A Simulator of Multiple Interactive Users to Drive a Time-Shared Computer System. M.S. Thesis, MIT Project MAC report MAR-TR-54, Oct.1968.

[8] Huck, J.C. *Comparative Analysis of Computer Architectures.* Ph.D. thesis, TR No. 83-243, Computer Systems Lab., Stanford, May 1983.

[9] Levy, H.M., and Eckhouse, R.H. Computer Programming and Architecture: The VAX-11. Digital Press, Bedford, MA, 1980.

[10] Lunde, A. Empirical Evaluation of Some Features of Instruction Set Processor Architectures. *CACM 20*, 3 (March 1977), 143-153.

[11] McDaniel, G. An Analysis of a Mesa Instruction Set Using Dynamic Instruction Frequencies. *Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, pp. 167-176.

[12] Peuto, B.L., and Shustek, L.J. An Instruction Timing Model of CPU Performance. *Proc. 4th Annual Symp. on Computer Architecture*, 1977, pp. 165-178.

[13] Strecker, W.D., VAX-11/780--A Virtual Address Extension for the PDP-11 Family Computers. *Proc. NCC*, AFIPS Press, Montvale, N.J., 1978.

[14] Watkins, S.W., and Abrams, M.D. Survey of Remote Terminal Emulators. NBS Special Publication 500-4, April 1977.

[15] Wiecek, C.A. A Case Study of VAX-11 Instruction Set Usage for Compiler Execution. *Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, pp. 177-184.

[16] Winder,. R.O. A Data Base for Computer Performance Evaluation. IEEE Computer 6, 3. (March 1973), pp. 25-29.