VLSI Processor Architecture

JOHN L. HENNESSY

Abstract — **A processor architecture attempts to compromise between the needs of programs hosted on the architecture and the performance attainable in implementing the architecture. The needs of programs are most accurately reflected by the dynamic use of the instruction set as the target for a high level language compiler. In VLSI, the issue of implementation of an instruction set architecture is significant in determining the features of the architecture. Recent processor architectures have focused on two major trends: large microcoded instruction sets and simplified, or reduced, instruction sets. The attractiveness of these two approaches is affected by the choice of a single-chip implementation. The two different styles require different tradeoffs to attain an implementation in silicon with a reasonable area. The two styles consume the chip area for different purposes, thus achieving performance by different strategies. In a VLSI implementation of an architecture, many problems can arise from the base technology and its limitations. Although circuit design techniques can help alleviate many of these problems, the architects must be aware of these limitations and understand their implications at the instruction set level.**

Index Terms — **Computer organization, instruction issue, instruction set design, memory mapping, microprocessors, pipelining, processor architecture, processor implementation, VLSI.**

I. Introduction

ADVANCES in semiconductor fabrication capabilities have made it possible to design and fabricate chips with tens of thousands to hundreds of thousands of transistors, operating at clock speeds as fast as 16 MHz. Single-chip processors that have transistor complexity and performance comparable to CPU's found in medium- to large-scale mainframes can be designed. Indeed, both commercial and experimental nMOS processors have been built that match the performance of large minicomputers, such as DEC's VAX 11/780.

In the context of this paper, a processor architecture is defined by the view of the programmer; this view includes user visible registers, data types and their formats, and the instruction set. The memory system and I/O system architectures may be defined either on or off the chip. Because we are concerned with chip-level processors we must also include the definition of the interface between the chip and its environment. The chip interface defines the use of individual pins, the bus protocols, and the memory architecture and I/O architecture to the extent that these architectures are controlled by the processor's external interface.

Manuscript received April 30, 1984; revised July 31, 1984. This work was supported by the Defense Advanced Research Projects Agency under Grants MDA903-79-C-680 and MDA903-83-C-0335.

The author is with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305.

In many ways, the architecture and organization of these VLSI processors is similar to the designs used in the CPU's of modern machines implemented using standard parts and bipolar technology. However, the tremendous potential of MOS technology has not only made VLSI an attractive implementation medium, but it has also encouraged the use of the technology for new experimental architectures. These new architectures display some interesting concepts both in how they utilize the technology and in how they overcome performance limitations that arise both from the technology and from the standard barriers to high performance encountered in any CPU.

This paper investigates the architectural design of VLSI uniprocessors. We divide the discussion into six major segments. First, we examine the goals of a processor architecture; these goals establish a framework for examining various architectural approaches. In the second section, we explore the two major styles: reduced instruction set architectures and high level microcoded instruction set architectures. Some specific techniques for supporting both high level languages and operating systems functions are discussed in the third and fourth sections, respectively. The fifth section of the paper surveys several major processor architectures and their implementations; we concentrate on showing the salient features that make the processors unique. In the sixth section we investigate an all-important issue — implementation. In VLSI, the organization and implementation of a CPU significantly affect the architecture. Using some examples, we show how these features interact with each other, and we indicate some of the principles involved.

II. Architectural Goals

A computer architecture is measured by its effectiveness as a host for applications and by the performance levels obtainable by implementations of the architecture. The applications are written in high level languages, translated to the processor's instruction set by a compiler, and executed on the processor using support functions provided by the operating system. Thus, the suitability of an architecture as a host is determined by two factors: its effectiveness in supporting high level languages, and the base it provides for system level functions. The efficiency of an architecture from an implementation viewpoint must be evaluated both on the cost and on the performance of implementations of that architecture. Since a computer's role as program host is so important, the

0018-9340/84/1200-1221$01.00

instruction set designer must carefully consider both the usefulness of the instruction set for encoding programs and the performance of implementations of that instruction set.

Although the instruction set design may have several goals, the most obvious and usually most important goal is *performance*. Performance can be measured in many ways; typical measurements include instructions per second, total required memory bandwidth, and instructions needed both statically and dynamically for an application. Although all these measurements have their place, they can also be misleading. They either measure an irrevelant point, or they assume that the implementation and the architecture are independent.

The key to performance is the ability of the architecture to execute high level language programs. Measures based on assembly language performance are much less useful because such measurements may not reflect the same patterns of instruction set usage as compiled code. Of course, compiler interaction clouds the issue of high level language performance; that is to be expected. The architecture also influences the ease and difficulty of building compilers.

Implementation related effects can cause serious problems if the abstract measurements are used as a gauge of the real hardware performance. The architecture profoundly influences the complexity, cost, and potential performance of the implementation. On the basis of abstract architecturally oriented benchmarks, the most complex, highest level instruction sets seem to make the most sense; these include machines like the VAX [1], the Intel-432 [2], the DEL approaches [3], and the Xerox Mesa architectures [4]. However, the cost of implementing such architectures is higher, and their performance is not necessarily as good as architectural measures, such as instructions executed per high level statement, might indicate. Many VAX benchmarks show impressive architectural measurements, especially for instruction bytes fetched. However, data from implementations of the architecture show that the same performance is not attained. VAX instructions are short; the instruction fetch unit must constantly prefetch instructions to keep the rest of the machine busy. This includes fetching one or more instructions that sequentially follow a branch. Since branches are frequent and they are taken with higher than 50 percent probability, the instructions fetched following a branch are most often not executed. This leads to a significantly higher instruction bandwidth than the architectural measurements indicate.

Since most programs are written in high level languages, the role of the architecture as a host for programs depends on its ability to serve as a target for the code generated by compilers for high level languages of interest. The effectiveness is a function of the architecture, the compiler technology, and, to a lesser extent, the programming language. Much commonality exists among languages in their need for hardware support; furthermore, compilers tend to translate common features to similar types of code sequences. Some special language features may be significant enough to influence the architecture. Examples of such of features are support for tags, support for floating point arithmetic, and support for parallel constructs.

Program optimization is becoming a standard part of many compilers. Thus, the architecture should be designed to support the code produced by an optimizing compiler. An implication of this observation is that the architecture should expose the details of the hardware to allow the compiler to maximize the efficiency of its use of that hardware. The compiler should also be able to compare alternative instruction sequences and choose the more time or space efficient sequence. Unless the execution implications of each machine instruction are visible, the compiler cannot make a reasonable choice between two alternatives. Likewise, hidden computations cannot be optimized away. This view of the optimizing compiler argues for a simplified instruction set that maximizes the visibility of all operations needed to execute the program.

Large instruction set architectures are usually implemented with microcode. In VLSI, silicon area limitations often force the use of microcode for all but the smallest and simplest instruction sets: all of the commercial 16 and 32 bit processors make extensive use of microcode in their implementations. In a processor that is microcoded, an additional level of translation, from the machine code to microinstructions, is done by the hardware. By allowing the compiler to implement this level of translation, the cost of the translation is taken once at compile-time rather than repetitively every time a machine instruction is executed. The view of an optimizing compiler as generating microcode for a simplified instruction set is explained in depth in a paper by Hopkins [5]. In addition to eliminating a level of translation, the compiler "customizes" the generated code to fit the application [6]. This customizing by the compiler can be thought of as a realizable approach to dynamically microcoding the architecture. Both the IBM 801 and MIPS exploit this approach by "compiling down" to a low level instruction set.

The architecture and its strength as a compiler target determine much of the performance at the architectural level. However, to make the hardware usable an operating system must be created on the hardware. The operating system requires certain architectural capabilities to achieve full functional performance with reasonable efficiency. If the necessary features are missing, the operating system will be forced to forego some of its user-level functions, or accept significant performance penalities. Among the features considered necessary in the construction of modern operating systems are

- privileged and user modes, with protection of specialized machine instructions and of system resources in user mode;
- support for external interrupts and internal traps;
- memory mapping support, including support for demand paging, and provision for memory protection; and
- support for synchronization primitives, in multiprocessor configurations, if conventional instructions cannot be used for that purpose.

Some architectures provide additional instructions for supporting the operating system. These instructions are included for two primary reasons. First, they establish a standard interface for hardware dependent functions. Second, they may enhance the performance of the operating system by supporting some special operation in the architecture.

Standardizing an interface by including it in the architec-

ture has been cited as a goal both for conventional high level instructions, e.g., on the VAX [7], and for operating system interfaces [2]. Standardizing an interface in the architectural specification can be more definitive, but it can carry performance penalties when compared to a standard at the assembly language level. This standard can be implemented by macros, or by standard libraries. Putting the interface into the architecture commits the hardware designers to supporting it, but it does not inherently enforce or solidify the interface.

Enhancing operating system performance via the architecture can be beneficial. However, such enhancements must be compared to alternative improvements that will increase general performance. Even when significant time is spent in the operating system, the bulk of the time is spent executing general code rather than special functions, which might be supported in the architecture. The architect must carefully weigh the proposed feature to determine how it affects other components of the instruction set (overhead costs, etc.), as well as the opportunity cost related to the components of the instruction set that could have been included instead. Many times the performance gained by such high level features is small because the feature is not heavily used or because it yields only a minor improvement over the same function implemented with a sequence of other instructions. Often the combination of a feature's cost and performance merit forms a strong argument against its presence in the architecture.

Hardware organization can dramatically affect performance. This is especially true when the implementation is in VLSI where the interaction of the architecture and its implementation is more pronounced. Some of the more important architectural implications are as follows.

- The limited speed of the technology encourages the use of parallel implementations. That is, many slower components are used rather than a smaller number of fast components. This basic method has been used by many designers on projects as varied as systolic arrays [8] to the MicroVAX I datapath chip [9].
- The cost of complexity in the architecture. This is true in any implementation medium, but is exacerbated in VLSI, where complexity becomes more difficult to accommodate. A corollary of this rule is that no architectural feature is free.
- Communication is more expensive than computation. Architectures that require significant amounts of global interaction will suffer in implementation.
- The chip boundaries have two major effects. First, they impose hard limits on data bandwidth on and off the chip. Second, they create a substantial disparity between on-chip and off-chip communication delays.

The architecture affects the performance of the hardware primarily at the organizational level where it imposes certain requirements. Smaller effects occur at the implementation level where the technology and its properties become relevant. The technology acts strongly as a weighting factor favoring some organizational approaches and penalizing others. For example, VLSI technology typically makes the use of memory on the chip attractive: relatively high densities can be obtained and chip crossings can be eliminated.

A goal in implementation is to provide the fastest hardware possible; this translates into two rules.

1) Minimize the clock cycle of the system. This implies both reducing the overhead on instructions as well as organizing the hardware to minimize the delays in each clock cycle.

2) Minimize the number of cycles to perform each instruction. This minimization must be based on the expected dynamic frequency of instruction use. Of course, different programming languages may differ in their frequency of instruction usage.

This second rule may dictate sacrificing performance in some components of the architecture in return for increased performance of the more heavily used parts.

The observation that these types of tradeoffs are needed, together with the fact that larger architectures generate additional overhead, have led to the reduced (or simplified) instruction set approach [10], [11]. Such architectures are streamlined to eliminate instructions that occur with low frequency in favor of building such complex instructions out of sequences of simpler instructions. The overhead per instruction can be significantly reduced and the implementor does not have to discriminate among the instructions in the architecture. In fact, most simplified instruction set machines use single cycle execution of every instruction; this eliminates complex tradeoffs both by the hardware implementor and the compiler writer. The simple instruction set permits a high clock speed for the instruction execution, and the one-cycle nature of the instructions simplifies the control of the machine. The simplification of control allows the implementation to more easily take advantage of parallelism through pipelining. The pipeline allows simultaneous execution of several instructions, similar to the parallel activity that would occur in executing microinstructions for the interpretation of a more complex instruction set.

III. Basic Architectural Trends

The major trend that has emerged among computer architectures in the recent past has been the emphasis on targeting to and support for high level languages. This trend is especially noticeable within the microprocessor area where it represents an abrupt change from the assembly-language-oriented architectures of the 1970's. The most recent generation of commercially available processors, the Motorola 68000, the Intel 80X86, Intel iAPX-432, the Zilog 8000, and the National 16032, clearly show the shift from the 8-bit assembly language oriented machines to the 16-bit compiled language orientation. The extent of this change is influenced by the degree of compatibility with previous processor design. The machines that are more compatible (the Intel 80X86 and the Zilog processors) show their heritage and the compatibility has an effect on the entire instruction set. The Motorola and National products show much less compatibility and more of a compiled language direction.

This trend is more obvious among designs done in universities. The Mead and Conway [12] structured design approach has made it possible to design VLSI processors within the university environment. These projects have been language-directed. The RISC project at Berkeley and the MIPS project at Stanford both aim to support high level

languages with simplified instruction sets. The MIT Scheme project [13] supports LISP via a built-in interpreter for the language.

A. RISC-Style Machines

A RISC, reduced instruction set computer, is a machine with simplified instruction set. The architectures that are generally considered to be RISC's are the Berkeley RISC I and II processors, the Stanford MIPS processor, and the IBM 801 processor (which is *not* a microprocessor). These machines certainly have instruction sets that are simpler than most other machines; however, they may still have many instructions: the 801 has over 100 instructions, MIPS has over 60. They may also have conceptually complex details: the 801 has instructions for programmer cache management, while MIPS requires that pipeline dependence hazards be removed in software. All three architectures avoid features that require complex control structures, though they may use a complex implementation structure where the complexity is merited by the performance gained.

The adjective *streamlined* is probably a better description of the key characteristics of such architectures. The most important features are

1) regularity and simplicity in the instruction set allows the use of the same, simple hardware units in a common fashion to execute almost all instructions;

2) single cycle execution — most instructions execute in one machine (or pipeline) cycle. These architectures are register-oriented: all operations on data objects are done in the registers. Only load and store instructions access memory; and

3) fixed length instructions with a small variety of formats.

The advantages of streamlined instruction set architectures come from a close interaction between architecture and implementation. The simplicity of the architecture lends a simplicity to the implementation. The advantages gained from this include the following.

1) The simplified instruction formats allow very fast instruction decoding. This can be used to reduce the pipeline length (without reducing throughput), and/or shorten the instruction execution time.

2) Most instructions can be made to execute in a single cycle; the register-oriented (or load/store) nature of the architecture provides this capability.

3) The simplicity of the architecture means that the organization can be streamlined; the overhead on each instruction can be reduced, allowing the clock cycle to be shortened.

4) The simpler design allows silicon resources and human resources to be concentrated on features that enhance performance. These may be features that provide additional high level language performance, or resources may be concentrated on enhancing the throughput of the implementation.

5) The low level instruction set provides the best target for state-of-the-art optimizing compiler technology. Nearly every transformation done by the optimizer on the intermediate form will result in an improved running time because the transformation will eliminate one or more instructions. The benefits of register allocation are also enhanced by eliminating entire instructions needed to access memory.

6) The simplified instruction set provides an opportunity to eliminate a level of translation at runtime, in favor of translating at compile-time. The microcode of a complex instruction set is replaced by the compiler's code generation function.

The potential disadvantages of the streamlined architecture come from two areas: memory bandwidth and additional software requirements. Because a simplified instruction set will require more instructions to perform the same function, instruction memory bandwidth requirements are potentially higher than for a machine with more powerful and more tightly encoded instructions. Some of this disadvantage is mitigated by the fact that instruction fetching will be more complicated when the architecture allows multiple sizes of instructions, especially if the instructions require multiple fetches due to lack of alignment or instruction length.

Register-oriented architectures have *significantly* lower data memory bandwidth [10], [14]. Lower data memory bandwidth is highly desirable since data access is less predictable than instruction access and can cause more performance problems. The existing streamlined instruction set implementations achieve this reduction in data bandwidth from either special support for on-chip data accessing, as in the RISC register windows (see Section IV-A), or the compiler doing register allocation. The load/store nature of these architectures is very suitable for effective register allocation by the compiler; furthermore, each eliminated memory reference results in saving an entire instruction. In a memory-oriented instruction set only a portion of an instruction is saved.

If implementations of the architecture are expected to have a cache, trading increased instruction bandwidth for decreased data bandwidth can be advantageous. Instruction caches typically achieve higher hit rates than data caches for the same number of lines because of greater locality in code. Instruction caches are also simpler since they can be read-only. Thus, a small on-chip instruction cache might be used to lower the required off-chip instruction bandwidth.

The question of instruction bandwidth is a tricky one. Statically, programs for machines with simplier, less densely encoded instruction sets, will obviously be larger. This static size has some secondary effect on performance due to increased working set sizes both for the instruction cache and the virtual memory. However, the potentially higher bandwidth requirements are much more important. Here we see a more unclear picture.

While the streamlined machines will definitely need more instruction bytes fetched at the architectural level, they have some benefits at the implementation level. The MIPS and RISC architectures use *delayed branches* [15] to reduce the fetching of instructions that will not be executed. A delayed branch means that instructions following a branch will be executed until the branch destination can be gotten into the pipeline. Data taken on MIPS had shown that 21 percent of the instructions that are executed occur during a branch delay cycle; in the case of an architecture without the delayed branch, that 21 percent of the cycles would be wasted. In many machine implementations the instructions are independently fetched by an instruction prefetch unit so that when the branch is taken the instruction prefetch is wasted. Another

data point that points to the same conclusion is from the VAX; Clark found that 25 percent of the VAX instructions executed are *taken* branches. This means that 25 percent of the time, the fetched instruction (i.e., the one following the branch) is not executed. Thus the bandwidth is only 80 percent of its effective bandwidth.

There are some important differences in peak bandwidth and average bandwidth for instruction memory. To be competitive in performance the complex instruction set machines must come close to achieving single cycle execution for the simple instructions, e.g., register–register instructions. To achieve this goal, the peak bandwidth must at least come close to the same bandwidth that a reduced instruction set machine will require. This peak bandwidth determines the real complexity of the memory system needed to support the processor.

Code generation for both streamlined machines and simplified machines is believed to be equally difficult. In the case of the streamlined machine, optimization is more important, but code generation is simpler since the alternative implementations of code sequences do not exist [16]. The use of code optimization, which is usually done on an intermediate form whose level is below the level of the machine instruction set, means that code generation must coalesce sequences of low level intermediate form instructions into larger more powerful machine instructions. This process is complicated by the detail in the machine instruction set and by complex tradeoffs the compiler faces in choosing what sequence of instructions to synthesize. Experience at Stanford with our retargetable compiler system [17] has shown that the streamlined instruction sets have an easier code generation problem than the more complex instruction machines. We have also found that the simplicity of the instruction set makes it easier to determine whether an optimizing transformation is effective. In retargetting the compiler system to multiple architectures, we have found better optimization results for simpler machines [18]. In an experiment at Berkeley, a program for the Berkeley RISC processor showed little improvement in running time between a compiled and carefully hand-coded version, while substantial improvement was possible on the VAX [19]. Since the same compiler was used in both instances, a reasonable conclusion is that less work is needed to achieve good code for the RISC processor when compared to the VAX and that a simpler compiler suffices for the RISC processor.

B. Microcoded Instruction Sets

The alternative to a streamlined machine is a higher level instruction set. For the purposes of this paper, we will use the term *high level instruction set* to mean an architecture with more powerful instructions; one of the key arguments of the RISC approach is that the high level nature of the instruction set is not necessarily a better fit for high level languages. The reader should take care to keep these two different interpretations of "high level" architecture distinct. The complications of such an instruction set will usually require that the implementation be done through microcode. A large instruction set with support for multiple data types and addressing modes must use a denser instruction encoding. In addition to more opcode space, the large number of combinations of opcode, data type, and addressing mode must be encoded efficiently to prevent an explosion in code size.

A high level instruction set has one major technological advantage and several strategic advantages. The denser encoding of the instruction set lowers the static size of the program; the dynamic instruction bandwidth depends on the static size of the most active portions of the program. The major strategic advantage for a high level microcoded instruction set comes from the ability to span a wide range of application environments. Although compilers will tend to use the simpler and straightforward instructions more often, different applications will emphasize different parts of the instruction set [7], [20]. A large instruction set can attempt to accommodate a wide range of application with high level instructions suited to the needs of these applications. This allows the standardization of the instruction set and the ability to interchange object code across a wide range of implementations of the architecture.

In addition to not sharing some of the implementation advantages of a simplified instruction set, a more complex architecture suffers from its own complexity. Instruction set complexity makes it more difficult to ensure correctness and achieve high performance in the implementation. The latter occurs because the size of the instruction set makes it more difficult to tune the sections that are critical to high performance. In fact, one of the advantages claimed for large instruction set machines is that they do not *a priori* discriminate against languages or applications by prejudicing the instruction set. However, similarities in the translation of high level languages could easily allow prejudices that benefited the most common languages and which penalized other languages. There is also a question of design and implementation efficiency with this type of instruction set: some portions of it may see little use in many environments. However, the overhead of that portion of the instruction set is paid by all instructions to the extent that the critical path for the instructions runs through the control unit.

IV. Architectural Support for High Level Languages

Several computers have included special language support in the architecture. This support most often focuses on a small set of primitives for performing frequent language-oriented actions. The most often attacked area is support for procedure calls. This may include anything from a call instruction with simple program counter (PC) saving and branching, to very elaborate instructions that save the PC and some set of registers, set up the parameter list and create the new activation record. A wide range of machines, from the Intel-432, to the VAX, to the Berkeley RISC microprocessor all have special reasonably powerful instructions for supporting procedure calls.

Extensive measurements of procedure call activity have been made. Source language measurements for C and Pascal have been done on the VAX by the RISC group at Berkeley [21]. Clark [7] has measured the VAX instruction set (including call) using a hardware monitor. These measurements confirm two facts. First, procedure calls are infrequent (about

10 percent of the high level statements) compared to the most common simpler instructions (data moves, adds, etc). Second, the procedure call is one of the most costly instructions in terms of execution time; the data from Berkeley indicates that it is the most costly source language statement (i.e., more machine instructions are needed to execute this source statement than most others). This high cost is sufficient to make call one of the most expensive statements, both at the machine instruction set level and at the source language level.

There are a few important caveats to examine when considering these data. The most important observation is that register allocation bloats the cost of procedure call. A simple procedure call in compiled code without register allocation is not very expensive: save the program counter, the old activation record pointer, and create a new activation record. This can be easily done in a few simple instructions, particularly if activation record maintenance is minimized. However, when an additional half-dozen register-allocated variables need to be saved the cost is in the neighborhood of 10–15 instructions. This additional cost is not inherent in the procedure call itself but is an artifact of the register allocator. Such costs should be accounted for by the register allocation algorithm [18], but are often ignored. Despite this, there is merit in lumping these saves and restores as part of the call, if this means that they can be reduced by an efficient method of executing procedure calls.

Before we look at such a method in detail, consider one other possible attack on the problem: reducing call frequency. Modern programming practice encourages the use of many small procedures; often procedures are called exactly once. While this may be good programming practice, an intelligent optimizer can expand inline any procedure that is called exactly once, and perhaps a large number of procedures that are small. For a small procedure, the call overhead may easily be comparable to the procedure size. In such cases, inline expansion of the procedure will increase the execution speed with little or no size penalty. The IBM PL.8 compiler [22] does inline expansion of all leaf-level procedures (i.e., ones that do not call another procedure), while the Stanford U-Code optimizer includes a cost-driven inline expansion phase [18].

A. Support for Procedure Call: The Register Stack

VLSI implementation greatly favors on-chip communication versus off-chip communication. This fact has led many designers to keep small caches (usually for instructions only) or instruction prefetch buffers on the chip as in the VAX microprocessors [23], [24] and the Motorola 68020. However, current limitations prevent the integration of a full size cache (e.g., 2K words) onto the same chip as the processor. An alternative approach is to use a large on-chip register set. This approach sacrifices the dynamic tracking ability of a cache, but it is possible to put a reasonably large register set on the chip because the area per stored bit can be smaller than in a cache. By allowing the compiler to allocate scalar locals and globals to the register set, the amount of main memory data traffic can be lowered substantially. Additionally, the use of register references versus memory references lowers the amount of addressing overhead. For example, in the Berkeley RISC register–register instructions execute twice as fast as memory accesses. The compiler can be selective about its allocation effectively increasing the "hit rate" of the register file. However, only scalar variables may be allocated to the registers. Thus, some programs may benefit little from this technique, although data [21] has shown that the bulk of the accessed variables are local and global scalars.

Any large register set can achieve the elimination of off-chip references and reduction of addressing overhead. However, to make use of such a large register set without burdening the cost of procedure call by an enormous amount, the register file can be organized as a stack of register sets, allocated dynamically on a per procedure basis. This concept was originally proposed for use in VLSI by Sites [25], expanded by Baskett [26], and has been studied by a wide range of people including Ditzel for a C machine [27], the BBN C machine [28], Lampson [29], and Wakefield for a direct execution style architecture [30]. A full exploration of the concept was done by the Berkeley RISC design group and implemented with some important extensions in their RISC-I microprocessor [21]. The Pyramid supermini computer [31] has a register stack as its main innovative architectural feature. We will explain the register stack concept in detail using the RISC design.

Numerous on-chip registers are arranged in a stack. On each call instruction a new frame, or window, of registers is allocated on the stack and the old set is pushed; on a return instruction the stack is popped. Of course, the push and pop actions are done by manipulation of pointers that indicate the current register frame. Each procedure addresses the registers as $0 \cdots n$ and gets a set of n registers. The compiler attempts to allocate variables to the register frame, eliminating memory accesses. Scalar global variables can be allocated to a base level frame that is accessible to all procedures and does not change during the running of the program. The effectiveness of this scheme for allocating global scalars is limited for languages that may use large numbers of base-level variables; many modern languages with module support, e.g., Ada and Modula, have this property. In addition, any variables that are visible to multiple, separately compiled routines cannot be allocated to registers. There are similar problems in allocating local variables to registers, when those variables may be referenced by inward-nested procedures; we will discuss this problem in detail shortly.

Although this concept is straightforward, there are a number of complications to consider. First, should these frames be fixed in size or variable, and if fixed how large? The advantage of using a fixed frame size is that an appropriately chosen frame size can avoid an addition cycle which is otherwise needed to choose the correct register from the register file. It also has some small simplifications in the call instruction. However, a fixed size frame will provide insufficient registers for some procedures and waste registers for others. Studies by various groups have shown that a small number of registers (around eight) works for most procedures and that an even smaller number can obtain over 80 percent

of the benefits. Most implementations of register files use a fixed size frame with from 8 to 16 registers per frame. The stack cache design of Ditzel demonstrates an elegant variable size approach.

In today's technology a processor can contain only a small number of such register frames; e.g., the RISC-II processor has 8 such frames of 16 registers each. Increasing integrated circuit densities may allow more frames but the diminishing returns and implementation disadvantages, which we will discuss shortly, indicate that the number of frames should be kept low. Because it is impossible either to bound at compile-time, or to restrict the calling depth *a priori,* the processor must deal with register stack overflow.

When the register stack overflows, which only happens on a call instruction as a new frame is allocated, the oldest frame must be migrated off the chip to main memory. This function can be done with hardware assist, in microcode as on the Pyramid, or in macrocode as on RISC. In a more complex processor, the oldest stack frames might be migrated off-chip in background using the available data memory cycles. When the processor returns from the call that caused the overflow, the register stack will have an empty frame and the frame saved on the overflow can be reloaded from memory. Alternatively, the reloading can be postponed until execution returns to the procedure whose frame was migrated.

One of the interesting results obtained by the studies done for the RISC register file concerns measurements done of call patterns and the implications for register migration strategies [32]. If we assume that calls are quite random in their behavior, the benefits of the register stack can be quite small. In particular, if the call depth varies widely, then a large number of saves and restores of the register stack frames will be needed. In such a case, the register stack with a fixed size frame can even be slower than a processor without such a stack because all registers are saved and restored whether or not they are being used. However, if the call pattern tends to be something like "call to depth k, make a significant number of calls from level k and higher but mostly within a few levels of k, before backing out," then register stack scheme can perform quite well. It will need to save and restore frames getting to and returning from level k, but once at level k the number of migrations could be very small.

Data collected by the Berkeley RISC designers indicate the latter behavior dominates. This also leads to another important insight: it may be more efficient to migrate frames in batches, thus cutting down on the number of overflows and underflows encountered. However, a recent paper [32] shows that the optimal number of frames to move varies between programs. Furthermore, that study shows that past behavior is not necessarily a good guide when choosing the number of frames to migrate. Simple strategies of moving a single frame or two frames are a good static approximation and should be used.

Because the language C does not have nested scopes of reference, a register file scheme for C need provide addressability only to the local frame and the global frame. This can be easily done by splitting the register set seen by the procedure so that registers $0 \cdots m$ address $m + 1$ global registers and registers $m \cdots n$ reference the $n - m + 1$ local registers. Furthermore, since these global registers are the only globally accessible registers they are never swapped out.

Languages like Ada, Modula, and Pascal have nested scopes and allow up-level referencing from any nested scope to a surrounding scope. This means that the processor must allow addressing to all the register frames that are global to the currently active procedure. Because up-level referencing to intermediate scopes (i.e., to a scope that is not the most global scope) is rare, such addressing can be penalized without significant overall performance loss. In the simple case, the addressing is straightforward: the instruction can give a relative register-set number and a register number (offset in the register-set) and the processor can do the addressing. Even if this instruction is very slow, the performance penalty will be negligible. The complicated case arises when a register stack overflow has occurred and the addressed register frame has been swapped out. In this case, the register reference must become a memory reference.

A similar problem exists with reference (or pass by address) parameters. Variables that are passed as reference parameters may be allocated in a register and may not even have a memory address that can be passed. The language C allows the address of a variable to be obtained by an operator; this causes problems since register-allocated variables will not have addresses.

Fortunately, there are two solutions [29] to these problems. The first is to rely on a two-pass compilation scheme to detect all up-level references or address references and to prevent the associated variable from being allocated in the register stack. This requires a slightly more complex compiler and has some small performance impact. An alternative solution uses some additional hardware capability and will handle both types of nonlocal references. Let us assume that each register frame (and hence each register) has a main memory address, where it resides if it is swapped out. A nonlocal reference (up-level in the scope of the reference) can be translated by computing the address of the desired frame, which is a function of

- the address in memory for the current frame (which is based only on the absolute frame number), and
- the number of frames offset from the current frame, which is based on the differences in lexical levels between the current procedure and the scope of the referenced variable.

With these two pieces of information we can calculate the memory address of the desired frame. Likewise, for a reference parameter that is in a register we can calculate and pass the memory address assigned to the register location in the frame.

Now, this leaves only one problem: some memory addresses can refer to registers that may or may not currently be in the processor. If the referenced register window has overflowed into memory, then we can treat the reference as a conventional memory reference. If the register is currently on-chip, then we need to find the register set and access the on-chip version. Since this access need not be fast, it is easy to check the current register file and get the contents, or to allow the memory references to complete [33].

A register stack allows the use of a fairly simple register allocator, as well as mitigating the cost of register save/restore at call statements. Compilers often attempt to speed up procedure linkage by communicating parameters and return values in the registers. If the compiler is not doing global register allocation, this task is easy; otherwise, the compiler must integrate the register allocation in existence at the call site with the register usage needed for parameter passing. This communication of parameters in registers can improve performance by about 10 percent. However, using this improvement with a straightforward register stack is impossible since neither procedure can address the registers of the other in a fast and efficient manner.

The RISC processor extended the idea of the register stack to solve this problem [14]. On RISC the frames of a caller and callee overlap by a small number of registers. That is, the j high order registers of the caller correspond to the j low order registers of the callee. The caller uses these registers to pass the actual parameters, and the callee can use them to return the procedure result. The number of overlapping registers is based on the number of expected parameters and on hardware design considerations.

The disadvantages of the register set idea come from three areas. First, improved compiler technology, mostly in the form of good models for register allocation [34]–[36], makes it possible for compilers to achieve very high register "hit" rates and to more efficiently handle saving and restoring at procedure call boundaries. Good allocation of a single register set with a cache for unassigned references could be extremely effective. Since the registers are multiport, the size of the individual register cells and their decode logic means the silicon area per word of storage may approach the area occupied per word in a set associative cache. Another disadvantage with respect to a cache is that the register stack is inefficient: only a small fraction (i.e., one frame) is actively being used at any time. In a cache a larger portion of the storage could be used. Of course, the effectiveness of the register stack is increased when procedure calls are frequent and the portion of the register stack being used changes quickly.

A second disadvantage is that the use of a register set clearly increases the process switching time, by dramatically increasing the processor state. Although process switches happen much less frequently than procedure calls, the true cost of this impact is not known. Studies [37]–[39] have shown that the effect of process switches on TLB and cache hit ratios can be significant.

Third, the register set concept presents a challenging implementation problem, particularly in VLSI. The number of frames is ideally as large as possible; however, if the register file is to be fast it must be on-chip and close to the central data path. The size and tight coupling to the data path will result in slowing down the data path at a rate dependent on the size of the register file; this cost at some point exceeds the merit of a larger register file. The best size for the register stack and its impact on the cycle time is difficult to determine since it depends a great deal on both the implementation and the benchmarks chosen to measure performance. We will discuss the issue of implementation impact later in the paper. The final worth of the register stack ideas remains to be seen; they have been incorporated in a commercial machine [31] and used in the RISC chip. However, when measured against improved compiler technology and the cost in the cycle time, the real benefits remain unknown.

V. Systems Support

A processor executes compiled programs; however, without an operating system the processor is essentially useless. The operating system requires certain architectural capabilities to achieve functional performance with reasonable efficiency. Perhaps the most important area for operating systems support is in memory management.

Support for memory management has become a feature of almost all computer architectures. The initial microprocessors did not provide such support and even in machines as late as the M68000 no support for demand paging is provided, although support is provided in the M68010. Current microprocessors must compromise between providing all necessary memory management features on-chip and the real limitations of silicon area and interchip delays. Thus, some design compromises are usually made to achieve an acceptable memory mapping mechanism. After looking at the requirements, we will examine the memory mapping support in three processors: the 8-chip VLSI VAX, the Intel iAPX432, and the Stanford MIPS processor. Each of these processors makes a different set of design compromises.

Modern memory systems provide virtual memory support for programs. In addition, the system must also implement memory protection and help to minimize the cost of using virtual memory as well as improve memory utilization. Program relocation is a function of the memory mapping system; segmentation provides a level of relocation that may be used instead of or in addition to paging. Implementing a paged virtual memory requires translation of virtual addresses into real addresses via some type of memory map. Support for demand paging will require the ability to stop and restart instructions when page faults occur. Protection can be provided by the hardware on a segment and/or page basis.

A. VAX and VLSI VAX Memory Management

The memory management scheme used in the VAX architecture is a fairly conventional paging strategy. Some of the more interesting aspects of the memory architecture arise when the implementation techniques used in the VLSI VAX's are examined.

The 2^{32} byte virtual address space is broken into several segments. The main division into two halves provides for a system space (a system wide common address space) and a user process address space. The process address space is further subdivided into a P0 region, used for programs, and a P1 region, used for stack-allocated data. The heap, from which dynamically managed nonstack data are allocated, is placed above the code in the P0 region. Fig. 1 shows this breakdown. The P0 and P1 regions grow towards each other, while the system region grows towards its upper half, which is currently reserved. The decomposition into system and process space has two main effects: it guarantees in the architecture a shared region for processes as well as for the operating system, and it allows the processor implementation to

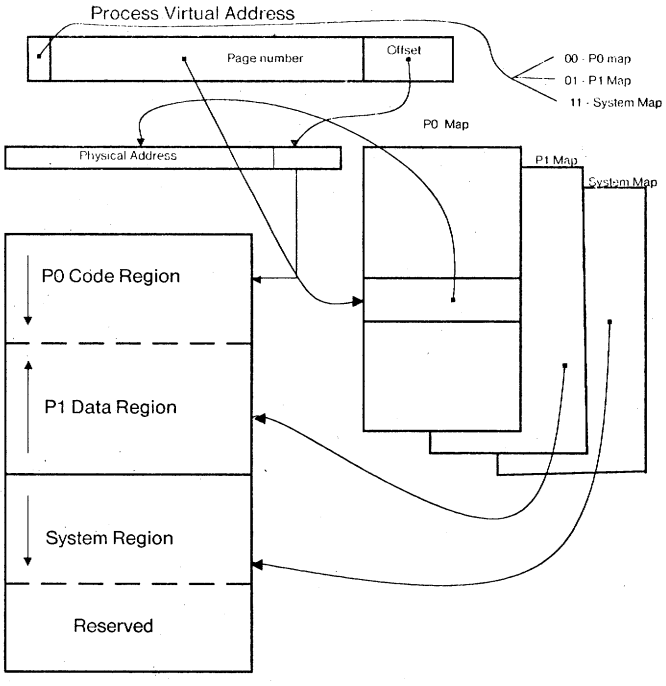

Fig. 1. VAX address space mapping.

distinguish between memory references that belong to a single process and those that are owned by the operating system or shared among processes. Both the operating system and the memory mapping hardware can take advantage of this knowledge. The operating system can use the page address to determine if a page is shared; this may affect the way in which it is handled by the page replacement routines. The use of the P0 and P1 spaces results in increased efficiency in page table utilization, as we will see shortly.

The two high order bits of a virtual address serve to classify the reference into the P0, P1, or system region. Each region uses its own page table. The next twenty high order bits are used to index the page table, while the low order nine bits are used as the page offset. A set of registers tracks the location of the page table for each region. These registers also keep the current length of the page table, so that the entire table need not be allocated in memory, if it is not used. The relatively small page size (512 bytes) is probably not optimal for most VAX machines that are used with real memory of 1–8 MB.

From an architectural viewpoint the major distinguishing factor of the VAX memory management scheme is the decomposition of the address space into four regions with separate page tables. An advantage of this scheme is that it helps prevent contention in caches and translation lookaside buffers (TLB's) by separating those portions of the address space. Another advantage is that the size of the page tables needed can be reduced since each area can have its own table with its own limit register. A single page table with a limit register cannot be used for this purpose because high level language programs typically include two areas whose memory allocations must grow: the heap (for dynamically allocated objects) and the stack. Furthermore, in growing the stack, the compiler assumes that stack frames will be contiguous in virtual address space. Thus, if a single page table is to be used it will require a pair of limit registers. This need is obviated by splitting user space into the P0 and P1 region, each with a single register. This solution is an interesting contrast to the MIPS approach, which we will discuss shortly.

On the VAX 11/780, the translation lookaside buffer uses some portion of the high order part of the virtual address as the index. This splits the buffer into two partitions: the first to hold references to pages in system space and the second to hold references to pages in the active process' space. The benefit of this approach is that only the second partition of the TLB need be cleared on a process switch; the first partition is process independent. However, studies by Clark [40] have shown that this split is not necessarily beneficial. For example, substantially higher TLB miss rates for system space references, indicate that the partition in the TLB sizes is suboptimal.

There are two VAX implementations that are in VLSI; we discuss these in further detail in the survey section. We will look at the memory management implementation on the 8-chip VLSI VAX. In the 8-chip set, memory management is handled at two levels.

1) The main processor chip, responsible for instruction fetch and execution, has a mini-TLB with 5 entries.

2) The Memory/Peripheral Subsystem chip contains the tag array for a 512-entry TLB as well as the tag array for a 2K cache.

The mini-TLB allows very fast (50 ns) address translation on-chip. The small size allows the buffer to be fully associative; however, the TLB is partitioned into a one-entry instruction-stream buffer (always used for the currently executing instruction) and a four-entry data-stream buffer. This prevents the ambitious instruction prefetch unit from interfering with the execution of the current instruction, which may require up to five operands to be mapped. When a hit is obtained on the internal TLB, a physical address is driven out to the memory subsystem chip, which acts as the cache. This whole process occurs in a 200 ns cycle. If the internal TLB misses, but the external TLB hits, a single cycle penalty is taken and the data are moved into the internal TLB.

This design is an interesting compromise between the limitations of silicon area that prohibited a large on-chip TLB and the need to have efficient memory address translation. The relatively small penalty incurred when the mini-TLB misses but the main TLB hits, allows operation as if the TLB were quite large. A substantial penalty is only incurred if the main TLB misses, and microcode intervention is required to compute the physical address. The larger 512 entry TLB will yield a higher hit ratio than the 128 entry TLB used in the VAX 11/780. In fact the judicious choice of a small on-chip TLB coupled with a larger off-chip TLB with a minimal penalty, can probably achieve performance comparable to the one-level TLB used in the 11/780.

B. Intel iAPX432 Memory Management

The 432 supports a capability-based addressing scheme. Every memory address consists of a segment and an offset; there may be up to 2^{24} segments and each segment has at most 2^{16} bytes. Although few individual objects will require more than one segment, many programs will use a total stack or heap size that requires multiple segments. Because such allocation requirements are nearly impossible to predict at

compile-time, the compiler must assume that references to other parts of the stack and references to the heap will require a segment change. This will result in a performance loss if the number of segments that are simultaneously active becomes large.

The 432 uses a more powerful scheme than segment plus offset: the segment designator is an access descriptor that contains the access rights for the segment, as well as information for addressing the segment. These access descriptors are similar to the concept of capabilities [41]. The access descriptors are collected into an access segment, which is indexed by a segment selector. The address portion of the access descriptor contains a pointer to a segment table, which specifies the entry providing the base address of the segment. The offset to the segment is part of the original operand address, whose format is described in a following section. This two-level mapping process is illustrated in Fig. 2. The 432's data processor chip contains a 22 element cache on the access segment and the segment table; 14 of the 20 entries are preassigned for each procedure, two are reserved for object table entries, and six entries are available for generic use. This cache reduces the frequency with which the hardware must examine the two-level map in memory.

The 432 architecture uses the access segment to define a domain for a program. A program's domain of access consists of an access segment that provides addressing to multiple data and program segments. For program segments, the access descriptor indicates that the object is a program and checks that the execution of instructions occurs only from an instruction segment. Similarly, all branches are checked to be sure that they will transfer to an instruction segment. In addition to the instruction segments, the 432 defines both data and stack segments, as well as constant segments.

The 432 addressing scheme achieves two primary objectives: support for capabilities, and support for fine-grained protection. The major objection raised to the addressing scheme is that it is more complicated and powerful than is necessary. The use of capabilities has been explored in several systems [42], [43] with limited success at least partially due to a lack of hardware support. Most of these systems found that capability based addressing was expensive and this may have prevented its use. An interesting discussion of the issues is contained in a paper by Wilkes [44]. The other major advantage claimed for the 432 is that it provides fine grained protection to allow users to protect against array bounds violations and references out of a module, by limiting the size of the segment. However, a careful examination of the requirements imposed by Ada, the host language for the 432, shows that the segment based approach is only usable when each object that can be indexed or addressed dynamically is in a single segment. When this is not the case, runtime checks are required by the compiler and these checks guarantee that the reference is legal, making the hardware segment checking superfluous. There are several reasons why allocating each such data object to a unique segment is an unsuitable approach. The most important reason is that it will cause a large increase in the number of segments (one per data object to be protected), which will decrease the locality of segment references and hamper the effectiveness of the address cache. Address cache misses must be translated at a much slower rate (approximately 5 μs per translation) causing a substantial degradation in performance.

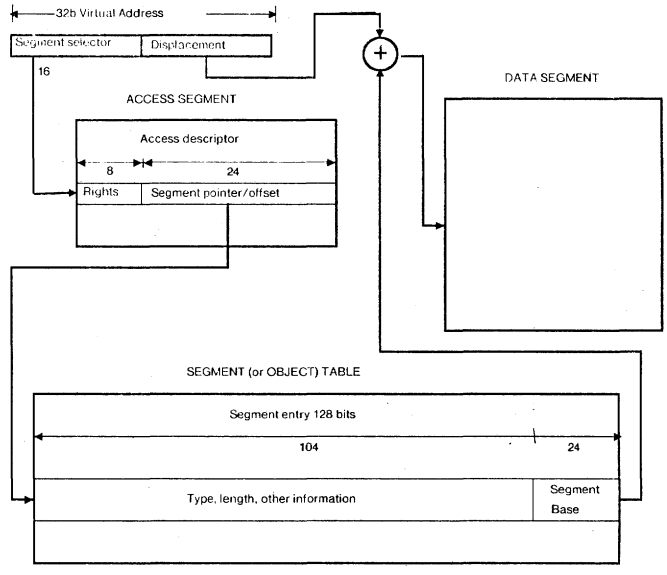

Fig. 2. iAPX-432 address mapping.

Despite these objections, the 432 addressing mechanism does provide the most cost effective implementation of capabilities in hardware to date. Future evolution of software systems, such as Smalltalk, may make object-based environments more important. When such environments are very dynamic and a high level of protection is desired, the 432 capability-based mechanism offers an attractive vehicle for implementation. The challenge to such architectures will be to make the performance penalties for a capability-based system insignificant when compared to their functional benefits.

C. MIPS Memory Management

In addition to the standard requirements for virtual memory mapping, the Stanford MIPS processor attempts to support a large uniform address space for each process, and fast context switching. One mechanism for facilitating context switching is the incorporation of a process identification number into the virtual memory address. The use of the process id number helps achieve fast context switches by allowing the cache and memory address translation units to avoid the cold start penalties. These penalties appear in systems that require caches and translation buffers to be flushed because processes share the same virtual address space. The process id approach also allows the use of a large linear address space, avoiding the difficulties that arise when segment boundaries are introduced. The realities of the MIPS implementation technology (a 4 μm channel length nMOS) meant that it was not feasible to include all of the virtual to physical translation on the same chip as the processor.

Consequently, a novel memory segmentation scheme was added to the architecture; it is designed to work with a conventional page mapping scheme implemented with the use of an off-chip TLB. Each process has a process address space of 2^{32} words. The first step of the translation is to remove the top n bits of the address and replace them by an n-bit *process identifier* (PID). Fig. 3 shows the generation of this virtual

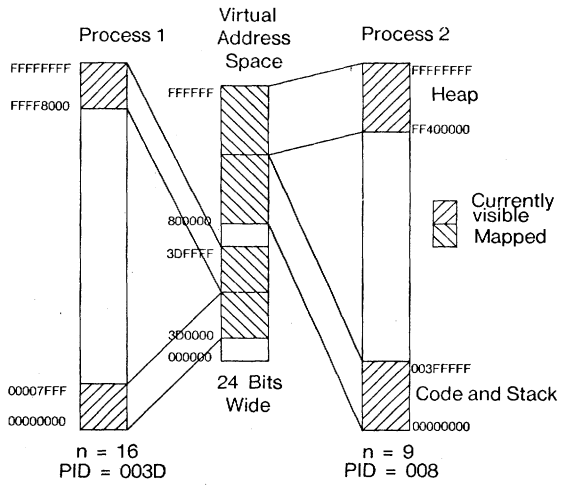

Fig. 3. MIPS on-chip address translation.

address. By restricting the process address under the mask to be all ones or all zeros, the accessible portion of the process address space becomes the low 2^{32-n-1} and the high 2^{32-n-1} words; this allows two large-address segments to grow towards one another but not overlap. An attempt to access any of the nonvisible words (by generating an address that fails to have all ones or all zeros under the mask) will cause an exception. The operating system can then remap the process identifier in such a way as to give the faulting process a smaller PID number and thus a larger visible portion of its process address space. This will only happen if the program used more heap or stack space then it was initially allocated. This is similar to the procedure of expanding the VAX page tables by increasing the page table limit register for one of the address spaces.

An additional level of translation maps the processor's virtual address to a physical address. This level of mapping could be implemented with a wide variety of hardware ranging from a direct map to a TLB.

The constraining factor on MIPS is that the total size of all the visible process address spaces must be less than the size of the implementation's virtual address space. This restricts the number of processes that can actively use the memory map; should this number of processes become very large, the operating system will need to periodically reuse a PID. Whenever a process with a shared PID is made active, a process and cache sweep will be needed. This should not happen frequently, since the number of small processes that can be created is very large.

D. Summary

Modern memory management techniques seem to have become an issue of major concern for VLSI processor architectures. The approaches described in this section have interesting differences.

The VAX architecture defines a partitioning of its address space and a two-level memory mapping scheme. This scheme permits the use of a small number of page limit registers to control the mapping of several potentially growing user storage areas. Table I makes the comparison among translation buffers (which are required to achieve acceptable memory mapping performance on the VAX) used on the VAX 11/780 mainframe, the 9-chip VLSI VAX, and the single-chip MicroVAX-32.

TABLE I
SUMMARY OF TRANSLATION BUFFER FEATURES ON VAX IMPLEMENTATIONS

Machine	Entries	Features
11/780	128	Direct mapped; 1/2 system, 1/2 user
VLSI VAX	5 on processor chip	Fully associative; limit one instruction entry
	512 on campanion-chip	Set associative; not reserved
MicroVAX-32	8	Fully associative

The Intel 432 offers one of the most sophisticated memory mapping and protection schemes. It provides access to a large segmented address space (with small segments, unfortunately). Segments are protected by capabilities that provide extremely flexible control over access to the segment. An address cache (or translation buffer) reduces the need for a costly two-level translation of addresses.

MIPS uses the simplest memory mapping scheme of these three architectures. Its important features are a large, (optionally) unpartitioned address space that allows any of a variety of page mapping schemes implemented off-chip. The architecture has the novel feature of a variable-sized process id that is included as part of the virtual address; this helps decrease the loss of performance when a context switch occurs as well as communicating the process id to allow for protection checks by off-chip hardware.

VI. Some Interesting VLSI Architectures

In this section we discuss some of the interesting points of several commercial and experimental microprocessor architectures. The purpose of this examination is not a definitive comparison of the architectures. Our goal is to discuss some of the architectural features, examine tradeoffs in the architecture, and analyze the methods used to achieve performance as well as the limitations on performance. The architectures chosen represent only a portion of the available VLSI processors. We have chosen these processors because they provide interesting and contrasting viewpoints. The implementations of these processors vary: some are commercial, and some are university-based experiments. Thus, implementation specific data should be used for interpreting the general behavior of an architecture; differences in the levels of implementation and the size of the implementation efforts make direct performance comparisons unreasonable.

A. The Berkeley RISC Microprocessor

The Berkeley RISC I and II processors [21] were the first microprocessors to explore the concept of a simplified, or reduced, instruction set. The architecture has a total of 31 instructions and is a load/store machine. The 32-bit integer ALU operations (which include add, subtract, logical, and shift operations, but not multiply or divide) all have a 3-operand format, where the operands are registers, or one of the source operands may be a 13-bit immediate constant. The Berkeley RISC provides memory addressing support for bytes, half-words (16 bits), and words (32 bits). The single addressing mode is: register contents plus offset; it can be

used to synthesize absolute addressing (by ignoring the register) and register indirect (by making the offset zero).

In addition to the ALU, load, and store instructions, RISC has a set of delayed branch instructions, call and return instructions, and processor status instructions. By simplifying the instruction set, instruction fetch and decode was straightforward, and the amount of control logic needed on the processor was substantially reduced. However, this very loose encoding of instructions means that instruction density is much lower than for other architectures, in the range of 40–70 percent lower. The RISC processor is able to achieve a one machine cycle execution of register–register instructions and a two machine cycle memory access instruction. Its instruction set is summarized in Table II.

The major innovation of the RISC processor has been the addition of a large register stack with overlapping register windows. This idea was explained in detail in the section on register stacks. The register window concept is responsible for much of the performance benefits that RISC demonstrates. The simplicity of the other parts of the instruction set allow reduction in the silicon area needed to implement the processor's control portion, thus freeing up space for the large register file.

There have been two implementations of RISC. The first, RISC-I, did not obtain the desired speed but was nearly perfect functionally. Its 1.5 MHz clock cycle with three clock cycles per instruction, yields an execution rate of one-half million register–register instructions per second. The RISC-II implementation is a completely new, and more sophisticated design; it is functionally correct and runs with an 8 MHz clock cycle at room temperature using a 4 μm single-metal, nMOS process. With four clock cycles per machine cycle, the 4 μm part has a register–register instruction execution rate of 2 MIPS. A 3 μm version has also been fabricated; it achieves a 3 MIPS execution rate for register–register instructions by using a 12 MHz clock. All the Berkeley RISC processors execute load or store instructions at one-half the rate of register–register instructions. The RISC architecture and the microengine are discussed in detail in Katevenis' thesis [33].

B. The Stanford MIPS Processor

The MIPS processor [45]–[47] probably represents the most streamlined processor design to date. A key point in the MIPS philosophy is to expose, in the instruction set, all the processor activity that could affect performance. This philosophy coupled with the concept of a streamlined instruction set allows a shift of functions from hardware to software. This shift in responsibility simplifies the requirements placed on the hardware allowing the machine to have a higher clock speed (4 MHz) than it would otherwise and a fast pipeline (one instruction initiated every two clock cycles).

The best example of this shift of responsibility is that the compiler assumes responsibility for simple register access conflicts in the pipeline. The pipeline hardware has no register interlocks; instead, the compiler is forced to compile code that accounts for the pipeline structure and the register usage patterns. Since storage access may take an undetermined amount of time (due to cache misses, memory delays, etc.), the pipeline control must delay instructions in the pipeline; however, the compiler is still responsible for scheduling the minimum appropriate delay (i.e., the cache access time). This process amounts to scheduling the code on the processor and has two benefits: it speeds up the code by eliminating pipeline delays due to register utilization, and it makes the pipeline hardware simpler and more regular. This simplicity and regularity allows for a faster clock cycle for the pipeline. This pipeline scheduling as well as other features in the MIPS architecture are oriented towards the use of optimizing compiler technology.

TABLE II
SUMMARY OF RISC INSTRUCTION SET

Instruction	Operands		Comments
add	Rd,Rs,S2	Rd := Rs + S2	integer add
addc	Rd,Rs,S2	Rd := Rs + S2 + carry	add with carry
sub	Rd,Rs,S2	Rd := Rs - S2	integer subtract
subc	Rd,Rs,S2	Rd := Rs - S2 - borrow	subtract with borrow
subi	Rd,Rs,S2	Rd := S2 - Rs	integer subtract reverse
subci	Rd,Rs,S2	Rd := S2 - Rs - borrow	reverse subtract with borrow
and	Rd,Rs,S2	Rd := Rs & S2	bitwise AND
or	Rd,Rs,S2	Rd := Rs | S2	bitwise OR
xor	Rd,Rs,S2	Rd := Rs xor S2	bitwise EXCLUSIVE OR
sll	Rd,Rs,S2	Rd := Rs shifted by S2	shift left
srl	Rd,Rs,S2	Rd := Rs shifted by S2	shift right logical
sra	Rd,Rs,S2	Rd := Rs shifted by S2	shift right arithmetic
ldw	Rd,(Rx)S2	Rd:=M[Rx+S2]	load word
ldhu	Rd,(Rx)S2	Rd:=M[Rx+S2] (align, zero-fill)	load half unsigned
ldhs	Rd,(Rx)S2	Rd:=M[Rx+S2] (align, sign-ext)	load half signed
ldbu	Rd,(Rx)S2	Rd:=M[Rx+S2] (align, zero-fill)	load byte unsigned
ldbs	Rd,(Rx)S2	Rd:=M[Rx+S2] (align, sign-ext)	load byte signed
stw	Rm,(Rx)S2	M[Rx+S2]:=Rm	store word
sth	Rm,(Rx)S2	M[Rx+S2]:=Rm (align)	store half
stb	Rm,(Rx)S2	M[Rx+S2]:=Rm (align)	store byte
jmp	COND,(Rx)S2	if COND then PC:=Rx+S2	cond. jump, indexed
jmpr	COND,Y	if COND then PC:=PC+Y	cond. jump, PC-rel.
callx	Rd,(Rx)S2	Rd:=PC; PC:=Rx+S2; CWP--	call indexed
callr	Rd,Y	Rd:=PC; PC:=PC+Y; CWP--	call PC-rel.
ret	(Rx)S2	PC:=Rx+S2; CWP++	return
ldhi	Rd,Y	Rd<31:13>:=Y; Rd<12:0>:=0	load immediate high
gtlpc	Rd	Rd:=lastPC	save value for restarting pipeline
getpsw	Rd	Rd:=PSW	read status word
putpsw	Rm	PSW:=Rm	set status word
reti	(Rx)S2	PC:=Rx+S2; CWP++;	return from interrupt
calli			call an interrupt

Rd, Rs, Rx, Rm: a register (one of 32, where R0=0)
S2: either a register or a 13-bit immediate constant;
COND: 4-bit condition;
Y: 19-bit immediate constant;
PC: Program-Counter;
CWP: Current-Window-Pointer;
All instructions can optionally set the Condition-Codes.

Like most streamlined machines MIPS is a load/store architecture. However, unlike most other new processor architectures, it provides only word-addressing. This choice was made because word addressing dominates in frequency and is also simpler and faster. Byte addressing is supported using a set of instructions for manipulating byte pointers, which are single-word addresses whose two low order are bits used to specify a byte within a word. The MIPS' byte pointer facilities are similar to those supported in the DEC-10 architecture, but are slightly easier to deal with in a compiler and handle primarily 8-bit bytes.

MIPS has 16 orthogonal general purpose registers. All ALU instructions are register–register and are available in two and three operand formats. One of the source registers may be replaced by a small constant. Support for integer multiplication and division consists of special multiply and divide instructions that provide two bits of a Booth multiply sequence and one bit of a divide sequence. The assembly language includes multiply and divide instructions that are expanded into sequences of multiply or divide steps. No hardware is provided for floating point arithmetic.

The architecture has no condition codes. Instead, there is a compare-and-branch operation. Abandoning condition codes in favor of a compare-and-branch instruction has bene-

TABLE III
MIPS ASSEMBLY INSTRUCTIONS

Operation	Operands	Comments	
Arithmetic and logical operations			
Add	src1, src2, dst	dst: = src2 + src1	Integer addition
And	src1, src2, dst	dst: = src2 & src1	Logical and
Ic	src1, src2, dst	dst: = byte src1 of dst is replaced by src2	Insert byte
Or	src1, src2, dst	dst: = src2 | src1	Logical or
Rlc	src1, src2, src3, dst	dst: = src2||src3 rotated by src1 positions	Rotate combined
Rol	src1, src2, dst	dst: = src2 rotated by src1 positions	Rotate
Sll	src1, src2, dst	dst: = src2 shifted left by src1 positions	Shift left logical
Sra	src1, src2, dst	dst: = src2 shifted right by src1 positions	Shift right arithmetic
Srl	src1, src2, dst	dst: = src2 shifted right by src1 positions	Shift right logical
Sub	src1, src2, dst	dst: = src2 - src1	Integer subtraction
Subr	src1, src2, dst	dst: = src1 - src2	Reverse integer subtraction
Xc	src1, src2, dst	dst: = byte src1 of src2	Extract byte
Xor	src1, src2, dst	dst: = src2 ⊕ src1	Logical xor
Transport operations			
Ld	A[src], dst	dst: = M[A + src]	Load based
Ld	[src1 + src2], dst	dst: = M[src1 + src2]	Load based-indexed
Ld	[src1>>src2], dst	dst: = M[src1 shifted by src2]	Load based-shifted
Ld	A, dst	dst: = M[A]	Load direct
Ld	I, dst	dst: = I	Load immediate
Mov	src, dst	dst: = src	Move (byte or register)
St	src1, A[src]	M[A + src]: = src1	Store based
St	src1, [src2 + src3]	M[src2 + src3]: = src1	Store based-indexed
St	src1, [src2>>src3]	M[src2 shifted by src3]: = src1	Store based-shifted
St	src, A	M[A]: = src	Store direct
Control transfer operations			
Bra	dst	PC: = dst + PC	Unconditional relative jump
Bra	Cond, src1, src2, dst	PC: = dst + PC if Cond(src1,src2)	Conditional jump
Jmp	dst	PC: = dst	Unconditional jump direct
Jmp	A[src]	PC: = A + src	Unconditional jump based
Jmp	@A[src]	PC: = M[A + src]	Unconditional jump indirect
Trap	Cond, src1, src2	PC: = 0 if Cond(src1, src2)	Trap instruction
Other operations			
SavePC	A	M[A]: = PC_{-3}	Save multi-stage PC after trap or interrupt
Set	Cond, src, dst	dst: = -1 if Cond(src,dst) dst: = 0 if not Cond(src,dst)	Set conditional

fits for both the compiler and the processor implementation [48]. It simplifies pipelining and branch handling in the implementation and eliminates the need to attempt optimization of the condition code setting.

The compiler and operating system would prefer to see a simple well-structured instruction set. However, this conflicts with the goal of exposing all operations, and allowing the internal processor organization to closely match the architecture. To overcome these two conflicting requirements, the MIPS instruction set architecture is defined at two levels. The first level is visible to the compiler or assembly language programmer. It presents the MIPS machine as a simple streamlined processor. Table III summarizes the MIPS definition at this level.

Each MIPS assembly-level instruction is translated to machine level instructions; this translation process includes a number of machine-dependent optimizations: organizing the instructions to avoid pipeline interlocks and branch delays, expanding instructions that are macros, and packing multiple assembly language instructions into one machine instruction.

The machine-level instruction set of MIPS is closely tied to the pipeline structure. The pipeline structure can be explained by examining the memory and ALU utilization. Each instruction goes through the five stages of the pipeline, with an instruction started on every other stage. During a single instruction, two pipestages are allocated for instruction fetch and decode, two for ALU usage, and one for a data memory access cycle. The use of two ALU cycles makes it possible to accommodate compare and branch in a single instruction, although the data memory cycle is unused in such an instruction.

The execution of a load instruction requires the use of the ALU only once to compute the effective address of the item that is to be retrieved from memory. This arrangement leaves the ALU idle for one machine cycle during the execution of a simple load instruction. Since the ALU is not busy and most load instructions do not require a full 32-bit instruction word, an additional register–register operation can be done in the same instruction. The companion ALU instruction is an independent two operand register–register instruction. Some forms of the load (e.g., long immediate) sacrifice the ALU instruction encoding space for another use. Since an ALU instruction only uses the ALU once and is a small instruction, the instruction set allows a three operand ALU operation and a two operand ALU operation to be combined in every instruction word. This combination is particularly effective for executing arithmetic code that tends to be operation intensive. Store instructions, and to a lesser extent branch instructions that involve a reference to memory, are treated correspondingly.

These two component instructions consist of two separate and independent halves. The same is true of most other instructions. For example, a compare-and-branch instruction involves a condition test and a PC-relative address calcu-

lation. This separation of the instruction into two distinct parts allows the instruction to be viewed as a series of distinct single operators to be executed in the pipeline. This approach simplifies the pipeline control and allows the pipeline to run faster.

Translation between Assembly language (the architectural level) and the hardware instructions (organizational level) is done by the reorganizer [49]. The reorganizer reorders the instructions for each basic block to satisfy the constraints imposed by the pipeline organization; this reorganization establishes at compile time the schedule of instruction execution. Scheduling instructions in software has two benefits: it enhances performance by eliminating instances of pipeline interlocking, and it simplifies the pipeline control hardware allowing a shorter time per pipestage. The disadvantage in the MIPS case is that the absence of a legal instruction to schedule will force the insertion of a no-op instruction; this results in a slight code size increase (less than 5 percent in typical applications [49], [50]) but has no impact on execution speed. MIPS also includes a delayed branch, which is the natural extension of the absence of interlocks to the program counter.

Studies on the MIPS instruction set show that the combination of a simplified pipeline structure and the optimizations performed by the code reorganizer are responsible for a factor of two in performance improvement.

C. The Intel iAPX-432 Processor

The Intel iAPX-432 [2] represents the most complete approach to integrating the needs of an entire software environment onto silicon. Among the characteristics of the iAPX-432 architecture are the following:

- a dense encoding of instructions with variable instruction lengths in bits. Instructions may also start and stop on arbitrary bit boundaries;
- an object-oriented support mechanism, allowing for creation and protection of an object;
- a packet-switched bus protocol;
- support for many standard operating system functions;
- provision for transparent multiprocessing; and
- support for fault-tolerant operation.

The iAPX-432 also represents an architecture that meets many of the goals and specific design principles of Flynn's ideal machine [51]. The similarities between the iAPX-432 and the high level DEL machines proposed by Hoevel (for Fortran [52]) and Wakefield (for Pascal [30]) are considerable. The major difference is the absence of a register set or stack cache in the iAPX-432. However, the use of memory and stack operands, bit encoded instructions, data typing, and symmetric addressing are all key principles in the DEL designs.

The iAPX-432 implementation consists of three major chips: two of these comprise the general data processor (GDP) and the other is the interface processor (IDP). The two-chip GDP consists of an instruction decode unit, which also contains most of the microcode, and the microexecution unit. In addition to executing microinstructions, the microexecution unit performs the memory mapping and protection functions within the iAPX-432 architecture. In this section we will concentrate on the components of the instruction set not related to memory mapping; we will briefly mention the other operating system support features.

Among the most significant features in the iAPX-432 architecture is its support for a wide variety of data types, including

- 8-bit characters,
- 16-bit signed and unsigned integers,
- 32-bit signed and unsigned integers,
- a variable length bit field: 1–31, or 1–16 bits in length, and
- 32-bit, 64-bit, and 80-bit reals.

Complete sets of arithmetic and (where appropriate) logical operators are defined for each data type. The iAPX-432 is the first microprocessor to define and implement floating point support in the architecture. Conditional branch instructions are defined as taking Boolean operands. The remainder of the instruction set is largely devoted to operators for: object manipulation, protection, context management (which we discuss in Section V), and process communication.

Data operands reside either in data memory or on an operand stack implemented in memory with caching of the top element of the stack. Although the stack is efficient when measured by the number of bits needed to represent a computation, it is not believed to be a good representation for compilers and code optimization [53]. Memory–memory operations are efficient when measured by the number of operations needed for a program. However, since there are no on-chip registers, it is not possible to optimize references to commonly used variables.

The iAPX-432 instructions have one, two, or three operands and complete symmetry with respect to addressing modes. Since the instruction formats allow arbitrary bit lengths, memory operands can be mixed with stack operands with no loss of encoding efficiency. Of course, the task of instruction fetching and decoding is substantially more complex; we will discuss this topic further in a latter section.

The iAPX-432 uses a two-part memory address consisting of a segment and a displacement. Segment-based addressing has been discussed in the earlier section on memory management and is summarized by Fig. 2. The segments may be up to 2^{16} bytes long; although fixed limited size segments help provide memory protection, they pose a major problem for segments that need to grow larger than this size. Managing the activation record stack and heap as growing objects requires using a multisegment approach from the start. It also implies that most programs will need to include segment numbers in addresses.

The displacement portion of an address is the displacement within a segment and may be specified using one of four addressing modes. Each addressing mode is composed of a base address and an index; either component may be indirect, i.e., the address contains the value of the base or index. Indirect index values are scaled according to the byte size of the object being accessed.

The iAPX-432 is unique among architectures in its support for multiprocessing. Multiprocessing is supported by defining a number of instructions for both processor and process intercommunication and by the interconnect bus. The interconnect bus is a packet bus that allows multiple processors to

be connected. The bus offers up to a 16 Mbyte bandwidth when the packets are the maximum size. Data from memory can be 1–10 bytes in length per access.

The process communication instructions include operations to send and receive messages, as well as conditional send and receive. Operations that send to processors as well as broadcasting to all processors are supported. Since these operations are supported by the architecture and the bus provides communication of these messages, multiprocessing can be performed independently of the process count, processor count, or distribution of processes on processors. However, the single bus provides a limit on the ability to do multiprocessing; the current bus design for the iAPX-432 can handle approximately three processors without undue bus contention. Peripheral chips have been developed to allow multiple buses to be incorporated into a design.

D. The Motorola 68000

The 68000 [54], [55] represents the first microprocessor to support a large, uniform (i.e., unsegmented), virtual addressing space ($>2^{16}$ bytes) and complete support for a 32-bit data type. The MC68000 architecture has many things in common with the PDP-11 architecture. It offers a number of addressing modes and features orthogonality between instructions and addressing modes for many but not nearly all instructions (as compared to a VAX). The MC68000 is a 16-bit implementation, but almost all the instructions support 32-bit data.

Some interesting compromises were made in the MC68000 architecture. Possibly the most obvious is the partitioning of the 16 general purpose registers into two sets: address and data registers. For the compiler this partitioning is troublesome since most addressing modes require the use of an address register and most arithmetic instructions use data registers. Because of this dichotomy, excess register copies are required and the number of data registers is too small to allow register allocation to be easily done. Because the split lowers the number of bits needed for a register designator from four to three bits, this choice is motivated by the instruction coding.

For the most part the addressing modes of the MC68000 follow those of the PDP-11: the major change is the elimination of the infrequently used indirect modes and their replacement with an indexed mode that computes the effective address as the sum of the contents of two registers plus an offset. The MC68000 is a one and a half address machine: instructions have a source and a source/destination specifier and only one of these may be a memory operand. The major exception is the move instruction that can move between two arbitrary operands.

One interesting new instruction in the MC68000 is "check register against bounds." This instruction checks a register contents against an arbitrary upper bound and causes a trap if the contents exceeds the upper bound. If the register contents is a zero-based array index, then this instruction can be used to do the upper array bound check and trap if the bound is exceeded. The MC68000 also obtains reasonably high code density due to its useful addressing modes, a good match between instructions and compiled code, and its support for a wide variety of immediate data. Besides having immediate addressing formats for byte, word, and long word data types, many of the arithmetic and logical instructions allow a short immediate constant ($1 \cdots 8$) as an operand. This combination of immediate data types and the short immediate (quick) format helps increase code density substantially.

The MC68000 made two instruction set additions that help support high level languages. Support for procedure linkage was built in with several instructions; the most important addition was the link instruction, which can be used to set up and maintain activation records. The multiple register move instruction helps shorten the save/restore sequence during a call or return.

Since the original MC68000 has been announced two important new versions of the architecture have been produced. The MC68010 provides support for demand paging by providing instruction restartability in the event of a page fault. The three year delay between the original MC68000 and the MC68010 is a good indication of the complexity of this capability. The recently announced MC68020 provides some extensions to the instruction set, but more importantly represents a 32-bit implementation both internally in the chip and externally on the pins. This provides important performance improvements in instruction access and 32-bit data memory access.

E. The DEC VLSI-Based VAX Processors

There are now three VLSI-based implementations of the VAX architecture. They differ in chip count, amount of custom silicon, and performance. All three implementations are interesting because they reflect different design compromises needed to put the large instruction set into a chip-based implementation. The first implementation, the MicroVAX-I, uses a custom data path chip [9] and keeps the microcode and microsequencer off chip. The second implementation is the VLSI VAX [23], a nine-chip set that implements the full VAX instruction set. The third VLSI-based VAX, the MicroVAX-32 [24], is a single chip that implements a subset of the VAX architecture in hardware.

Several key features characterize the VAX instruction set and help provide organization for the 304 instructions and tens of thousands of combinations of instructions and addressing modes:

- a large number of instructions with nearly complete orthogonality among opcode, addressing mode, and data type;
- support for bytes, words (16 bits), and long words (32 bits) as data types. Special instructions for bit data types;
- many high level instructions including procedure call and return, string instructions, and instructions for floating point and decimal arithmetic; and
- a large number of addressing modes, summarized in Table IV.

The table gives the frequency as percent of all operand memory addressing; the notation (R) indicates the contents of register R. These memory addressing modes represent just less than one-half of the operands. The other operands are register and literal operands. The VAX supports a short literal mode (5 bits) and an immediate mode (defined as PC-relative followed by an autoincrement of the PC). Several common operand addressing formats are obtained using PC-relative addressing since the PC is in the register set. Hence PC-

TABLE IV
SUMMARY OF VAX ADDRESSING MODES

Addressing Mode	Form	Effective Address	Frequency
Register Deferred	(Rn)	(Rn)	7.7%
Autodecrement	-(Rn)	(Rn) - size of operand	0.7%
Autoincrement	(Rn)+	(Rn) Rn := (Rn) + size of operand	6.1%
Autoincrement Deferred	@(Rn)+	((Rn)) Rn := (Rn) + size of operand	0.2%
Byte,Word,Long Displacement	D(Rn)	D+(Rn) D is byte, word, longword	23.8%
Byte,Word,Long Displacement, Deferred	@D(Rn)	(D+(Rn)) D is byte, word, longword	1.2%
Index	base[Rn]	base is addr. mode the address is base + Rn*size of operand	5.3
Total			45%

relative and absolute addressing, as well as immediate addressing, are all done with standard addressing modes using the PC as the register operand.

The MicroVAX-I is not a self-contained VLSI processor since only the data path is integrated. The rest of the processor (including the microcode sequencing, the microcode, and instruction fetch unit) are implemented with standard MSI and LSI parts. The data path was designed to support the VAX architecture and improves upon the structure used in the VAX 11/730 implementation. The advantages of the custom data path are that it consumes far less space and power than a discrete implementation, and it yields higher performance. This performance advantage comes from the tailoring of the data path to the needs of the VAX architecture, as opposed to using off-the-shelf components, which results in a less than optimal implementation of the data path. This tailoring consists primarily of several improvements to the match between the data path and the architecture; these include

1) the ability to handle registers as byte, word, and long word quantities;

2) the ability to read two 32-bit registers in parallel and send them either to the ALU or the barrel shifter in a single cycle;

3) automatic back-up of registers that might be affected by autoincrement and autodecrement addressing modes; and

4) better support for multiply operations.

By limiting the use of custom silicon to the portion of the processor where it most effective and to a point where the design complexity could be handled, the MicroVAX-I achieved its design goals. Performance exceeds that of a VAX 11/730 and the design and implementation time was kept under one year [9].

Two new implementations of the VAX architecture use primarily custom VLSI chips. The first of these, the VLSI VAX, uses a nine-chip set to implement a version of the architecture comparable in performance to a VAX 11/780 CPU. These nine chips include most of the CPU functions, including memory mapping and cache control. The nine-chip set contains about 1.25 million transistors and consists of five different cutsom chips.

1) The instruction fetch/execution chip that performs instruction fetch and decode, ALU operations, and address translation using a small on-chip translation lookaside buffer (TLB).

2) The memory subsystem chip holds a larger TLB, the tag array and control for a 2 KW cache, and performs additional peripheral control functions.

3) A floating point accelerator chip uses an 81b-wide data path and a 100 ns cycle time to provide floating point speeds comparable to those on a 780.

4) The 480K bits of microprogram are stored in five patchable control store chips. Each chip contains nine bits of each 40 byte control word. The amount of microcode is comparable to the MSI based 11/780, 11/750, and 11/730 implementations.

5) The CPU uses a custom bus interface chip [56] to couple to a high speed external system bus.

The single chip implementation of the VAX-11 architecture [24], the MicroVAX-32, uses a single chip to implement a subset of the VAX instruction set including support for memory mapping. When operated with a 20 MHz clock, it is about 20 percent slower than the VLSI VAX in performance. The chip supports 6 of the 12 VAX data types and all 21 addressing modes. Only a subset of instructions are supported on the chip; the breakdown is as follows:

- 175 instructions are supported in the processor's hardware;
- the 70 floating point instructions are supported only with the addition of the floating point chip; and
- 59 instructions (including, e.g., instructions for the less heavily used data formats) are trapped by the processor and interpreted in macrocode.

Interestingly, the 58 percent of the instructions implemented in the processor require only 15 percent of the microcode of a full VAX implementation and constitute 98 percent of the most frequently executed instructions for typical benchmarks (ignoring floating point). A few instructions in the VAX have low utilization (1–2 percent) but long execution times [7] that inflate the effect of the instruction in determining program execution time. By including these instructions in the hardware implementation, the execution time effects of only implementing 58 percent of the instruction set are negligible.

Both VLSI-based VAX processors are implemented on a two-level metal, 3 μm drawn, nMOS process with four implants. First silicon for both processors was completed in February 1984. The design tradeoffs made in the MicroVAX-32 are in strong contrast to the ambitious design of the 9-chip VLSI VAX. Table V clearly shows the dramatic reductions in size and complexity of the implementation accomplished by the subsetting of the architecture that was used in the MicroVAX-32. The less than 20 percent performance impact makes it an effective technique and calls into doubt the need for the software-based part of the instruction set to be defined in the architecture.

VII. ORGANIZATION AND IMPLEMENTATION ISSUES

The interaction between a processor architecture and its organization has always had a profound influence on the cost-performance ratios attainable for the architecture. In VLSI this effect is extended through to low levels of the implementation. To explain some of these interactions and

TABLE V
SUMMARY COMPARISON OF THE VAX MICROPROCESSORS

	VLSI VAX	MicroVAX-32
Chip count (incl. floating pt.)	9	2
Microcode (bits)	480K	64K
Transistors	1250K	101K
TLB	5 entry mini-TLB 512 entries off chip	8 entry fully assoc.
Cache	Yes	No, instruction prefetch buffer

tradeoffs, we have used examples from the MIPS processor. Although the examples are specific to that processor, the issues that they illustrate are common to most VLSI processor designs.

A. Organizational Techniques

Many of the techniques used to obtain high performance in conventional processor designs are applicable to VLSI processors. Some changes in these approaches have been made due to the implementation technology; some of these changes have been adapted into designs for non-VLSI processors. We will look at the motivating influences at the organizational level and then look at pipelining and instruction unit design.

MOS offers the designer a technology that sacrifices speed to obtain very high densities. Although switching time is somewhat slower than in bipolar technologies, communication speed has more effect on the organization and implementation. The organization of an architecture in MOS must attempt to exploit the density of the technology by favoring local computation to global communication.

1) Pipelining: A classical technique for enhancing the performance of a processor is pipelining. Pipelining increases performance by a factor determined by the depth of the pipeline: if the maximum rate at which operators can be executed is r, then pipelining to a depth of d provides an *idealized* execution rate of $r \times d$. Since the speed with which individual operations can be executed is limited, this approach is an excellent technique to enhance performance in MOS.

The depth of the pipeline is an idealized performance multiplier. Several factors prevent achievement of this increase. First, delays are introduced whenever data needed to execute an instruction is still in the pipeline. Second, pipeline breaks occur because of branches. A branch requires that the processor calculate the effective destination of the branch and fetch that instruction; for conditional branches, it is impossible to do this without delaying the pipe for at least one stage (unless both successors of the branch instruction are fetched, or the branch outcome is correctly predicted). Conditional branches may cause further delays because they require the calculation of the condition, as well as the target address. For most programs and implementations, pipeline breaks due to branches are the most serious cause of degraded pipeline performance. Third, the complexity of managing the pipeline and handling breaks adds additional overhead to the basic logic, causing a degradation in the rate at which pipestages can be executed.

The designer, in an attempt to maximize performance, might increase the number of pipestages per instruction; this meets with two problems. First, not all instructions will contain the same number of pipestages. Many instructions, in particular the simpler ones, fit best in pipelines of length two, three, or four, at most. On average, longer pipelines will waste a number of cycles equal to the difference between the number of stages in the pipeline and the average number of stages per instruction. This might lead one to conclude that more complex instructions that could use more pipestages would be more effective. However, this potential advantage is negated by the two other problems: branch frequency and operand hazards.

The frequency of branches in compiled code limits the length of the pipeline since it determines the average number of instructions that occurs before the pipeline must be flushed. This number of course depends on the instruction set. Measurements of the VAX taken by Clark [7] have shown an average of three instructions are executed between every *taken* branch. For simplicity, we call any instruction that changes the program counter (not including incrementing it to obtain the next sequential instruction) a taken branch. Measurements on the Pascal DEL architecture Adept have turned up even shorter runs between branches. Branches that are not taken may also cause a delay in the pipeline since the instructions following the branch may not change the machine state before the branch condition has been determined, unless such changes can be undone if the branch is taken.

Similar measurements for more streamlined architectures such as MIPS and the 801 have shown that branches (both taken and untaken) occupy 15–20 percent of the dynamic instruction mix. When the levels of the instruction set are accounted for and some special anomalies that increase the VAX branch frequency are eliminated, the VAX and streamlined machine numbers are equivalent. This should be the case: if no architectural anomalies that introduce branches exist, the branch frequency will reflect that in the source language programs. The number of operations (not instructions) between branches is independent of the instruction set. This number, often called the run length, and the ability to pipeline individual instructions should determine the optimal choice for the depth of the pipeline. Since more complex instruction sets have shorter run lengths, pipelining across instruction boundaries is less productive.

The streamlined VLSI processor designs have taken novel approaches to the control of the pipeline and attempted to improve the utilization of the pipeline by lowering the frequency of pipeline breaks. The RISC and MIPS processor have only delayed branches; thus, a pipeline break on a branch only occurs when the compiler can not find useful instructions to execute during the stages that are needed to determine the branch address, test the branch condition, and prefetch the destination if the branch is taken. Measurements have found that these branch delays can be effectively used in 80–90 percent of the cases [15]. In fact, measurements of MIPS' benchmarks have shown that almost 20 percent of the instructions executed by the processor occur during a branch delay slot! The 801 offers both delayed and nondelayed branches; the latter allow the processor to avoid inserting a no-op when a useful instruction cannot be found. This delayed branch approach is an interesting contrast to the branch prediction and multiple target fetch techniques used on high-

end machines. The delayed branch approach offers performance that is nearly as good as the more sophisticated approaches and does not consume any silicon area.

A stall in the pipeline caused by an instruction with an operand that is not yet available is called a data or operand hazard. MIPS, the 801 and some larger machines, such as the Cray-1, include pipeline scheduling as a process done by the compiler. This scheduling can be completely done for operations with deterministic execution times (such as most register–register operations) and be optimistically scheduled for operations whose execution time is indeterminate (such as memory references in a system with a cache). This optimization typically provides improvements in the 5–10 percent range. In MIPS, this improvement is compounded by the increase in execution rate achieved by simplifying the pipeline hardware when the interlocks are eliminated for register–register operations. Dealing with indeterminate occurrences, such as cache misses, requires stopping the pipeline. The algorithms used for scheduling the MIPS pipeline are discussed in [49]; Sites describes the scheduling process for the Cray-I in [57].

Because the code sequences between branches are often short, it is often impossible for either the compiler or the hardware to reduce the effects of data dependencies between instructions in the sequence. There are simply not enough unrelated instructions in many segments to keep the pipeline busy executing interleaved and unrelated sequences of instructions. In such cases, neither a pipeline scheduling technique nor a sophisticated pipeline that allows instructions to execute out-of-order can find useful instructions to execute.

Operand hazards cause more difficulty for architectures with powerful instructions and shorter run lengths. When no pipeline scheduling is being done, the dependence between adjacent instructions is high. When scheduling is used it may be ineffective since the small number of instructions between basic blocks makes it difficult to find useful instructions to place between two interdependent instructions.

Another approach to migrating the effect of operand hazards and increasing pipeline performance is to allow *out-of-order instruction execution*. In the most straightforward scenario, the processor keeps a buffer of sequential instructions (up to and including a branch) and examines each of the instructions in parallel to decide if it is ready to execute. An instruction is executed as soon as its operands are available. In most implementations, instructions also complete out-of-order. The alternative is to buffer the results of an instruction, until all the previous instructions have completed; this becomes complex, especially if an instruction can have results longer than a word (since as a block move instruction). Out-of-order completion leads to a fundamental problem: imprecise interrupts. An *imprecise interrupt* occurs when a program is interrupted at an instruction that does not serve as a clean boundary between completed and uncompleted instructions; that is, some of the instructions before the interrupted instruction may not have completed and some of the instructions after the interrupted instruction may have been completed. Continuing execution of a program after an imprecise interrupt is nearly impossible; at best, to continue requires extensive analysis of the executing code segment and simulation of the uncompleted instructions to create a precise interrupt location. Imprecise interrupts can be largely avoided by choosing the instruction to interrupt as the successor of the last (in the sequence) that has completed; this will guarantee that no completed instructions follow the interrupted instruction. This approach has some performance penalty on interrupt speed and prohibits interrupts that can not be scheduled, such as page faults. Because the occurrence of a page fault is not known until the instruction execution is attempted, imprecise interrupts cannot be tolerated on a processor that allows demand paging. This fundamental incompatibility has limited the use of out-of-order instruction issue and completion to high performance machines.

2) Instruction Fetch and Decode: One goal of pipelining is to approach as closely as possible the target of one instruction execution every clock cycle. For most instructions, this can be achieved in the execution unit of the machine. Long running instructions like floating point will take more time, but they can often be effectively pipelined within the execution box. More serious bottlenecks exist in the instruction unit.

As we discussed in an earlier segment, densely encoded instruction sets with multiple instruction lengths lower memory bandwidth but suffer a performance penalty during fetch and decode of the instructions. This penalty comes from the inability to decode the entire instruction in parallel due to the large number of possible interpretations of instruction fields and interdependencies among the fields. This penalty is serious for two reasons. First, it cannot be pipelined away. High level instruction sets have very short sequences between branches (due to the high level nature of the instruction set). Thus, the processor must keep the number of pipestages devoted to instruction fetch and decode to as near to one as possible. If more stages are devoted to this function, the processor will often have idle pipestages. This lack of ability to pipeline high level instruction sets has been observed for the DEL architecture Adept [30]. Note that the penalty will be seen both at instruction prefetch and instruction decode; both phases are made more complex by multiple instruction lengths.

The second reason is that most instructions that are executed are still simple instructions. The most common instructions for VAX, PDP-11, and S/370 style architectures are MOV and simple ALU instructions, combined with "register" and "register with byte displacement" addressing for the operands. Thus, the cost of the fetch and decode can often be as high or even higher than the execution cost. The complexities of instruction decoding can also cause the simple, short instructions to suffer a penalty. For example, on the VAX 11/780 register–register operands take two cycles to complete, although only one cycle is required for the data path to execute the operation. Half the cycle time is spent in fetch and decode; similar results can be found for DEL machines. In contrast, MIPS takes one third of the total cycle time of each instruction for fetch and decode. A processor can achieve single-cycle execution for the simple instructions in a complex architecture, but to do so requires very careful

design and an instruction encoding that simplifies fetch and decode for such instructions.

B. Control Unit Design

The structure of the control unit on a VLSI processor most clearly reflects the make-up of the instruction set. For example, streamlined architectures usually employ a single cycle decode because the simplicity of the instruction set allows the instruction contents to be decoded in parallel. Even in such a machine, a multistate microengine is needed to run the pipeline and control the processor during unpredictable events that cause significant change in the processor states, such as interrupts and page faults. However, the microengine does not participate in either instruction decoding or execution except to dictate the sequencing of pipestages. In a more complex architecture, the microcode must deal both with instruction sequencing and the handling of exceptional events. The cascading of logic needed to decode a complex instruction slows down the decode time, which impacts performance when the control unit is in the critical path. Since decoding is usually done with PLA's, ROM's, or similar programmable structures, substantial delays can be incurred communicating between these structures and in the logic delays within the structures, which themselves are usually clocked.

In addition to the instruction fetch and decode unit, the instruction set and system architecture has a profound effect on the design of the master control unit. This unit is responsible for managing the major cycles of the processor, including initiating normal processor instruction cycles under usual conditions and handling exceptional conditions (page faults, interrupts, cache misses, internal faults, etc.) when they arise. The difficult component of this task is in handling exceptional conditions that require the intervention of the operating system; the process typically involves shutting down the execution of the normal instruction stream, saving the state of execution, and transferring to supervisor level code to save user state and begin handling the condition. Simpler conditions, such as a cache miss or DMA cycle, require only that the processor delay its normal cycle.

Exceptional conditions that require the interruption of execution during an instruction have a significant effect on the implementation. Two distinct types of problems arise: state saving and partially completed instructions. To allow processing of the interrupt, execution of the current instruction stream must be stopped and the machine state must be saved. In a machine with multicycle instructions, some of the internal instruction state may not be visible to user-level instructions. Forcing such state to be visible is often unworkable since the exact amount of state depends on the implementation. Defining such state in the instruction set locks in a particular implementation of the instruction. Thus, the processor must include microcode to save and restore the state of the partially executed instruction. To avoid this problem, some architectures force instructions that execute for a comparatively long time and generate results throughout the instruction, to employ user visible registers for their operation; most architectures that support long string instructions use just this approach. For example, on the S/370 long string instructions use the general purpose registers to hold the state of the instruction during execution; shorter instructions, such as Move Character (MVC), inhibit interrupts during execution. Because the MVC instruction can still access multiple memory words, the processor must first check to ensure that no page faults will occur before beginning instruction execution.

Instructions that do not have very long running times can be dealt with by a two-part strategy. The architecture may prohibit most interrupts during the execution of the instruction. For those interrupts that cannot be prohibited, e.g., a page fault in the executing instruction, the architecture can stop the execution of instruction, process the interrupt, and restart the instruction. This process is reasonably straightforward, except when the instruction is permitted to alter the state of the processor before completion of the instruction without interrupt can be guaranteed. If such changes are allowed, then the implementation must either continue the instruction in the middle, or restore the state of the processor before restarting the instruction. Neither of these approaches is particularly attractive since they require either special hardware support, or extensive examination of the executing instruction. If the processor can decode the instruction and knows how much of the instruction was completed, the microcontrol could simulate the completion of the instruction, or (under most cases) undo the effect of the completed portions. However, both of these approaches incur substantial overhead for determining the exact state of the partially executed instruction, and taking the remedial action. Additionally, some classes of instructions may not be undone; for example, an instruction component that clears a register cannot be reversed, without saving the contents of the register. Since this overhead must be taken on common types of interrupts, such as page faults, this solution is not attractive.

To circumvent these problems, the architecture must either prohibit such instructions, as streamlined architectures do, or provide hardware assist. To keep the amount of special hardware assistance needed within bounds, only limited types of changes in the states are allowed before guaranteed completion. The most common example of such a limited feature is autoincrement/autodecrement addressing modes. Like most instructions that change state midway through the instruction, only the general purpose registers can be changed. This offers an opportunity to try to restore the machine state to its state prior to instruction execution.

Let us consider the possibilities that occur on the VAX. The most obvious scheme would be to decode the faulting instruction and unwind its effect by inverting the increment or decrement (which can only change the register contents by a fixed constant). However, on the VAX, with up to five operands per instruction, decoding the faulting instruction and determining which registers have been changed is a major undertaking. Because the instruction cannot be restarted until all values that have been altered are restored, the cost would be prohibitive. The solution used for the MVC instruction on the S/370—make sure you can complete the instruction before you start it—can be adapted. Because of

the possibility of page faults, this approach requires that the instruction be simulated to determine that all the pages accessed by the instruction's operands are in memory. This could be quite expensive, especially if the addressing mode is used often. Because only limited modifications to the processor state are allowed before instruction completion, there are several hardware-based solutions that have smaller impacts on performance.

1) Save the register contents before they are changed, along with the register designator. Restore all the saved registers using their designator when an interrupt occurs.

2) Save the register designator and the amount of the increment or decrement (in the range of 1–4 on the VAX). If an interrupt occurs, compute the original value of the registers corresponding to the saved designators and constants.

3) Compute the altered register value, but do not store it into the register until the end of the instruction execution.

The above list gives the rough order of the hardware complexity of these solutions. The last solution is complicated because a list of changed registers and the register numbers must be kept until the instruction ends. It is also the least efficient solution; since most instructions do not fault, the cost of the update must be added to the execution time. The second solution is simpler and requires the least storage, but it still requires some decoding overhead. The first solution is the simplest; it can be implemented by saving the registers as they are read for incrementing/decrementing.

C. Data Path Design

The data paths of most VLSI processors share many common features since most instruction sets require a small number of basic micro-operations. Special features may be included to support structures such as the queue that saves altered registers during instruction execution.

The main data path of the processor is usually distinguished by the presence of two or more buses, serving as a common communication link among the components of the bus. Many common components may be associated with smaller, auxiliary data paths because they do not need frequent or time-critical access to the resources provided by the main data path or for performance reasons, which we will discuss shortly.

The data path commonly includes the following components.

- A register file for the processor's general purpose registers and any other registers included in the main data path for performance. In a microprogrammed machine, temporary registers used by the microcode may reside here. The function of the register file depends on the instruction set. In some cases, it is removed from the data path for reasons we will discuss shortly.
- An ALU providing both addition/subtraction and some collection of logical operations, and perhaps providing support for multiplication and division. We will discuss the design of the ALU in some more detail shortly.
- A shifter used to implement shifts and rotates and to implement bit string instructions or assist instruction decoding. Some processors include a barrel shifter (rather than a single-bit shifter) because although they consume a fair amount of area, they dramatically increase the speed of multiple-bit shifts.
- The program counter. Positioning the program counter in the main data path simplifies calculation of PC-based displacements. In a high performance or pipelined processor, the program counter will usually have its own incrementer. This allows both faster calculation of the next sequential instruction address and overlap of PC increment with ALU operation. A pipelined processor will often have multiple PC registers to simplify state saving and returning from interrupts.

These are the primary components of the data path; microarchitectures may have special features designed to improve the performance of some particular part of the instruction set. Fig. 4 shows the data path from the MIPS processor. It is typical of the data path designs found on many VLSI processors. Some data paths are simpler (e.g., the RISC data path, ignoring the register stack) and some are more complicated (e.g., the VAX data path). Although the basic components are common, the communication paths are often customized to the needs of the instruction set and varying speed, space, and power tradeoffs may made in designing the data path components (e.g., a ripple carry adder versus a carry lookahead adder).

1) Data Bus Design: The minimum machine cycle time is limited by the time needed to move data from one resource to another in the data path. This delay consists of the propagation time on the control wires and the propagation time on the data buses, which are usually longer than the control lines. In a process with only one level of low resistance interconnect (metal) the data bus would be run in metal, while the control lines would run in polysilicon. The delay on the control lines can be reduced by minimizing the pitch in the data path. Partly because of these delays, almost all data paths in VLSI processors use a two bus design. The extra delays due to the wide data path pitch in a three bus design may not be compensated for by the extra throughput available on the third bus.

Power constraints and the need to communicate signals as quickly as possible across the data path lead to heavy use of bootstrapped control drivers. Large numbers of bootstrap drivers put a considerable load on clock signals, and the designer must be careful to avoid skew problems by routing clocks in metal and using low resistance crossovers. Bootstrap drivers require a setup period and cannot be used when a control signal is active on both clock phases. Static superbuffers can be used in such cases, but they have a much higher static power usage. The tight pitch and use of bootstrap drivers helps minimize the control delay time. In MIPS the tight pitch (33 λ) and the extensive use of dynamic bootstrap drivers holds the control delay to 10 ns.

Although reducing the control communication delays is important, the main bus delays normally constitute a much larger portion of the processor cycle time. The main reason for this is that the bus delay is proportional to the product of the bus capacitance and the voltage swing divided by the driver size. When the number of drivers on the bus gets large (25–50, or more), the bus capacitance is dominated by the drivers themselves, i.e., it is proportional to driver size times the driver count. Thus, the bus delay becomes proportional to the product of the driver count and the voltage swing!

This delay can be reduced by either lowering the number of drivers or by reducing the voltage swing. For many data

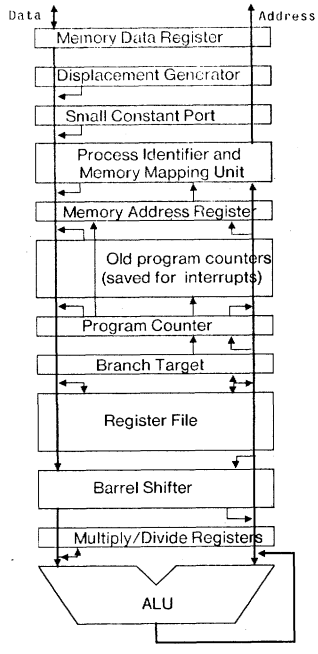

Fig. 4. MIPS data path block diagram.

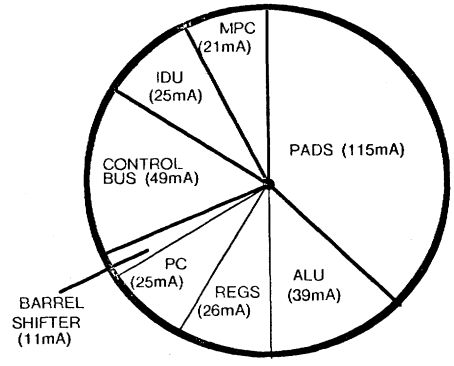

Fig. 5. MIPS current distribution.

paths the register file is the major source of bus drivers. Those bus drivers are directly responsible for a slower clock cycle. This penalty on processor cycle time is a major drawback for a large register file implemented in MOS technology. To partially overcome this problem, many processor designs implement the register file as a small RAM off of the data bus. Although this eliminates a large fraction of the load from drivers, it may introduce several other problems. The register file is usually a multiported device for at least reads and sometimes for writes. The smallest RAM cell designs may not provide this capability. Thus, maintaining the same level of performance requires operating the RAM at a higher speed or duplicating the RAM to increase bandwidth (a typical technique for high performance ECL machines). Isolating the RAM or register file from the bus may also incur extra delays due to communication time or the presence of latches between the registers and the bus.

Another approach is to try to reduce the switching time of the bus by circuit design techniques. There are three major styles of bus design that can be used:

- a nonprecharged rail-to-rail bus which has the above stated problem;
- a precharged bus which reduces the problem by replacing the slower pull-up time but having the same the pull-down time. Precharging requires a separate idle bus cycle to charge the bus to the high state; and
- a limited voltage-swing bus that still allows a bus active on every clock cycle.

The use of precharged buses is discussed in many introductory texts on VLSI design [12]. Precharging is most useful in a design when the bus is idle every other cycle due to the organization of the processor. For example, if the ALU cycle time is comparatively long and the processor is otherwise idle during that time, the ALU can be isolated from the bus, and the precharge can occur during that cycle. When such idle cycles are not present in the global timing strategy, the attraction of precharging vanishes. The limited swing bus uses an approach similar to the techniques used in dynamic RAM design [58]. The bus is clamped to reduce its voltage swing and sense-amplifier-like circuits are used to detect the change in voltage. A version of MIPS was fabricated using a clamped bus structure to reduce the effective voltage swing by about a factor of 4. This approach was the most attractive, since MIPS uses the bus on every clock phase. The use of a restricted voltage swing does require careful circuit design since important margins, such as noise immunity, may be reduced.

2) The Data Path ALU: Arithmetic operations are often in a processor's critical timing paths and thus require careful logic and circuit design. Although some designs use straightforward Manchester-carry adders and universal logic blocks (see, e.g., the description of the OM2 [12]), more powerful techniques are needed to achieve high performance. Since the addition circuitry is usually the critical path, it can be separated from the logic operation unit to achieve minimal loading on the adder. A fast adder will need to use carry lookahead, carry bypass, or carry select. For example, MIPS uses a full carry-lookahead tree, with propagate signals and generate signals produced for each pair of bits, which results in a total ALU delay of less than 80-ns with a one-level metal 3 μm process. To obtain high speed addition, the ALU may also consume a substantial portion of the processor's power budget.

Supporting integer multiply and divide (and the floating point versions) with reasonable performance can provide a real challenge to the designer. One approach is to code these operations out of simpler instructions, using Booth's algorithm. This will result in multiply or divide performance at the rate of approximately one bit per every three or four instructions. The RISC processor uses this approach. Most microprocessors implement multiply/divide via microcode using either individual shift and add operations or relying on special support for executing Booth's algorithm. The 68000 uses this approach. MIPS implements special instructions for doing steps of a multiply or divide operation; these instructions are used to expand the macros for a 32-bit multiply or divide, into a sequence of 8 or 16 instructions, respectively. This type of support, similar to that used in the 68000 microengine, requires the ability to do an add (depending on the low-order bits of the register) and a shift in the same instruction step. Limited silicon area and power bud-

gets often make it impractical to include hardware for more parallel multiplication on the CPU chip.

Fast arithmetic operations can be supported in a coprocessor that does both integer and floating point operations as in the VLSI VAX. The design of a floating point coprocessor that achieves high performance for floating point operations can be extremely difficult. The coprocessor design must be taken into account in the design of the main CPU as well as in the software for the floating point routines. An inefficient or ineffective coprocessor interface will mean that the coprocessor does not perform as well as an integral floating point unit. Many current microprocessors exhibit this property: they execute integer operations at a rate close to that of a minicomputer, but are substantially slower on floating point instructions. Furthermore, the floating point instruction time is often dominated by communication and coordination with the coprocessor, not by the time for the arithmetic operation. A well-designed floating point coprocessor, such as the floating point processor for the VLSI VAX, can achieve performance equal to the performance obtained in an integral floating point unit.

3) The Package Constraint: Packaging introduces pin limitations and power constraints. Limited pins force the designer to choose his functional boundaries to minimize interconnection. Pin multiplexing can partially relieve the pin constraints, but it costs time, especially when the pins are frequently active.

Two types of power constraints exist: total static power and package inductance. The packaging technology defines the maximum static power the chip may consume. Because power can eliminate delays in the critical path, the power budget must be used carefully. Typical packages for processors with more than 64 pins can dissipate 2–3 W.

The problem of package inductance [59] is more subtle and can be difficult to overcome. Suppose the processor drives a large number of pins simultaneously, e.g., 32 data and 32 address pins, then the current required to drive the pins can be temporarily quite large. In such cases the package inductance (due largely to the power leads between the die and the package) can lead to a transient in the on-chip power supply voltage. This problem can be mitigated by using multiple power and ground wires or by more sophisticated die bonding and packaging technology.

The power distribution plot for MIPS (see Fig. 5) shows how this power budget might be consumed. Power is used for three principle goals in nMOS: to overcome delays due to serial combinations of gates, to reduce communication delays between functional blocks, and to reduce off-chip communication delays. The MIPS power distribution plot shows the major power consumers are

- the ALU with its extensive multilevel logic,
- the pins with the drive logic, and
- the control bus, which provides most of the time-critical interchip communication.

D. Summary

VLSI technology has a fundamental effect on the design decisions made in the architecture and organization of processors. Since pipelining is a basic technique by which VLSI processors achieve performance, the architect and designer must consider a series of issues that affect performance improvements achievable by pipelining. These issues include the suitability of the instruction set for pipelining, the frequency of branches, the ease of decomposing instructions, and the interaction between instructions.

Pipelining adds a major complication to the task of controlling the execution of instructions. The parallel and simultaneous interpretation of multiple instructions dramatically complicates the control unit since it must consider all the ways in which all the instructions under execution can require special control. Complications in the instruction set can make this task overwhelming. In addition to controlling instruction sequencing, the control unit (or its neighbor) often contains the instruction decoding logic. The complexity and size of the decoding logic is influenced by the size and complexity of the instruction set and how the instruction set is encoded. The observation that most microprocessors use 50 percent or more of their limited silicon area for control functions was a consideration when RISC architectures were proposed [60].

Although the high level design of the data path is largely functionally independent of the architecture, the detailed requirements of data path components are affected by the architecture. For example, an architecture with instructions for bytes, half-words, and words requires special support in the register file (to read and write fragments) and in the ALU to detect overflow on small fragments (or to shift smaller data items into the high order bits of the ALU). Although the functionality of most data path components is independent of the processor architecture, the architecture and organization affect the data path design in two important ways. First, different processors will have different critical timing paths, and data path components in the critical path will need to be designed for maximum performance. Second, specific features of the architecture will cause specialization of the data path; examples of this specialization include support for bytes and half words in a register file and the register stack used to handle autoincrement/autodecrement in VAX microprocessors. The role of good implementation is magnified in VLSI where what is obtainable is much broader in range and much more significantly affected by the technology.

VIII. Future Trends

VLSI processor technology combines several different areas: architecture, organization, and implementation technology. Until recently, technology has been the driving force: rapid improvements in density and chip size have made it possible to double the on-chip device count every few years. These improvements have led to both architectural changes (from 8- to 16- to 32-bit data paths, and to larger instruction sets) and organizational changes (incorporation of pipelining and caches). As the technology to implement a full 32-bit processor has become available, architectural issues, rather than implementation concerns, have assumed a larger role in determining what is designed.

A. Architectural Trends

In the past few years many designers have been occupied with exploring the tradeoffs between streamlined and more

complex architectures. Future architectures will probably embrace some combination of both these ideas. Three major areas, parallel processing, support for nonprocedural languages, and more attention to systems-level issues, stand out as foci of future architectures.

Parallel processing is an ideal vehicle for increasing performance using VLSI-based processors. The low-cost of replicating these processors makes a parallel processor attractive as a method for attaining higher performance. However, many unsolved problems still exist in this arena. Another paper in this issue address the development of concurrent processor architectures for VLSI in more detail [61].

Another architectural area that is currently being explored is the architecture of processors for nonprocedural languages, such as Lisp, Smalltalk, and Prolog. There are several important reasons for interest in this area. First, such languages perform less well than procedural languages (Pascal, Fortran, C, etc.) on most architectures. Thus, one goal of the architectural investigations is to determine whether there are significant ways to achieve improved performance for such languages through architectural support. A second important issue is the role of such languages in exploiting parallelism. Many advocates of this class of languages contend that they offer a better route to obtaining parallelism in programs. If efforts to develop parallel processors are successful, then this advantage can be best exploited by supporting the execution of programs in an efficient manner, both for sequential and parallel activities.

Several important VLSI processors have been designed to support this class of languages. The SCHEME chips [13], [62] (called SCHEME-79 and SCHEME-81) are processors designed at MIT to directly execute SCHEME, a statically-scoped variant of Lisp. In addition to direct support for interpreting SCHEME, the SCHEME chips include hardware support for garbage collection (a microcoded garbage collector) and dynamic type checking.

SCHEME-81 includes tag bits to type each data item. The tag specifies whether a word is a datum (e.g., list, integer, etc.), or an instruction. Special support is provided for accessing tags and using them either as opcodes, to be interpreted by the microcode, or as data type specifications, to be checked dynamically when the datum is used. A wide microcode word is used to control multiple sets of register-operator units that function in parallel within the data path. The SCHEME-81 design supports multiple SCHEME processors. The primary mechanism to support multiprocessing is the SBUS. The novel feature of the SBUS is that it provides a protocol to manipulate Lisp structures over the bus.

The SOAR (Smalltalk on a RISC) processor [63] is a chip designed at U.C. Berkeley to support Smalltalk. SOAR provides efficient execution of Smalltalk by concentrating on three key areas. First, SOAR supports the dynamic type checking of tagged objects required by Smalltalk. SOAR handles tagged data by executing instructions and checking the tag in parallel; if both operands are not simple integers, the processor does a trap to a routine for the data type specified by the tag. This makes the frequent case where both tags are integers extremely fast. Second, SOAR provides fast procedure call with a variation of the RISC register windowing scheme and with hardware support to simplify software caching of methods. In Smalltalk, the destination of a procedure call may depend on the argument passed. Caching the method in the instruction stream requires special support for nonreentrant code. Third, SOAR has hardware support for an efficient storage reclamation algorithm, called generation scavenging [64]. To support this technique requires the ability to trap on a small percentage of the store operations (about 0.2 percent). Checking for this infrequent trap condition is done by the SOAR hardware.

The SOAR architecture and implementation shows how the RISC philosophy of building support for the most frequent cases can be extended to a dynamic object-oriented environment. Smalltalk is supported by providing fast and simple ways to handle the most common situations (e.g., integer add) and using traps to routines that handle the exceptional cases. This approach is very different from the Xerox Smalltalk implementations [65], [66] that use a custom instruction set which is heavily encoded and implemented with extensive microcode.

Another major problem facing VLSI processor architects arises as the performance of these architectures approaches mainframe performance. Prior to the most recent processor designs, architects did not have to devote as much attention to systems issues: memory speeds were adequate to keep the processor busy, off-chip memory maps sufficed, and simple bus designs were fast enough for the needs of the processor. As these processors have become faster and have been adopted into complete computers (with large mapped memories and multiple I/O devices), these issues assume increasing importance. VLSI processors will need to be more concerned with the memory system: how it is mapped, what memory hierarchy is available, and the design of special processor-memory paths that can keep the processor's bandwidth requirements satisfied. Likewise, interrupt structure and support for a wide variety of high speed I/O devices will become more important.

B. Organizational Trends

Increasing processor speeds will bring increased need for memory bandwidth. Packaging constraints will make it increasing disadvantageous to obtain that bandwidth from off-chip. Thus, caches will migrate onto the processor chip. Similarly, memory address translation support will also move onto the processor chip. Two important instances of this move can be seen: the Intel iAPX432 includes an address cache, while the Motorola 68020 includes a small (256 byte) instruction cache. Cache memory is an attractive use of silicon because it can directly improve performance and its regularity limits the design effort per unit of silicon area.

Although today's microprocessors are used as CPU's in many computers, much of the functionality required in the CPU is handled off-chip. Many of the required functions not supported on the processor require powerful coprocessors. Among the functions performed by coprocessors, floating point and I/O interfacing are the most common. In the case of floating point, limited on-chip silicon area prevents the integration of a high performance floating point unit unto the chip. For the near future, designers will be faced with the difficult task of choosing what to incorporate on the processor chip. Without the cache both the fixed point and

floating point performance of the processor may suffer. Thus, using a separate coprocessor is a concession to the lack of silicon area. The challenge is to design a coprocessor interface that avoids performance loss due to communication and coordination required between the processor and the coprocessor.

I/O coprocessors allow another processor to be devoted to the detailed control of an I/O device. A separate I/O processor not only eliminates the need for such functionality on the processor chip, it also supports overlapped processing by removing the I/O interface from the set of active tasks to be executed by the processor. As I/O processors become more powerful, they migrate from a coprocessor model to a separate I/O processor that uses DMA and the bus to interface to the memory and main processor.

C. Technology Trends

One of the most fundamental changes in technology is the shift to CMOS as the fabrication technology for VLSI processors [67]. The major advantage of CMOS is the low power consumption: CMOS designs use essentially no static power. This advantage simplifies circuit design and allows designers to use their power budget more effectively to reduce critical paths. Another advantage of CMOS is the absence of ratios in the design of logic structures; this simplifies the design process compared to nMOS design.

The major drawbacks of CMOS are in layout density. These disadvantages come from two factors: logic design and design rule requirements. Static logic designs in CMOS will often require more transistors and hence more connections than the nMOS counterpart. Many designs will also require both a signal and its complement; this increases the wiring space needed for the logic. CMOS designs can also take more space because of well separation rules. The p and n transistor types must be placed into different wells; since the well spacing rules are comparatively large, the separation between transistors of different types must be large. This can lead to cell designs whose density and area are dominated by the well spacing rules.

One important development that will help MOS technologies (but is particularly important for CMOS) is the availability of multiple levels of low resistance interconnect. The larger number of connections in CMOS makes this almost mandatory to avoid dominating layout density by interconnection constraints. A two-level metal process provides another level of interconnect and is the best solution. A silicide process allows the designer access to a low resistance polysilicon layer; this allows polysilicon to be used for longer routes but does not provide an additional layer of interconnect.

The design of faster and larger VLSI processors will require improvements in packaging both to lower delays and to increase the package connectivity. The development of pin grid packages has helped solve both of these problems to a significant extent. Packaging technologies that use wafers with multiple levels of interconnect as a substrate are being developed. These wafer-based packaging technologies provide high density and a large number of connections; they offer an alternative to the multilayer ceramic package.

Two of the biggest areas of unknown opportunity are gallium arsenide (GaAs) and wafer-scale integration. A GaAs medium for integrated circuits offers the advantage of significantly higher switching speeds versus silicon-based integrated circuits [68]. The primary advantage of GaAs comes from increased mobility of electrons, which leads to improvements in transistor switching speed over silicon by about an order of magnitude; furthermore, its power dissipation per gate is similar to nMOS (but still considerably higher than CMOS). Several fundamental problems must be overcome before GaAs becomes a viable technology for a processor. The most mature GaAs processes are for depletion mode MESFETS; logic design with such devices is more complex and consumes more transistors than MOS design. Currently, many problems prevent the fabrication of large ($>10\,000$ transistors) GaAs integrated circuits with acceptable yields. Until these problems are overcome, the advantages of silicon technologies will make them the choice for VLSI processors.

Wafer scale integration allows effective use of silicon and high bandwidth interconnect between blocks on the same wafer. If the blocks represent components similar to individual IC's, their integration on a single wafer yields increased packing density and communication bandwidth because of shorter wires and more connections. Lower total packaging costs are also possible. There are several major hurtles that must be surpassed to make wafer-scale technology suitable for high performance custom VLSI processors. A major problem is to create a design methodology that generates individual testable blocks that will have high yields and that can be selectively interconnected to other working blocks. The need for multiple connection paths among the blocks and the high bandwidth of these connections makes this problem very difficult.

D. Summary

New and future architectural concepts are serving as driving forces for the design of new VLSI processors. Interest in nonprocedural languages leads to the creation of processors such as SCHEME and SOAR that are specifically designed to support such languages. The potential of a parallel processor constructed using VLSI microprocessors is an exciting possibility. The Intel iAPX432 specifically provides for multiprocessing. The importance of this type of system architecture will influence other processors to provide support for multiprocessing.

The increasing performance of VLSI processors will force designers to consider system performance, memory hierarchies, and floating point performance. Systems level products constructed using these processors will require support for memory mapping, interrupts, and high speed I/O. To attain the desired performance goals both on and off chip caches will be needed to reduce the bandwidth demands on main memory. The next generation of VLSI processors will be easily competitive with minicomputers and superminicomputers in integer performance; however, without floating point support they will be much slower than the larger machines with integrated floating point support. High performance floating point is a function both of the available coprocessor hardware for floating point and a low overhead coprocessor interface.

Despite the increased input from the architectural and software directions to the design of VLSI processors, technology remains a powerful driving force. CMOS will bring relief from the power problems associated with large nMOS integrated circuits; the problems presented by CMOS technology are minor compared to its benefits. Steady improvements in packaging technology can be predicted; more radical packaging technologies offer substantial increases in packaging density and interconnection bandwidth.

Wafer-scale integration and GaAs FET's stand as two new technologies that may substantially alter VLSI processor design. Wafer-scale integration offers several benefits but its success will depend on a balanced design methodology that can overcome fabrication defects without substantially impacting performance. GaAs offers very high speed devices; to be useful for large IC's, such as a VLSI CPU, will require major improvements in yield.

IX. Conclusions

A processor architecture supplies the definition of a host environment for applications. The use of high level languages requires that we evaluate the environment defined by the instruction set in terms of its suitability as a target for compilers of these languages. New instruction set designs *must* use measurements based on compiled code to ascertain the effectiveness of certain features.

The architect must trade off the suitability of the feature (as measured by its use in compiled code), against its cost, which is measured by the execution speed in an implementation, the area and power (which help calibrate the opportunity cost), and the overhead imposed on other instructions by the presence of this instruction set feature or collection of features. This approach can be used to measure the effectiveness of support features for the operating system; the designer must consider the frequency of use of such a feature, the performance improvement gained, and the cost of the feature. All three of these measurements must be considered before deciding to include the feature.

The investigation of these tradeoffs has led to two significantly different styles of instruction sets: simplified instruction sets and microcoded instruction sets. These styles of instruction sets have devoted silicon resources to different uses, resulting in different performance tradeoffs. The simplified instruction set architectures use silicon area to implement more on-chip data storage. Processors with more powerful instructions and denser instruction encodings require more control logic to interpret the instructions. This use of available silicon leads to a tradeoff between data and instruction bandwidth: the simplified architectures have lower data bandwidth and higher instruction bandwidth than the microcode-based architectures.

VLSI appears to be the first choice implementation media for many processor architectures. Increased densities and decreased switching times make the technology continuously more competitive. These advantages have motivated designers to use VLSI as the medium to explore new architectures. By combining improvements in technology, better processor organizations, and architectures that are good hosts for high level language programs, a VLSI processor can reach performance levels formerly attainable only by large-scale mainframes.

Acknowledgment

The material in this paper concerning MIPS and several of the figures are due to the collective efforts of the MIPS team: T. Gross, N. Jouppi, S. Przybylski, and C. Rowen; T. Gross and N. Jouppi also made suggestions on an early draft of the paper. M. Katevenis (who supplied the table of instructions for the Berkeley RISC processor) and J. Moussouris also made numerous valuable suggestions from a non-MIPS perspective.

References

[1] *DEC VAX11 Architecture Handbook,* Digital Equipment Corp., Maynard, MA, 1979.

[2] J. Rattner, "Hardware/Software Cooperation in the iAPX 432," in *Proc. Symp. Architectural Support for Programming Languages and Operating Systems,* Ass. Comput. Mach., Palo Alto, CA, Mar. 1982, p. 1.

[3] M. Flynn, "Directions and issues in architecture and language: Language → Architecture → Machine →," *Computer,* vol. 13, no. 10, pp. 5–22, Oct. 1980.

[4] R. Johnnson and D. Wick, "An overview of the Mesa processor architecture," in *Proc. Symp. Architectural Support for Programming Languages and Operating Systems,* Ass. Comput. Mach., Palo Alto, CA, Mar. 1982, pp. 20–29.

[5] M. Hopkins, "A perspective on microcode," in *Proc. COMPCON Spring '83,* IEEE, San Francisco, CA, Mar. 1983, pp. 108–110.

[6] ——, "Compiling high-level functions on low-level machines," in *Proc. Int. Conf. Computer Design,* IEEE, Port Chester, NY, Oct. 1983.

[7] D. Clark and H. Levy, "Measurement and analysis of instruction use in the VAX 11/780," in *Proc. 9th Annu. Symp. Computer Architecture,* ACM/IEEE, Austin, TX, Apr. 1982.

[8] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI processor arrays," in *Introduction to VLSI Systems,* C. A. Mead and L. Conway, Eds. Reading, MA: Addison-Wesley, 1978.

[9] G. Louie, T. Ho, and E. Cheng, "The MicroVAX I data-path chip," *VLSI Design,* vol. 4, no. 8, pp. 14–21, Dec. 1983.

[10] G. Radin, "The 801 minicomputer," in *Proc. SIGARCH/SIGPLAN Symp. Architectural Support for Programming Languages and Operating Systems,* Ass. Comput. Mach., Palo Alto, CA, Mar. 1982, pp. 39–47.

[11] D. A. Patterson and D. R. Ditzel, "The case for the reduced instruction set computer," *Comput. Architecture News,* vol. 8, no. 6, pp. 25–33, Oct. 1980.

[12] C. Mead and L. Conway, *Introduction to VLSI Systems.* Menlo Park, CA: Addison-Wesley, 1980.

[13] J. Holloway, G. Steele, G. Sussman, and A. Bell, "SCHEME-79—LISP on a chip," *Computer,* vol. 14, no. 7, pp. 10–21, July 1981.

[14] R. Sherburne, M. Katevenis, D. Patterson, and C. Sequin, "Local memory in RISCs," in *Proc. Int. Conf. Computer Design,* IEEE, Rye, NY, Oct. 1983, pp. 149–152.

[15] T. R. Gross and J. L. Hennessy, "Optimizing delayed branches," in *Proc. Micro-15,* IEEE, Oct. 1982, pp. 114–120.

[16] W. A. Wulf, "Compilers and computer architecture," *Computer,* vol. 14, no. 7, pp. 41–48, July 1981.

[17] J. Hennessy, "Overview of the Stanford UCode compiler system," Stanford Univ., Stanford, CA.

[18] F. Chow, "A portable, machine-independent global optimizer—design and measurments," Ph.D. dissertation, Stanford Univ., Stanford, CA, 1984.

[19] J. R. Larus, "A comparison of microcode, assembly code, and high level languages on the VAX-11 and RISC-I," *Comput. Architecture News,* vol. 10, no. 5, pp. 10–15, Sept. 1982.

[20] L. J. Shustek, "Analysis and performance of computer instruction sets," Ph.D. dissertation, Stanford University, Stanford, CA, May 1977; also published as SLAC Rep. 205.

[21] D. A. Patterson and C. H. Sequin, "A VLSI RISC," *Computer,* vol. 15, no. 9, pp. 8–22, Sept. 1982.

[22] M. Auslander and M. Hopkins, "An overview of the PL.8 compiler," in *Proc. SIGPLAN Symp. Compiler Construction,* Ass. Comput. Mach., Boston, MA, June 1982, pp. 22–31.

[23] W. Johnson, "A VLSI superminicomputer CPU," in *Dig. 1984 Int. Solid-State Circuits Conf.*, IEEE, San Francisco, CA, Feb. 1984, pp. 174–175.
[24] J. Beck, D. Dobberpuhl, M. Doherty, E. Dornekamp, R. Grondalski, D. Grondalski, K. Henry, M. Miller, R. Supnik, S. Thierauf, and R. Witek, "A 32b microprocessor with on-chip virtual memory management," in *Dig. 1984 Int. Solid-State Circuits Conf.*, IEEE, San Francisco, CA, Feb. 1984, pp. 178–179.
[25] R. Sites, "How to use 1000 registers," in *Proc. 1st Caltech Conf. VLSI*, California Inst. Technol., Pasadena, CA, Jan. 1979.
[26] F. Baskett, "A VLSI Pascal machine," Univ. California, Berkeley, lecture.
[27] D. Ditzel and R. McLellan, "Register allocation for free: The C machine stack cache," in *Proc. Symp. Architectural Support for Programming Languages and Operating Systems*, Ass. Comput. Mach., Palo Alto, CA, Mar. 1982, pp. 48–56.
[28] *The C/70 Macroprogrammer's Handbook*, Bolt, Beranek, and Newman, Inc., Cambridge, MA, 1980.
[29] B. Lampson, "Fast procedure calls," in *Proc. SIGARCH/SIGPLAN Symp. Architectural Support for Programming Languages and Operating Systems*, Ass. Comput. Mach., Mar. 1982, pp. 66–76.
[30] S. Wakefield, "Studies in execution architectures," Ph.D. dissertation, Stanford Univ., Stanford, CA, Jan. 1983.
[31] R. Ragan-Kelly, "Performance of the pyramid computer," in *Proc. COMPCON*, Feb. 1983.
[32] Y. Tamir and C. Sequin, "Strategies for managing the register file in RISC," *IEEE Trans. Comput.*, vol. C-32, no. 11, pp. 977–988, Nov. 1983.
[33] M. Katevenis, "Reduced instruction set computer architectures for VLSI," Ph.D. dissertation, Univ. California, Berkeley, Oct. 1983.
[34] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation by coloring," IBM Watson Research Center, Res. Rep. 8395, 1981.
[35] B. Leverett, "Register allocation in optimizing compilers," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, Feb. 1981.
[36] F. C. Chow and J. L. Hennessy, "Register allocation by priority-based coloring," in *Proc. 1984 Compiler Construction Conf.*, Ass. Comput. Mach., Montreal, P.Q., Canada, June 1984.
[37] A. J. Smith, "Cache memories," *Ass. Comput. Mach. Comput. Surveys*, vol. 14, no. 3, pp. 473–530, Sept. 1982.
[38] D. Clark, "Cache Performance in the VAX 11/780," *ACM Trans. Comput. Syst.*, vol. 1, no. 1, pp. 24–37, Feb. 1983.
[39] M. Easton and R. Fagin, "Cold start vs. warm start miss ratios," *Commun. Ass. Comput. Mach.*, vol. 21, no. 10, pp. 866–872, Oct. 1978.
[40] D. Clark and J. Emer, "Performance of the VAX-11/780 translation buffer," to be published.
[41] R. Fabry, "Capability based addressing," *Commun. Ass. Comput. Mach.*, vol. 17, no. 7, pp. 403–412, July 1974.
[42] W. Wulf, R. Levin, and S. Harbinson, *Hydra:C.mmp: An Experimental Computer System*. New York: McGraw-Hill, 1981.
[43] M. Wilkes and R. Needham, *The Cambridge CAP Computer and Its Operating System*. New York: North Holland, 1979.
[44] M. Wilkes, "Hardware support for memory management functions," in *Proc. SIGARCH/SIGPLAN Symp. Architectural Support for Programming Languages and Operating Systems*, Ass. Comput. Mach., Mar. 1982, pp. 107–116.
[45] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, and T. Gross, "Design of a high performance VLSI processor," in *Proc. 3rd Caltech Conf. VLSI*, California Inst. Technol., Pasadena, CA, Mar. 1983, pp. 33–54.
[46] S. Przybylski, T. Gross, J. Hennessy, N. Jouppi, and C. Rowen, "Organization and VLSI implementation of MIPS," *J. VLSI Comput. Syst.*, vol. 1, no. 3, Spring 1984; see also Tech. Rep. 83-259.
[47] J. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "MIPS: A VLSI processor architecture," in *Proc. CMU Conf. VLSI Systems and Computations*, Rockville, MD: Computer Science Press, Oct. 1981, pp. 337–346; see also Tech. Rep. 82-223.
[48] J. L. Hennessy, N. Jouppi, F. Baskett, T. R. Gross, and J. Gill, "Hardware/software tradeoffs for increased performance," in *Proc. SIGARCH/SIGPLAN Symp. Architectural Support for Programming Languages and Operating Systems*, Ass. Comput. Mach., Palo Alto, CA, Mar. 1982, pp. 2–11.
[49] J. L. Hennessy and T. R. Gross, "Postpass code optimization of pipeline constraints," *ACM Trans. on Programming Lang. Syst.*, vol. 5, no. 3, July 1983.
[50] T. R. Gross, "Code optimization of pipeline constraints," Ph.D. dissertation, Stanford Univ., Stanford, CA, Aug. 1983.
[51] M. Flynn, *The Interpretive Interface: Resources and Program Representation in Computer Organization*. New York: Academic, 1977, ch. I-3, pp. 41–70; see also *Proc. Symp. High Speed Computer and Algorithm Organization*.
[52] M. Flynn and L. Hoevel, "Execution architecture: The DELtran experiment," *IEEE Trans. Comput.*, vol. C-32, no. 2, pp. 156–174, Feb. 1983.
[53] G. Meyer, "The case against stack-oriented instruction sets," *Comput. Architecture News*, vol. 6, no. 3, Aug. 1977.
[54] *MC68000 Users Manual*, 2nd ed., Motorola Inc., Austin, TX, 1980.
[55] E. Stritter and T. Gunther, "A microprocessor architecture for a changing world: The Motorola 68000," *Computer*, vol. 12, no. 2, pp. 43–52, Feb. 1979.
[56] R. Schumann and W. Parker, "A 32b bus interface chip," in *Dig. 1984 Int. Solid-State Circuits Conf.*, IEEE, San Francisco, CA, Feb. 1984, pp. 176–177.
[57] R. Sites, "Instruction ordering for the Cray-1 computer," University of California, San Diego, Tech. Rep. 78-CS-023, July 1978.
[58] J. Mavor, M. Jack, and P. Denyer, *Introduction to MOS LSI Design*. London, England: Addison-Wesley, 1983.
[59] A. Rainal, "Computing inductive noise of chip packages," *Bell Lab. Tech. J.*, vol. 63, no. 1, pp. 177–195, Jan. 1984.
[60] D. A. Patterson and C. H. Sequin, "RISC-I: A reduced instruction set VLSI computer," in *Proc. 8th Annu. Symp. Computer Architecture*, Minneapolis, MN, May 1981, pp. 443–457.
[61] C. Seitz, "Concurrent VLSI architectures," *IEEE Trans. Comput.*, this issue, pp. 1247–1265.
[62] J. Batali, E. Goodhue, C. Hanson, H. Shrobe, R. Stallman, and G. Sussman, "The SCHEME-81 architecture — system and chip," in *Proc. Conf. Advanced Research in VLSI*, Paul Penfield, Jr., Ed, Cambridge, MA: MIT Press, Jan. 1982, pp. 69–77.
[63] D. Ungar, R. Blau, P. Foley, D. Simples, and D. Patterson, "Architecture of SOAR: Smalltalk on a RISC," in *Proc. 11th Symp. Computer Architecture*, ACM/IEEE, Ann Arbor, MI, June 1984, pp. 188–197.
[64] D. Ungar, "Generation scavening: A nondisruptive high performance storage reclaimation algorithm," in *Proc. Software Eng. Symp. Practical Software Development Environments*, ACM, Pittsburgh, PA, Apr. 1984, pp. 157–167.
[65] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*. Reading, MA: Addison-Wesley, 1983.
[66] L. Deutsch, "The Dorado Smalltalk-80 implementation: Hardware architecture's impact on software architecture," in *Smalltalk-80: Bits of History, Words of Advice*, Glenn Krasner, Ed. Reading, MA: Addison-Wesley, 1983, pp. 113–126.
[67] R. Davies, "The case for CMOS," *IEEE Spectrum*, vol. 20, no. 10, pp. 26–32, Oct. 1983.
[68] R. Eden, A. Livingston, and B. Welch, "Integrated circuits: The case for gallium arsenide," *IEEE Spectrum*, vol. 20, no. 12, pp. 30–37, Dec. 1983.

John L. Hennessy received the B.E. degree in electrical engineering from Villanova University, Villanova, PA, in 1973 and is the recipient of the 1983 John J. Gallen Memorial Award. He received the Masters and Ph.D. degrees in computer science from the State University of New York, Stony Brook, in 1975 and 1977, respectively.

Since September 1977 he has been with the Computer Systems Laboratory at Stanford University where he is currently an Associate Professor of Electrical Engineering and Director of the Computer Systems Laboratory. He has done research on several issues in compiler design and optimization. Much of his current work is in VLSI. He is the designer of the SLIM system, which constructs VLSI control implementations from high level language specifications. He is also the leader of the MIPS project. MIPS is a high performance VLSI microprocessor designed to execute code for high level languages.