And Now a Case for More Complex Instruction Sets

Michael J. Flynn, Chad L. Mitchell, and Johannes M. Mulder
Stanford University

With the spate of recent papers and product announcements,[1,2] it might seem that the reduced instruction set computer, or RISC, approach to instruction set design has been universally accepted as superior. These RISC designs[3] have been characterized as having few simple instruction types with fixed instruction size and formats. The antithesis of RISC designs have been designated CISC, for complex instruction set computer, and characterized as having large instruction vocabularies with multiple sizes, formats, and addressing modes.[4]

RISC performance estimates, when compared to alternative conventional CISC (such as VAX, S/360, etc.) approaches, seem impressive. While there have been critical appraisals of the RISC approach,[5] the comparative performance evaluations provide formidable, although qualified, evidence in its favor.

Our purpose here is to evaluate and compare RISC-type designs with non-RISC instruction-set extensions using a level playing field: with similar compiler strategies, without compatibility considerations, and with similar implementation constraints. We also deal with instruction set evaluation. The data presented is based on five benchmark programs discussed later.

Using a computer architecture simulation platform, we can perform instruction set tradeoffs with a common optimizing compiler and workload.

Instruction sets have many attributes as well as constraints. The key to careful instruction set evaluation is to consider key attributes and the tradeoffs possible among them in light of implementation constraints.

Modeling performance in instruction set design

It is difficult to create a truly fair basis for comparing instruction set designs[6] because of the myriad of considerations and compromises in achieving the final design. Broadly, these factors fall into two classes: those concerned with functional requirements and those directly related to performance.

The former class includes issues such as compatibility, design time, and technology selection. We will discuss it briefly in a later section.

Performance-oriented considerations. Quantitative evaluation certainly makes for interesting comparisons, especially when driven by a common and representative workload. Typical runtime measurements include both static and dynamic characteristics of processor execution:

(1) Static measures. They simply represent the static size of program representation. In the absence of other considerations, smaller size is better, as more concise code should have better locality in a memory hierarchy and requires less memory bandwidth.

(2) Dynamic measures. They include the number and type of instructions executed, number of data references required for reads and writes, and memory traffic as measured in number of bytes transferred.

(3) Compilation time. An item frequently overlooked in comparisons is the time it takes to create a program represen-

0018-9162/87/0900-0071$01.00 ©1987 IEEE

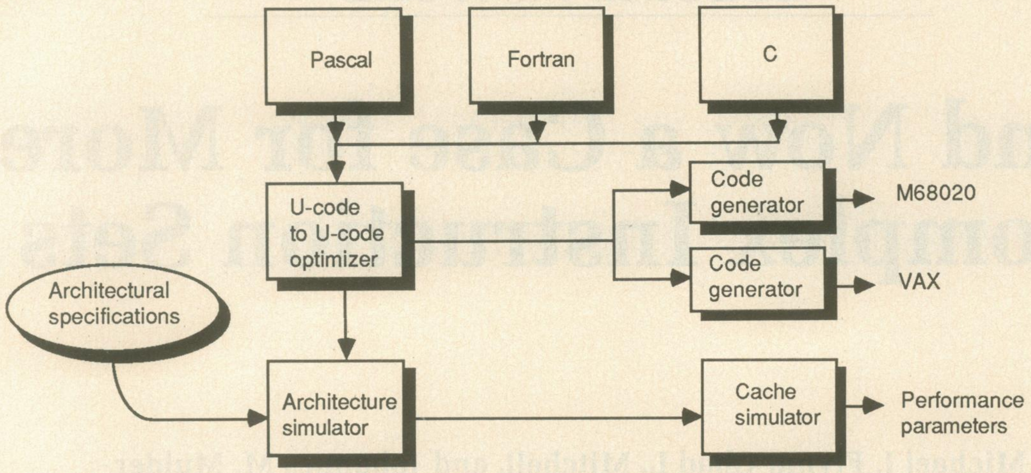


Figure 1. Computer architect's workbench.

tation. It has been variously estimated that half of mainframe problem-state activity involves compiling programs.

(4) Operating system execution time. Depending on the system and the application, a significant fraction of machine execution time is spent in the operating system. A good instruction set must support system functions.

A basis for comparison. In the creation of a new instruction set design, the evaluation of expected runtime parameters is an important facet in understanding the required tradeoffs. While it is imperative to compare a projected design against existing designs, it is just as important to understand the limitations of such comparisons. To achieve a small runtime advantage, for example over a VAX model, is necessary but not conclusive evidence of design validity. Similarly, comparing a design and technology which will be available to customers in future years with designs done several years ago and currently in production may give a misleading impression of the role of the instruction set in determining performance.

A further problem is that comparisons are made not simply across processors, but usually across processor-compiler pairs. Processor-instruction-set variations can become lost in the noise of differences in compilers. We need to create a level playing field for the evaluation of instruction set alternatives using a common compile-time strategy.

In the remainder of this article we assume

- a common workload (benchmarks),
- a simple register-oriented base instruction set,
- a fixed implementation technology,
- similar compiler optimizations for all instruction set variations, and
- the same arithmetic logic unit (ALU), data paths, and instruction vocabulary for operations for all instruction set variants (thus, the same ALU cost). Instruction count differences will arise *only* due to format and register differences.

We will use a simple base design to evaluate the usefulness of several possible additions.

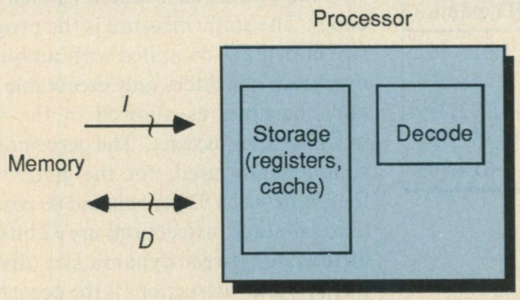
A computer architect's workbench

The computer architect's workbench[7] is a set of tools developed at Stanford which allows the evaluation of architectural and memory system parameters for a variety of different instruction sets using a common compiler front end. As shown in Figure 1, applications written in Pascal, Fortran, or C compile into an intermediate code called U-code. If desired, we can optimize the U-code representation of the application by means of the global U-code to U-code global optimizer.

The actual simulation consists of a static and a dynamic part. During the static part, the simulator extracts static information per *basic block* (a code segment with a single entry and exit point). The information includes the code size of the basic block for the target architecture and data-reference information. The architectural specification drives this stage and determines the code size. To examine the relative performance of different instruction sets on an application, an architect simply parameterizes an architecture or instruction set family.

During the dynamic part, the benchmark executes and passes dynamic information to the simulator. Information includes the currently executing basic block and dynamic data references. The simulator associates the dynamic basic-block trace with the static basic-block information; generates simple architectural characteristics such as program size, number of executed instructions, number of memory references, and so forth; and generates an address trace to drive a subsequent cache simulator.

The architect's workbench allows rapid evaluation of multiple architectural and memory system designs. Among the instruction sets currently available are



Tradeoff #1
Within limits, more complex instruction decode
(More formats, operations, etc.)
→ reduces memory traffic for instruction (without cache)
or → reduces instruction-cache size for constant memory traffic.

Tradeoff #2
Within limits, increasing register-set size (and/or complexity)
→reduces memory traffic for data (without cache)
or → reduces data-cache size for constant memory traffic.

Tradeoff #3
More complex instruction decode and/or increased register-set size may increase cycle time and decode area.

Figure 2. Basic tradeoffs.

- stack machines, including fixed-size stack machines, byte-encoded stack machines, and B6700-type stack machines;
- register-set machines, including load-store architectures and System/360-type architectures; and
- direct-correspondence architectures.[8]

Using the architect's workbench to evaluate architectures. With the workbench we normalize the effects of compiler optimization. The optimizer can be turned on or off, but all architectures receive the same degree of optimization. With a program trace we can view the effect of register allocation and create perfect allocation by reallocating after initial program execution.

The workbench was developed to allow top-down evaluation of a variety of architectures for a particular workload. The designer uses the workbench to select an instruction set customized to the particular application. The system is easy to use since in the initial evaluation only the basic instruction set parameters need to be specified. Unless explicitly added, the system will default to (assume that)

(1) All architectures have the same functional (ALU-type) operations and these operations correspond to the actions defined by the high-level source language of the benchmarks (Pascal, for our benchmarks).

(2) All architectures have the same data paths (32 bits for this study).

(3) All instructions execute in unit time. We do not evaluate the effects of pipelining, but we do calculate the processor cycles spent in the data buffer and memory.

The system is designed to allow basic high-level tradeoffs, such as in instruction format selection, instruction encoding, register-set size and organization, and cache size and organization. After making an initial evaluation to select several promising candidate architectures, the designer would supply the additional information specifying ALU vocabulary and timing, pipeline timing templates, pipeline interlocks, etc., for a complete behavioral simulation.

Currently our system does not include facilities for pipeline timing evaluation, although it is being extended to include all of the above mentioned features.

For this article we based our tradeoff measurements on memory traffic; they do not directly include pipeline-cycle counts. Where differences arise, we will estimate the effect on cycle count.

An instruction set consists of a tradeoff between memory bandwidth and processor storage, as well as processor-decode requirements (see Figure 2). There are several basic tradeoffs based on instruction encoding and available processor storage. The amount of complexity associated with instruction decode (the conciseness of the encoding) determines both the number of instructions fetched and the number of bits required from memory to interpret a program. The number of registers available for the allocator, together with the allocation strategy, determine the data traffic. Both instruction and data references to memory can be diminished by the presence of cache, either separate caches for the instruction and data stream or an integrated cache for both.

If we use care in evaluating relatively close architectural alternatives, we can get a good idea as to which architectural strategies are better than others, even though we may not be able to determine an exact optimum. In dealing with two similar designs, we can invoke a principle of marginal utility: Enhance a base design by an alternative that provides the maximum performance per unit cost. In order to use this in our analysis, we make the assumption that all processor variants under study have the same operational instruction set—they execute the same data transformational instructions (add, shift, etc.), even though they may differ in architectural instructions required by their instruction set (number of loads and stores in a register architecture, or number of pushes and pops in a stack architecture). By

Table 1. Benchmark sizes for stack architecture.

Benchmark	Static Size (bytes)	% Actually Referenced	Dynamic Size (bytes)	Instructions (Dynamic)
CCAL	12,980	63%	4,391,864	1,058,262
Compare	8,948	60%	35,113,324	8,538,373
PCOMP	71,276	69%	22,084,724	5,323,939
PASM	15,424	80%	17,814,260	4,352,798
Macro	73,980	53%	2,538,512	617,765

Table 2. Four instruction sets studied.

Type	Formats	Encoding	Addressability
Fix32	Two basic types: (1) Load R_1, with Mem [addr] or (2) $R_1 \leftarrow R_2$ op R3 (all operands in registers)	All instructions occupy 32^b	Word (32^b)
OBI360	As with Fix32 plus RX type,* $R_1 \leftarrow R_1$ op Mem [Addr]	Memory refr instr use 32^b; register refr instr use 16^b and 32^b	Half word (16^b)
Stack	Stack formats	Instructions with opcode only 8^b; memory refr instr use 32^b	Byte
B6700	Stack formats with special encoding of constants and pointer-referenced memory	Instr with opcode only 8^b; memory refr instr use 16^b	Byte

*Register set machine operations take two independent source operands (R1 and R2) and place the result in either an independent register (R3) or one of the source operands (say, R1). Most RISC machines as well as our Fix32 use the former convention, while the IBM System/360 uses the latter. For our code generator there is little advantage for the independent R3 specification, i.e., the OBI360 data is approximately the same (within 1%) for either convention.

assuming the same operational (ALU) vocabulary, the same data path size, and the same arithmetic performance, we create a standard cost for that part of the processor. We are left with the comparison of only those parts of the architecture directly affected by the instruction set tradeoffs. These include the decoder, the register set size, instruction cache size, and data cache size. We assume that small tradeoffs in register set size and decoder size will not materially influence the cycle time itself. We will comment on this assumption later before drawing general conclusions.

The result of enhancing a base design with various alternatives provides insight into a *local* optimum; it does not find a *global* optimum. While we are able to comment on RISC and register-set-based instruction variations, we will not comment here on significantly different instruction sets (such as complex stack machines).

The benchmarks. We selected the benchmarks used in this study as representative of workstation applications. They consist of five Pascal programs originally used by Alpert.[9] Some static and dynamic measures for the benchmarks are given in Table 1. All data is for a stack machine with fixed 32-bit instructions (similar to P-code). The static measure is the program size in bytes as compiled without linkage overhead; it includes only executable code and constants as allowed in the stack machine architecture.[7] The percentage of code actually used, for the given input files, is between 53 percent and 80 percent. Since all stack instructions are 32 bits, the difference between dynamic size (divided by four) and instructions is the occurrence of pointers, especially those associated with procedure calls.

The CCAL benchmark emulates a desk calculator. It reads a script of calculations from a text file and produces results in another text file. As with the other benchmarks, any input files required are specified as part of the benchmark, thus defining a standard execution. The Compare benchmark compares two text files, producing a description of their differences (similar to the Unix Diff command). The PCOMP benchmark compiles a Pascal program by recursive descent and produces P-code output. The PASM benchmark assembles the P-code output from the P-code compiler. The Macro benchmark is a macro processor for the SCALD computer-aided design system.

The chosen benchmarks are representative Pascal programs of medium size. They represent program generation, file processing, and calculation. CCAL also represents an interactive (as opposed to batch) program, although it is driven from a script file to keep its execution standard.

Each of these benchmarks was executed once for each target architecture after analysis of its basic blocks. Subsequently the address trace of that execution was fed into the cache simulator. The results presented are for the mean of the (equally weighted) benchmarks.

RISC-CISC code analysis

In the following analysis we define the RISC-type architecture[3] as follows:

(1) Load-store architecture. It does not allow memory operands for ALU operations.

(2) Register-file oriented.

(3) Pipelined execution with short cycle time, delayed branch, and a few register-oriented instructions.

(4) Fixed 32-bit instruction size.

The base RISC we simulate is called Fix32. It is a load-store architecture with a 32-bit fixed instruction size with a register-set size of 16. (We examine other register set sizes in later sections.) Because delayed branching can be applied to our base architecture and its extensions, the effect of delayed branching is not simulated. This does not influence the results, however, because delayed branching affects the extensions of our base architecture exactly the same way as it affects the base architecture itself.

While we present some data on a number of architectural possibilities (see Table 2), one is particularly interesting: OBI360 (Only-Binary IBM 360). This variation is generally similar to IBM System 360 with the storage-to-storage instruction format excluded. OBI360 makes two modifications to the RISC strategy:

1. It adds the "RX" format (32 bits), allowing one instruction operand to reside in memory:

$R_1 := R_1$ op Mem$[R_2 + \text{offset}]$

2. It adds half-size instructions (16-bit) register-to-register instructions:

$R_1 := R_1$ op R_2

Starting off with the minimum Fix32 architecture, we perturb this design in various ways, such as

(1) by increasing the complexity of the instruction encoding (using OBI360 instruction formats), or

(2) by increasing the number of registers available to the data stream.

We make the assumption that other things remain constant—the base instruction set operational vocabulary, cycle time (discussed later), etc. We first consider the effect of instruction set selection on memory traffic in the presence of various-sized caches, then we consider the issue of register set size, organization, and allocation policy on memory traffic, again in the presence of various-sized caches.

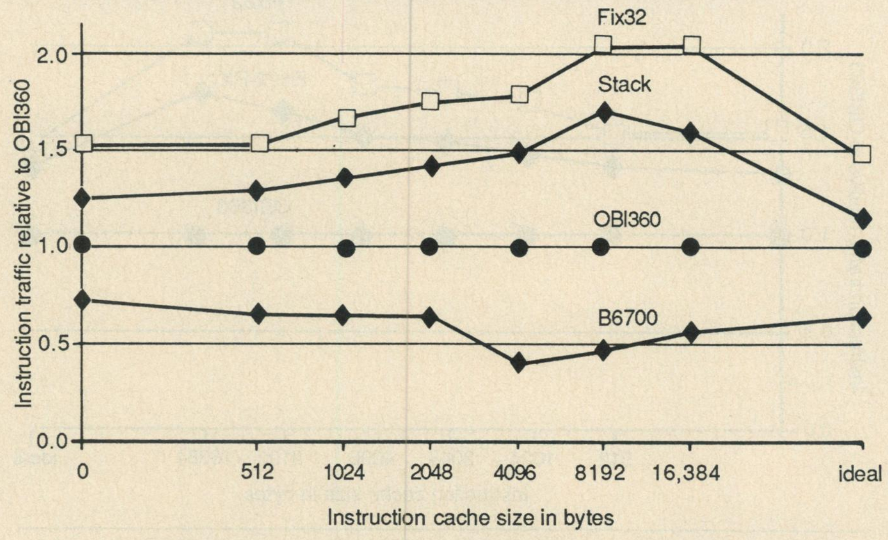


Figure 3. All architecture families (two-way associative 16-byte lines).

The instruction set

The instruction set itself is largely a compromise between the complexity of the decoder (and thus the ensuing size of the microcode, cycle-time, etc.) and the required memory traffic to support execution (and thus the number of memory references). The RISC approach has opted for minimum decode complexity and accepted a relatively high bandwidth requirement for the instruction execution.

We can see the effects of instruction set selection on instruction traffic in Figure 3, which shows several instruction set families (see Table 2) relative to the memory traffic generated by OBI360 (with 16 general-purpose registers). The memory traffic is plotted for various cache sizes from no cache (zero bytes) to an infinite cache (ideal). A common cache policy of two-way associativity with a line size of 8 bytes has been selected across all caches and instruction sets. The relatively small line size tends to reduce the absolute amount of memory traffic required to support program execution and to diminish the difference among the instruction set families. Keeping the line size constant normalizes the cost for instruction caches for all of the families.

Figure 3 is interesting in a number of ways. Without a cache, the difference among architectures concerning the number of instruction bytes required to execute a program is about two to one (from the least dense architecture, the Fix32 with 16 registers, to the most dense architecture, the B6700).

Because an instruction cache benefits all architectures, it is difficult to see the relative benefit unless we normalize the traffic, as we have done in Figure 3. In the absence of a cache, the Fix32 architecture has 50 percent more instruction traffic than OBI360; this figure rises to 100 percent for certain intermediate cache sizes (8K and 16K bytes). For these intermediate sizes, the OBI360 architecture has captured its working set, while the Fix32 has not yet done so. Ultimately, as all caches capture their working sets, the original relationship is restored, representing simply the number of references required to initially bring a program into the cache. Notice that for certain intermediate size caches, we can see relative differences of greater than five to one between the most dense and least dense architectures (B6700 to Fix32 in the 4K- to 8K-byte range).

Figure 4 shows the effect of marginal increases in the instruction decoder on overall dynamic program size and the resultant effect on cache performance. In Fix32-RX the RX format is added to Fix32. The RX format simply combines a load with a register-register operation. Since the RX format is the conjunction of two existing instruction types, its implementation cost can be minimal, depending on pipeline organization. About 10 percent fewer instructions will be executed with 10 percent fewer instruction-decode cycles. One might expect the pipelined program execution to improve by the same amount. It does not, because the RX instruction requires an extra cycle of interpretation, which may or may not be overlapped. Thus, in addition to a 10 percent reduction in instruction traffic, the

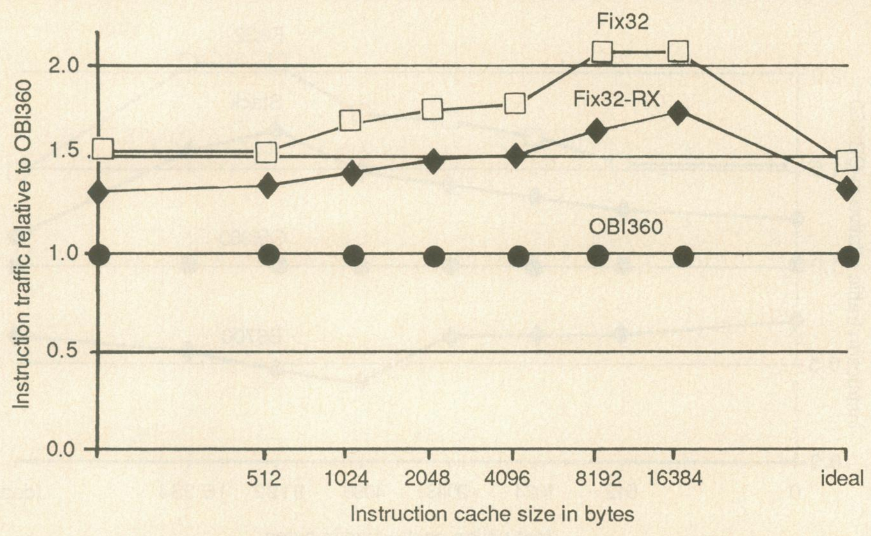


Figure 4. Memory referencing characteristics of register set architectures.

The effects of memory-to-register (RX) instructions

If no change is made to the pipeline, there is no reduction in the number of cycles executed, only reduction in instruction bandwidth. Basically, the sequence

Load R_1,R_2, Mem Disp; $R_1 \leftarrow$ Mem [[R_2] + Mem Disp]
Add R_3,R_1 ; $R_3 \leftarrow R_3 + R_1$

is replaced by

Add R_3, R_2, Mem Disp; $R_3 \leftarrow R_3$ + Mem [[R_2] + Disp]

In both cases above, the sum is placed in R_3 in the same cycle. With the RX format, the pipeline is available one cycle early to admit a new instruction. However, if the pipeline uses the ALU for both address generation and execution, an RX instruction sequence will be limited to the same performance as before due to ALU contention.

If the processor is extended to include a separate adder for address generation and appropriate pipeline control, much (but not all) of this contention is avoided and performance enhancement can be realized.

As a rough estimate, suppose conditional branches represented 20 percent of instructions executed (and Fix32 load instructions, 30 percent). Now Fix32-RX reduces the frequency of load instructions to 20 percent (combining the load with an operation). However, the RX operation completes execution at the same time as the load-operate instruction pair. Thus, when the RX instruction immediately precedes a branch instruction that tests its result, no time is saved; otherwise a cycle is saved. Thus, if a branch is occupied every fifth position, a cycle is saved when the RX is located in any but the fourth position, or immediately preceding the branch. In the other three cases, the cycle is saved, giving a performance improvement potential (without register/ALU usage conflicts) of 7.5 percent rather than 10 percent.

number of execution cycles also reduces, but less than 10 percent (see the sidebar, "The effects of memory-to-register (RX) instructions").

Adding the RX format and a half-sized (16-bit) RR format to Fix32 results in OBI360 with one-third improvement in instruction bandwidth requirements for the no-cache case. The addition of half-size instructions to the format, however, does require additional decoder complexity to realize the alignment of instructions for proper decoding (see the sidebar, "The cost of half-size instructions"). Whether the two modifications to Fix32 to obtain OBI360 extend the cycle time we will discuss later.

OBI360 provides a striking improvement over Fix32 in the effectiveness of an instruction cache. Table 3 shows that OBI360 realizes the same miss rate as Fix32 with an instruction cache of exactly half the size of Fix32. Thus, by moving to OBI360 and adding decoder hardware, the resultant design would require fewer instruction cycles and would realize the same memory traffic with half of the Fix32 cache.

The data stream

What is the value of large register sets? What is the value of register windows? How effective is a data cache in the presence of a register set? These are some of the questions that the architect must address in creating a balanced instruction set design.

Registers have many roles. They hold: temporary values in expression evaluation, variables from statement to statement within a procedure, constants and pointers. Figure 5 shows types of references to data memory for variable and temporary accesses for source (Pascal and C) programs. If one had an architecture without registers (an all-memory architecture), 47 percent of the data references would be for temporary storage of intermediate results within the evaluation of expressions. The addition of two or three registers, whether through use of a stack or a register-instruction format, basically eliminates these references. When we ignore expression evaluation, the resultant traffic is for *extended source variables*—variables whose values are to be carried statement to statement within a procedure because of high probability of use.

The register allocator is responsible for predicting the optimum assignment of variables to registers. Let us define this data traffic as the unity data traffic (thus excluding expression evaluation). Unity data traffic is not exactly the same as the variable traffic seen in a source program, however. The difference includes accesses for dynamic links, static links (the runtime organization), the fact that source variables may have arbitrary lengths but the physical system has a fixed-length bus (assumed to be 4 bytes), and finally that the compiler must create variables to control, for example, With and For statements.

Given different instruction sets with the same register set size and same compiler optimizer, the resultant data traffic will be the same for all instruction sets. How does data traffic compare to instruction traffic? Without data or instruction cache, the instruction traffic dominates the data traffic, but this quickly decreases when even a small instruction cache is added.

Figure 6 shows the instruction traffic for Fix32 and OBI360 instruction sets relative to unity data traffic. Both Fix32 and OBI360 include 16 general-purpose registers. For OBI360, the figure shows 10 percent more traffic for instructions than for data, while for Fix32, this difference is 60 percent. The addition of an instruction cache reduces the instruction traffic well below the unity data traffic for all architectures; thus, we need to rebalance the data traffic when considering a small instruction cache for the processor. The following four sections present different ways to rebalance the total memory traffic between instructions and data.

Single register set. Figure 7 shows the allocation of registers within a typical register set. Unity data traffic requires about eight registers in the set; they are allocated for constants, the evaluation stack, and state variables. The marginal value of adding eight registers to the initial eight depends upon register allocation. Figure 8 presents two allocation strategies: a very simple strategy wherein registers are allocated only among variables within a basic block, and a more elaborate strategy which allocates variables within a procedure. Global register allocation is by means of priority-based coloring. Notice that with or without optimization, the marginal value of more than eight additional registers is moot, and that a reduction from unity data traffic down to 0.65 can be achieved with straightforward global register allocation.

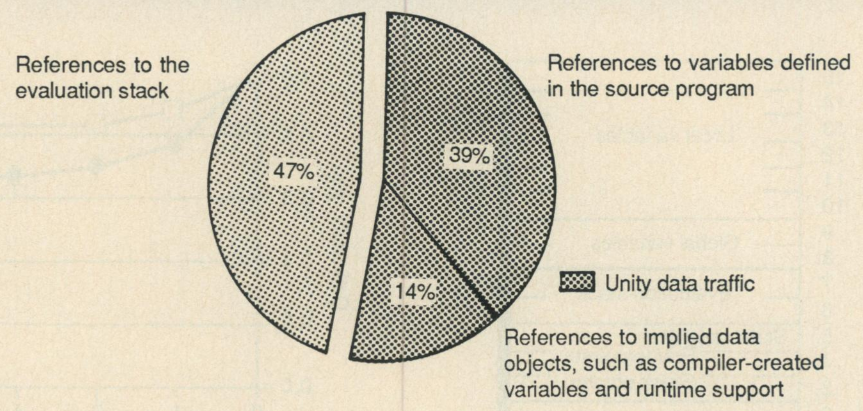


Figure 5. Distribution of data reference types.

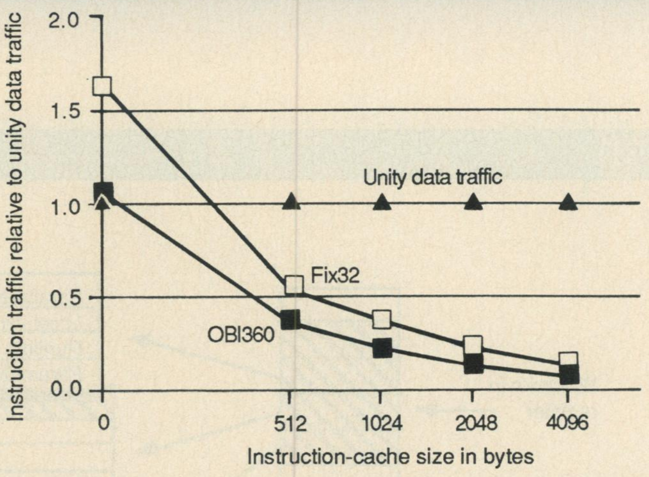


Figure 6. Instruction and data traffic as a function of architecture and instruction cache size.

The cost of half-size instructions

If a base processor design were intended to include 16^b instructions the following hardware might be added:

(1) A two-word instruction buffer (IB) 2×32^b
(2) Multiplexors from the IB into the instruction decoder
(3) Finite-state machine for IB control
(4) Modified program counter to allow both 16^b increment (for instruction control) and 32^b increment (for IB control)

Table 3. Relative traffic or miss rates. All traffic is relative to OBI360 without a cache and with 32^b-wide paths to memory.

For cache* size:	0	512	1^K	2^K	4^K
Fix32	1.52	.44	.30	.19	.11
Fix32-RX	1.33	.37	.24	.17	.09
OBI360	1.0	.28	.17	.11	.05

*Two-way set-associative, 8-byte lines

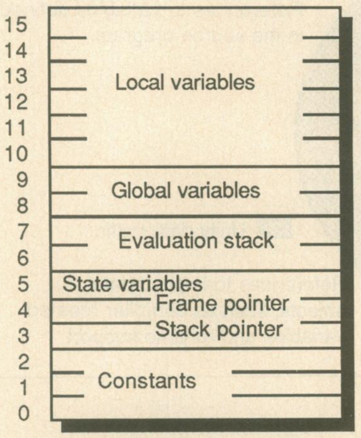


Figure 7. A possible register set usage outline.

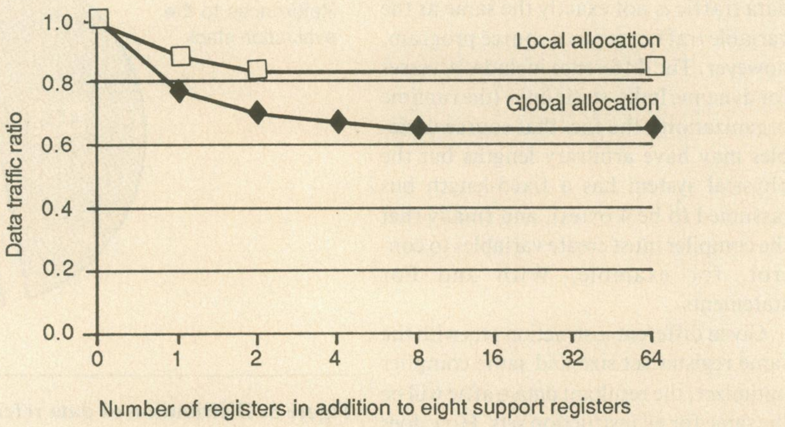


Figure 8. Single-register-set performance relative to unity data traffic.

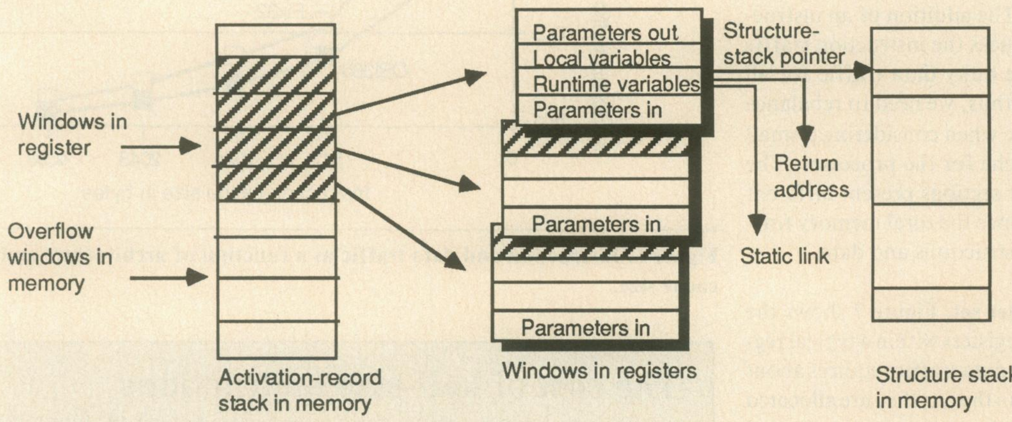


Figure 9. Multiple overlapping register windows.

To improve traffic beyond this requires allocation of registers across procedures. This can be done either in software, by interprocedural register allocation, or in hardware, through register windows.

Interprocedural register allocation. An interprocedural register allocator reduces the penalty of saving and restoring registers around procedure calls by allocating some registers as private to a particular procedure. Because procedures which mutually exclude each other from being in the call chain at the same time can share private registers, the number of registers required is reasonably small. Interprocedural register allocation has the additional advantage that runtime management variables, such as dynamic links, static links, and procedure-return addresses, can be allocated to registers using a method similar to that used for procedure variables.

A simple and efficient interprocedural allocation scheme[10] assigns private registers depth first in the call tree. Leaf procedures come first, callers of leaf procedures second, and so forth. This scheme treats recursion paths as single nodes and does not allocate private registers to procedures in such a path. Figure 10 shows the performance of an allocator based on this scheme, but extended with an extra pass to detect variables potentially accessed through pointers[11]; these variables cannot be allocated to private registers. The figure shows a traffic ratio of 0.5 with 8 registers and 0.4 with 32 registers available to the allocator.

Interprocedural allocation also has its drawbacks:

- procedures called recursively cannot have private registers,
- separately compiled modules must have knowledge of the register usage of all intra-module calls, or require register allocation during link time, and
- incremental compilation without an explicit linking stage such as Lisp environments can profit only partially from this scheme.

Interprocedural register allocation may turn single register sets into the most efficient local memory. Because of the drawbacks, however, unless explicitly stated otherwise we use only global (procedural) allocation in comparing different data buffers in the following sections.

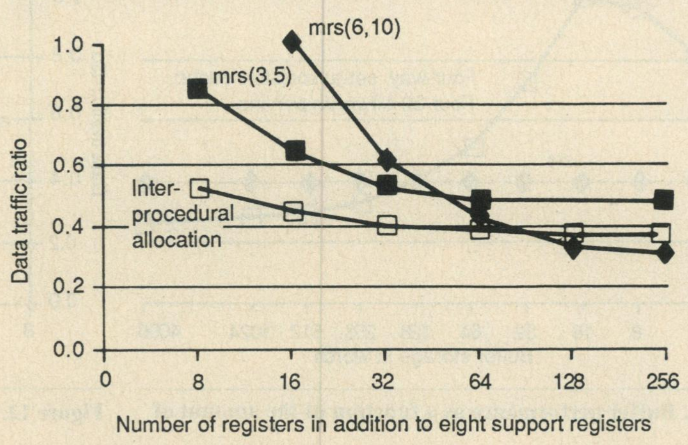


Figure 10. Performance of multiple overlapping register windows as a function of the total number of registers (relative to unity data traffic).

Multiple overlapping register windows. Register windows allow a new set of registers to be made available for each procedure, with an overlap of registers between the caller and the called procedure to allow for passing of parameters. When a procedure call exhausts the number of windows, a window is freed by saving its data in memory. Whenever the data is needed again, it is restored from memory. The conditions which require a window save and restore are called overflow and underflow, respectively.

The hardware windows are organized as a circular buffer always covering the top part of the runtime stack. The outgoing parameters of the top window are the same as the incoming parameters of the bottom window. The hardware windows can be envisioned as rolling back (returns) and forth (calls) over the window stack in memory. Every activation record in the runtime stack has a fixed size, the window size. A second runtime stack, called the *structure stack*, keeps additional procedure variables which do not fit in the window stack. Every record in the window stack maintains a pointer to that part of the second stack which holds these variables.

Figure 9 shows the facets of a multiple-overlapping-window organization: the hardware windows covering the top of the window stack; the overlapping windows, holding parameters, local and runtime management variables; and the structure stack. An additional facility, which improves the performance of overlapping windows, allows pointer access to windows other than the one on top of the stack. This allows variables accessed through *var* parameters, the static-link, or pointers to be allocated in a window. For this article, we assume that overlapping windows include this facility.

We designate a window organization with N shared registers and M local registers by mrs(N, M). The effectiveness of register windows depends mainly on two parameters: the size of the window and the number of windows. Figure 10 shows that a large window size, mrs(6, 10), has a detrimental effect on performance for organizations with few windows. Few windows imply frequent under- and overflow conditions, and therefore a high penalty for large sets. However, a small window, mrs(3, 5), has a negative effect on performance for an organization that has many windows. A small window captures fewer local variables and parameters, and therefore is less efficient compared with a large window when the over- and underflow traffic stops dominating total traffic.

To achieve the best of both large and small windows, global register allocation can be combined with small windows. This combination on the average outperforms both mrs(3, 5) and mrs(6, 10) for our benchmarks. In this article, however, we are concerned with the utility of large register sets and especially those organizations actually implemented in general-purpose microprocessors. The buffer comparison in the following section, therefore, does not take small-window buffers into account. The performance of small-window buffers with and without register allocation and additional caching is presented elsewhere.[11]

Single register set and cache combination. An argument in favor of larger register sets is that they will reduce the number of accesses to memory. Of course, a cache will do the same thing, and an interesting tradeoff occurs between increasing register set size and introducing or enlarging a data cache. Viewed from the memory system, enlarging either the register-set size or the cache size reduces the required memory traffic.

Using the traffic ratio as a function of provided storage for buffer comparison is not fair for two reasons: first, the storage provided does not accurately reflect the usage of chip area; and second, the traffic ratio does not completely determine processor performance.

To present buffer performance as a function of occupied area instead of the number of bits of storage, we define a simple storage-to-area mapping. The key points in this mapping are the inclusion of tag and status bits for caches, the distinction between the size of a cache RAM cell and a multiport register RAM cell, and the

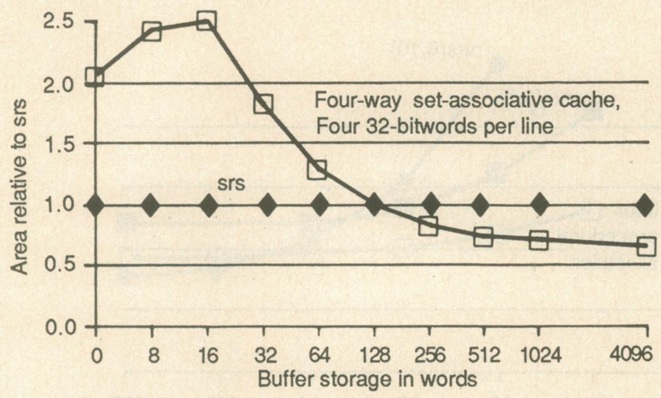


Figure 11. Buffer performance as a function of the amount of chip area.

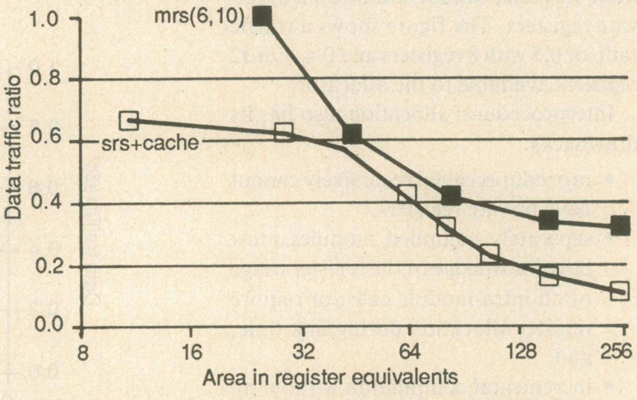


Figure 12. Data traffic ratio as a function of chip area.

inclusion of area for drivers, sense amplifiers, and tag comparators. The intent of this model is to penalize small register sets and caches for their inherent area overhead and to penalize register sets for their relatively large RAM cells, which they need to supply the high bandwidth required for pipelined processors. The main assumption underlying this model is that the RAM-cell size of a particular buffer is independent of the size of the buffer. Figure 11 shows the area of the cache* relative to the area of a register set as a function of the provided storage. The initial area disparity between the two buffers is mainly due to the tag comparators, and the state machine needed to control the cache. When the cache becomes larger than 32 lines, or 128 words, it actually takes less area than the register set, because the cache RAM cell is significantly smaller than the register RAM cell. Mulder gives a complete description and a validity assessment of the area model elsewhere.[11] The model parameters are all based on actual CMOS register set and cache designs.[12,13]

The traffic ratio depicts the effectiveness of the data buffer viewed from the memory, but the situation is not quite the same when viewed from the processor. Access to a cache is usually a one- or two-cycle operation, whereas access to a register set can be included within a cycle. Very large register sets may add a cycle or extend the cycle (see next section), but caches invariably add additional cycles to total program execution. A more accurate measure than the traffic ratio would be the ratio of the processor cycles spent in the buffer and main-memory combination and the cycles spent in the memory system without the presence of a buffer. Mulder describes this measure, the *cycle ratio*, in detail.[11]

In a highly pipelined organization which keeps its buffer and memory system busy all the time, the cycle ratio describes the exact performance of the processor. Note that pipeline breaks, not caused by the memory and buffer system, may cause the real performance to deviate from the cycle ratio. Nonetheless, it is a good measure for evaluating different buffer organizations and their timing parameters.

Figure 12 summarizes the traffic ratio, and Figures 13 and 14 the cycle ratio of two buffering strategies. All three figures show their ratios as a function of occupied area in register equivalents; one register equivalent is the area occupied by 32 register bit cells. Figures 13 and 14 show the cycle ratio for a main memory access time of two and three cycles, respectively.* The figures show the ratios for a multiple-register-window organization, mrs(6,10), and a single-register-set and cache combination, srs + cache. The register allocator only uses four registers for allocation, but the architecture is assumed to have an additional eight as described before.

Both Figures 12 and 14 show a slight advantage for srs + cache over mrs(6,10) between 40 and 80 register equivalents. Before and after this interval, srs + cache performs significantly better than mrs(6,10). Reducing the memory access time from three to two cycles, however, undoes the srs + cache advantage. Now mrs(6,10) is slightly better than srs + cache for the 40- to 80-register interval, and performs approximately the same for larger buffers. Nonetheless, an increase in average memory access time is always an advantage of the srs + cache organization. The case for multiple register sets with relatively large sets is slight, at best, and occurs only in the region of 32 to 128 registers for one- or two-cycle main memory access time. An important advantage for the srs + cache organization is its relative independence from the reference distribution. A smaller percentage of references to the window stack immediately degrades mrs(6,10) performance, while this is not necessarily the case for srs + cache.

Several buffer characteristics are not taken into account in this comparison because they lie outside the scope of the article or because insufficient data was available. These include the effect of system functions (interrupts, I/O, and context switches) and data-consistency

*The cache is four-way set associative with four one-word transfer units per line. The first data point, however, is a direct mapped one-line cache, and the second data point is a two-way set associative two-line cache.

*A cache hit takes one cycle, while a cache miss takes one cycle plus a memory access.

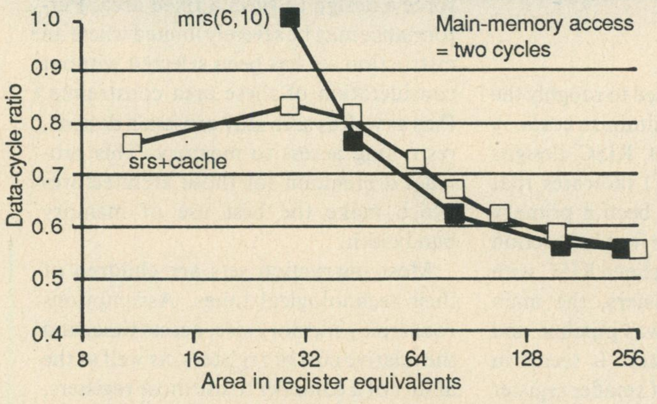


Figure 13. Cycle ratio as a function of chip area (two-cycle memory).

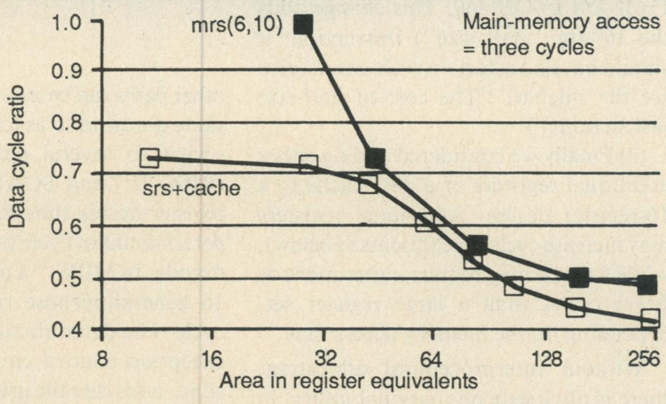


Figure 14. Cycle ratio as a function of chip area (three-cycle memory).

requirements in the case of multiprocessing.

Trading instruction traffic for data traffic

Memory traffic is the sum of the instruction traffic plus the data traffic. Assume that a processor with a Fix32 instruction set has eight general-purpose registers. Is it better to add registers, or to increase instruction complexity? Table 4 gives insight into these tradeoffs. Note that the table presents total traffic relative to the total instruction and data traffic of Fix32 with eight registers. If we increase the Fix32 register set size from eight registers to 16, we reduce the relative traffic from 1.00 to 0.76. However, if we increase the instruction complexity and retain a register set size of eight by moving from Fix32 to OBI360, the relative traffic also reduces from 1.00 to 0.76. Clearly it is desirable to do both, which would reduce the relative traffic to 0.56.

Table 4. Instruction and data traffic measured relative to the total traffic of our initial architecture, the Fix32 with eight registers.

Architecture	Register Set Size	I-traffic	D-traffic	Total
Fix32	8	0.68	0.32	1.00
Fix32	16	0.55	0.21	0.76
Fix32	32	0.55	0.21	0.76
Fix32-MRS	128	0.52	0.12	0.64
OBI360	8	0.44	0.32	0.76
OBI360	16	0.35	0.21	0.56
OBI360	32	0.35	0.21	0.56
OBI360-MRS	128	0.32	0.12	0.44

If we assume a Fix32 base design with a register set size of 16, there is almost no saving in increasing the size to 32 (unless coupled with interprocedural register allocation). The instruction traffic and the data traffic remain essentially constant. By adding eight register windows, mrs(6,10), for a total of 128 registers, we reduce data traffic to 0.12 and instruction traffic to 0.52—a net savings of 0.12 references from Fix32 with 16 general-purpose registers. On the other hand, if we simply retain the 16 general-purpose registers and modify the instruction set from Fix32 to OBI360, we realize a savings of 0.20 references. Thus, the addition of 112 registers and window control results in only 60 percent of the traffic reduction we can achieve with better instruction encoding (the change from Fix32 to OBI360).

Cycles and cycle time

So far we have dealt with design alternatives assuming that the processor execution (independent of memory traffic) was unaffected. In this section we review the alternatives examined and assess the possible impact on either the number of cycles required to execute a program, or the cycle time itself.

The cycle count. Concerning the cycle count, consider the principal design alternatives presented.

(1) We first considered Fix32 compared with Fix32 plus a memory-to-register instruction format (Fix32-RX). This change will either leave cycle count unaffected or it will allow a modest (less than 10 percent) decrease in the number of cycles (see the sidebar, "The effects of memory-to-register (RX) instructions")

when hardware is added to the pipeline.

(2) We next considered the change from Fix32-RX to OBI360. This change adds the 16^b (or "half size") instruction; it should have no effect on cycle count (also see the sidebar, "The cost of half-size instructions").

(3) Finally we considered adding either additional registers or a data cache to a 16-register design. Additional registers may increase cycle time (discussed below), while a cache may require either more or fewer cycles than a large register set, depending on the memory access time.

Without interprocedural allocation, there is obviously no marginal utility, in terms of traffic ratio or cycle count, in enlarging a single register set beyond 16 registers. With interprocedural register allocation, we see an improvement in memory traffic by increasing the total register set to 32, after which again there is little or no marginal utility. The cycle count will decrease as memory traffic decreases.

Part of the effect on cycle count of moving from a base 16-register design to multiple register set versus the move to register set and cache combination was discussed in "The data stream" and shown in Figures 13 and 14. The design with a cache clearly expends fewer cycles accessing objects not in the register set. On the other hand, when we call a new procedure, the multiple register set has an advantage because it requires fewer cycles to save and restore register values. However, many parameters are involved (frequency of calls, number of parameters, cache miss penalty, etc.), and a specific situation may find one approach significantly superior.

The cycle time. Concerning cycle time, we have considered two design extensions that could adversely affect the internal processor cycle time:

(1) The extension of Fix32 to OBI360 increases the instruction decoder complexity.

(2) Increasing the number of registers may increase register access time.

Cycle time is typically determined by one of four paths:

(1) Instruction decode
(2) Register access
(3) Cache access
(4) ALU operation and condition-code set

The designer usually is confronted with physical limitations which determine the longest path from one of the above considerations. Once that is determined, the other paths can be increased to roughly the same duration so as to minimize costs. A review of several recent RISC designs (RISC II[14] and SOAR[15]) indicates that register access time has been a primary determinant of cycle time, not instruction decode. In MIPS,[16] a pipelined RISC with 16 general-purpose registers, the main cycle time determinant was pipeline and exception control circuitry. It seems in some cases that the use of smaller register sets could improve the cycle time, while the use of OBI360 should have a negligible effect on cycle time.

Other considerations in instruction set design

So far in this article, we have examined such aspects of performance as the basic criteria in instruction set selection and design. Many important functional considerations are not performance related, or are related to performance only in a secondary way. In this section we briefly remind the reader of several of these considerations.

Compatibility. The primary level of transportability of programs is the instruction set, not the higher-level language. The reason that the VAX series from Digital Equipment Corporation and the 68000 series from Motorola Corporation resemble their antecedents is not simply an issue of designer's preference, but of customer preference. The VAX instruction set is a good case in point. It is derived from (built upon) the PDP-11, a 16-bit architecture noted for its flexible use of addressing modes. Rather than abandon the addressing modes, the VAX designers enhanced them, preserving subset compatibility with the PDP-11 yet achieving generous functionality in the 32-bit arena. The resultant design provides for relatively concise encoding of programs. Unfortunately, the flexible object identification introduces extra cycles into instruction interpretation and makes instruction pipelining more difficult. Still, the VAX series has become an industry standard because of compatibility, connectivity (I/O capabilities), and availability of software, not simply performance.

Technology. In the context of microprocessors, chip area constraints force a design to fit in a fixed area. Performance may be severely limited where an instruction set has been selected without consideration of these area constraints. Pins as well as area may constrain designs, restricting access to memory. This provides a premium for those architectures which make the best use of memory bandwidth.

Most instruction sets are children of their technological times. Assumptions concerning memory size, access time, and the relative cost of registers, as well as the ability of a compiler to use those registers, determine the tradeoffs that go into a resultant instruction set specification.

Life cycle costs. Compatibility, technology, performance, software and hardware development effort, maintenance costs, and hardware reliability are ingredients in determining the total system cost (and profitability) for the product over its lifetime. Life cycle cost is clearly an ultimate measure, and our performance discussion is simply one factor of it.

While instruction set design involves many considerations, performance data is an important component. Indeed, performance may well be a primary consideration in certain custom application-specific designs.

In evaluating design tradeoffs, the architecture-simulation platform described above provides valuable relative performance data and greatly assists in optimizing design choices from the top down. The workbench provides an early assessment of the relative merits of different design options and facilitates the selection of the option with the greatest marginal utility.

In this article we limited the data presented to five benchmarks selected from a Pascal-type environment similar to that reported on by other researchers in the area. Different environments might produce significantly different results.

The principal design alternatives we have examined are Fix32 and OBI360:

(1) Fix32 is RISC-like in formats and encoding, but *does not reduce* the ALU instruction vocabulary. The functional instruction set is the same for all architectures studied, so as to keep the ALU cost relatively constant. RISC implementation with fewer instruction types may require additional instruction memory traffic and exaggerate the difference between

architectures (see Figure 3).

(2) OBI360 is Fix32 with the addition of both the RX and half-size instruction formats. OBI360 is not System 360 or System 370. Typical S/370 code includes the operating system interface, calling conventions, and runtime prolog/epilog, which significantly distorts the locality and cache results presented here. However, to evaluate System 370 code is to evaluate an evolution of software, not simply instruction set technology. The relatively good results of OBI360, however, are a compliment to the basic tradeoffs made by System 360 instruction set designers over twenty years ago.

Concerning the various alternatives:

(1) Fix32 (a simple load-store architecture) with 16 or fewer registers and *without cache* represents a reasonable design point for a minimum cost processor. Note that the minimum processor cost was achieved with a 50 percent to 100 percent increase in instruction traffic (Figure 3), compared with more complex designs.

(2) If Fix32 is extended to reduce memory traffic, instruction encoding and formats should receive priority attention over the expansion of an instruction cache. By adding modest decode hardware to Fix32, we create OBI360, which achieves the same memory performance as Fix32, but uses an instruction cache of only half the Fix32 cache size (Table 3).

(3) From data traffic considerations alone, it seems that OBI360 with a register set of about size 16 plus a small data cache is preferable to multiple register sets for most area combinations (Figures 12 and 13).

More generally, instruction set designers cannot afford to ignore issues of code density in favor of instruction simplicity or decoding ease. Instruction bandwidth can be a significant component of memory traffic and, ultimately, processor performance. Using larger register sets to reduce data traffic from memory makes sense only when efficient instruction encoding is used to make a corresponding reduction in instruction traffic. *Balanced optimization* is the key to overall instruction set efficiency. □

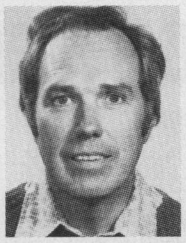
Acknowledgments

Several basic ideas used in the computer architecture simulator were based on the work of our colleague Don Alpert. We also used his benchmark programs. The U-code system included in our platform was the work of John Hennessy and Gio Wiederhold and their students here at Stanford.

This research was supported by NASA under contracts NAG2 248 and NAGW 419, and by an IBM graduate fellowship.

References

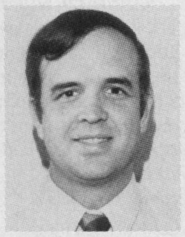
1. Also see product announcements for Fairchild Clipper, Hewlett-Packard Spectrum, Advanced Micro Devices AM29000, and MIPS Corp.
2. Mark Hill et al., "Design Decisions in SPUR," *Computer*, Nov. 1986, pp. 8-22.
3. John Hennessy, "VLSI Processor Architecture," *IEEE Trans. Computers*, Dec. 1984, pp. 1221-1246.
4. David A. Patterson and Carlo H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," *Proc. 8th Ann. Symp. Computer Architecture*, May 1981, pp. 443-458.
5. R.P. Colwell et al., "Computers, Complexity, and Controversy," *Computer*, Sept. 1985, pp. 8-19.
6. Michael J. Flynn, "Towards Better Instruction Sets," *Proc. 16th Ann. Microprogramming Workshop*, Oct. 1983, IEEE Computer Society Press, pp. 3-8.
7. Chad L. Mitchell, *Processor Architecture and Cache Performance*, Tech. Report CSL-TR-86-296, Computer Systems Laboratory, June 1986.
8. Scott Wakefield, *Studies in Execution Architectures*, PhD thesis, Stanford University, Jan. 1983.
9. Donald Alpert, *Memory Hierarchies for Directly Executed Language Microprocessors*, PhD thesis, Stanford University, June 1984.
10. Peter A. Steenkiste, *LISP on a Reduced-Instruction-Set Processor: Characterization and Optimization*, PhD thesis, Stanford University, Mar. 1987.
11. Johannes M. Mulder, *Tradeoffs in Data-Buffer and Processor-Architecture Design*, PhD thesis, Stanford University, 1987. In preparation.
12. M. Horowitz and P. Chow, "The MIPS-X Microprocessor," *Proc. Wescon 1985*, Stanford University, 1985.
13. Anant Agarwal et al., "On-chip Instruction Caches for High Performance Processors," *Proc. Advanced Research in VLSI*, Stanford University, Mar. 1987.
14. M.G.H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, PhD thesis, UC Berkeley, Oct. 1983.
15. David Ungar et al., "Architecture of SOAR: Smalltalk on a RISC," *11th Ann. Symp. Computer Architecture*, Ann Arbor, Mich., June 1984, pp. 188-197.
16. Steven A. Przybylski et al., *Organization and VLSI Implementation of MIPS*, Tech. Report 84-259, Computer Systems Laboratory, Apr. 1984.



Michael J. Flynn is a professor of electrical engineering at Stanford University, where he served as director of the Computer Systems Laboratory from 1977 to 1983. He is also a senior consultant at Palyn Associates, a computer design firm in San Jose, California. He was a cofounder and vice president of Palyn in 1973 while on leave from Johns Hopkins University.

Flynn worked at IBM for ten years in the areas of computer organization and design. He was design manager of prototype versions of the IBM 7090 and 7094/II, and later for the System 360 Model 91 CPU.

Flynn received his PhD from Purdue University in 1961.



Chad L. Mitchell is currently Chief Technical Officer at Great Wave Software, which he helped found in 1984. He has written several programs for personal computers, including the award-winning Concertware music system.

Mitchell received a BA in mathematics in 1979 and a BS in computer science in 1980 from the University of Utah, both Magna Cum Laude. He received an MS in computer engineering in 1982 and a PhD in computer science in 1986 from Stanford University. While there, he coauthored the Gambit prototyping language.

Johannes M. Mulder is currently completing the requirements for the PhD degree at Stanford University. His primary research interests are in computer architecture, VLSI, and computer and memory system design and evaluation.

Mulder received the MS degree in electrical engineering in 1982 from Delft University of Technology in the Netherlands.

Readers may write to Flynn at Electrical Engineering Dept., ERL 452, Stanford University, Stanford, CA 94305-4055.