Reduced Instruction Set Computer Architecture

WILLIAM STALLINGS, SENIOR MEMBER, IEEE

Since the earliest days of the computer era, the general trend in computer architecture and organization has been toward increasing CPU complexity: larger instruction sets, more addressing modes, more specialized registers, and the like. However, in the past several years, there has been increasing interest in an innovative approach to computer architecture: the reduced instruction set computer (RISC). The intended performance benefits of RISC, compared to a more conventional approach, include more effective compilers, no use of microcode, more effective pipelining, and improved response to interrupts. Key characteristics of the RISC approach include: a limited and simple instruction set; the use of either a large number of registers (hundreds) or an optimizing compiler, to maximize the use of registers and minimize references to main memory; an emphasis on optimizing the instruction execution pipeline.

This paper presents a tutorial on the RISC approach and highlights the key design issues involved in RISC architecture. We begin by looking at the results of a number of studies on the instruction execution characteristics of compiled high-level language programs. The results of these studies inspired the RISC movement. The paper then summarizes approaches to three key RISC design issues: optimized register usage, reduced instruction sets, and pipelining. As examples, an experimental system, the Berkeley RISC, and a commercial system, the MIPS R2000, are presented. The paper closes with a discussion of the RISC versus CISC (complex instruction set computer) controversy.

I. Introduction

Since the development of the stored-program computer around 1950, there have been remarkably few true innovations in the areas of computer organization and architecture. One of the most interesting and, potentially, one of the most important innovations is the reduced instruction set computer (RISC). The RISC architecture is a dramatic departure from the historical trend in CPU architecture and challenges the conventional wisdom expressed in words and deeds by most computer architects. An analysis of the RISC architecture brings into focus many of the important issues in computer organization and architecture.

Most of the work has been on experimental systems, but commercial RISC systems have begun to appear [1]–[12].

Manuscript received December 3, 1986; revised August 21, 1987. The submission of this paper was encouraged after review of an advance proposal.

The author is at 5 Chesterford Gardens, London NW3 7DD, England.

IEEE Log Number 8717881.

Recently, both IBM (with its RT PC) and Hewlett-Packard (with its 900 series) have introduced machines that have both RISC and conventional characteristics [13], [14]. Although RISC systems have been defined and designed in a variety of ways by different groups, the key elements shared by most (not all) designs are these:

- a limited and simple instruction set;
- the use of either a hardware or compiler strategy to maximize the use of registers and minimize references to main memory;
- am emphasis on optimizing the instruction execution pipeline.

This paper surveys key design issues relating to RISC architecture. To begin, we present a brief survey of some results on instruction sets that inspired much of the RISC work.

II. Instruction Execution Characteristics

One of the most visible forms of evolution associated with computers is that of programming languages. As the cost of hardware has dropped, the relative cost of software has risen. Along with that, a chronic shortage of programmers has driven up software costs in absolute terms. Thus the major cost in the life cycle of a system is software, not hardware. Adding to the cost, and to the inconvenience, is the element of unreliability: It is common for programs, both system and application, to continue to exhibit new bugs after years of operation.

The response from researchers and industry has been to develop ever more powerful and complex high-level programming languages (compare Fortran to Ada). These high-level languages (HLL) allow the programmer to express algorithms more concisely, take care of much of the detail, and often support naturally the use of structured programming.

Alas, this solution gave rise to another problem, known as the *semantic gap*, the difference between the operations provided in HLLs and those provided in computer architecture. Symptoms of this gap are alleged to include execution inefficiency, excessive program size, and compiler complexity. Designers responded with architectures intended to close this gap. Key features include large

0018-9219/88/0100-0038$01.00 © 1988 IEEE

instruction sets, dozens of addressing modes, and various HLL statements implemented in hardware. An example of the latter is the CASE machine instruction on the VAX-11. Such complex instruction sets are intended to

- ease the task of the compiler writer;
- improve execution efficiency, since complex sequences of operations can be implemented in microcode;
- provide support for even more complex and sophisticated HLLs.

Meanwhile, a number of studies have been done over the years to determine the characteristics and patterns of execution of machine instructions generated from HLL programs. The results of these studies inspired some researchers to look for an altogether different approach: namely, to make the architecture that supports the HLL simpler, rather than more complex.

So, to understand the line of reasoning of the RISC advocates, we begin with a brief review of instruction execution characteristics. The aspects of computation of interest are

- *Operations Performed:* These determine the functions to be performed by the CPU and its interaction with memory.
- *Operands Used:* The types of operands and the frequency of their use determine the memory organization for storing them and the addressing modes for accessing them.
- *Execution Sequencing:* This determines the control and pipeline organization.

In the remainder of this section, we summarize the results of a number of studies of high-level language programs. All of the results are based on dynamic measurements [15]. That is, measurements are collected by executing the program and counting the number of times some feature has appeared or a particular property has held true. In contrast, static measurements merely perform these counts on the source text of a program. They give no useful information on performance, because they are not weighted relative to the number of times each statement is executed.

A. Operations

A variety of studies have been made to analyze the behavior of HLL programs. Table 1 includes key results from the following studies. The earliest study of programming language, performed by Knuth [16] examined a collection of Fortran programs used as student exercises. Dynamic measurements showed that two-thirds of all statements were assignment and, of these one-third were of the type $A = B$. The remainder seldom had more than one operator. Tanenbaum [17] published measurements of HLL constructs, collected from over 300 procedures used in operating-system programs and written in a language that supports structured programming (SAL). Patterson and Sequin [18], two of the key figures in the Berkeley RISC project, analyzed a set of measurements taken in the early stages of the RISC effort. Measurements were collected from compilers and from programs for typesetting, CAD, sorting, and file comparison. The programming languages C and Pascal were studied. Huck [19] analyzed four programs intended to represent a mix of general-purpose and scientific computing, including fast Fourier transform and integration of systems of differential equations.

Table 1 Relative Dynamic Frequency of High-Level Language Operations

Study Language Workload	[19] Pascal Scientific	[16] Fortran Student	[18] Pascal System	[18] C System	[17] SAL System
ASSIGN	74	67	45	38	42
LOOP	4	3	5	3	4
CALL	1	3	15	12	12
IF	20	11	29	43	36
GOTO	2	9		3	
Other		7	6	1	6

There is quite good agreement in the results of this mixture of languages and applications. Assignment statements predominate, suggesting that the simple movement of data is of high importance. There is also a preponderance of conditional statements (IF, LOOP). These statements are implemented in machine language with some sort of compare and branch instruction. This suggests that the sequence control mechanism of the instruction set is important.

These results are instructive to the machine instruction set designer, indicating which types of statements occur most often and therefore should be supported in an "optimal" fashion. However, these results do not reveal which statements use the most time in the execution of a typical program. That is, given a compiled machine language program, which statements in the source language cause the execution of the most machine-language instructions?

To get at this underlying phenomenon, the Patterson programs [18] were compiled on the VAX, PDP-11, and Motorola 68000 to determine the average number of machine instructions and memory references per statement type. By multiplying the frequency of occurrence of each statement type by these averages, Table 2 is obtained. Columns 2 and 3 provide surrogate measures of the actual time spent executing the various statement types. The results suggest that the procedure call/return is the most time-consuming operation in typical HLL programs.

Table 2 Weighted Relative Dynamic Frequency of all Operations

	Dynamic Occurence		Machine-Instruction Weighted		Memory Reference Weighted	
	Pascal	C	Pascal	C	Pascal	C
ASSIGN	45	38	13	13	14	15
LOOP	5	3	42	32	33	26
CALL	15	12	31	33	44	45
IF	29	43	11	21	7	13
GOTO		3				
Other	6	1	3	1	2	1

Source: [18]

The reader should be clear on the significance of Table 2. This table indicates the relative significance of various statement types in an HLL, when that HLL is compiled for a typical contemporary instruction set architecture. Some other architecture could conceivably produce different results. However, this study produces results that are representative for contemporary complex instruction set com-

puter (CISC) architectures. Thus they can provide guidance to those looking for more efficient ways to support HLLs.

B. Operands

Much less work has been done on the occurrence of types of operands, despite the importance of this topic. There are several aspects that are significant.

The Patterson study already referenced [18] also looked at the dynamic frequency of occurrence of classes of variables (Table 3). The results, consistent between Pascal and C programs, show that the majority of references are to simple scalar variables. Further, over 80 percent of the scalars were local (to the procedure) variables. In addition, references to arrays/structures require a previous reference to their index or pointer, which again is usually a local scalar. Thus there is a preponderance of references to scalars, and these are highly localized.

Table 3 Dynamic Percentage of Operands

	Pascal	C	Average
Integer constant	16	23	20
Scalar Variable	58	53	55
Array/structure	26	24	25

The Patterson study examined the dynamic behavior of HLL programs, independent of the underlying architecture. As discussed earlier, it is necessary to deal with actual architectures to examine program behavior more deeply. One study, [20], examined DEC-10 instructions dynamically and found that each instruction on the average references 0.5 operands in memory and 1.4 registers. Similar results are reported in [19] for C, Pascal, and Fortran programs on S/370, PDP-11, and VAX-11. Of course, these figures depend highly on both the architecture and the compiler, but they do illustrate the frequency of operand accessing.

These latter studies suggest the importance of an architecture that lends itself to fast operand accessing, since this operation is performed so frequently. The Patterson study suggests that a prime candidate for optimization is the mechanism for storing and accessing local scalar variables.

C. Procedure Calls

We have seen that procedure calls and returns are an important aspect of HLL programs. The evidence (Table 2) suggests that these are the most time-consuming operations in the compiled HLL programs. Thus it will be profitable to consider ways of implementing these operations efficiently. Two aspects are significant: the number of parameters and variables that a procedure deals with, and the depth of nesting.

In Tanenbaum's study [17], he found that 98 percent of dynamically called procedures were passed fewer than six arguments, and that 92 percent of them used fewer than six local scalar variables. Similar results were reported by the Berkeley RISC team [21], as shown in Table 4. These results show that the number of words required per procedure activation is not large. The studies reported earlier indicated that a high proportion of operand references are to local scalar variables. The studies just mentioned show that those references are, in fact, confined to relatively few variables.

Table 4 Procedure Arguments and Local Scalar Variables

Percentage of Executed Procedure Calls with	Compiler, Interpreter and Typesetter (percent)	Small Nonnumeric Programs (percent)
> 3 arguments	0–7	0–5
> 5 arguments	0–3	0
> 8 words of arguments and local scalars	1–20	0–6
> 12 words of arguments and local scalars	1–6	0–3

The same Berkeley group also looked at the pattern of procedure calls and returns in HLL programs. They found that it is rare to have a long uninterrupted sequence of procedure calls followed by the corresponding sequence of returns. Rather, they found that a program remains confined to a rather narrow window of procedure-invocation depth. This is illustrated in Fig. 1 [22]. The graph illustrates call–return behavior. Each call is represented by the line moving down to the right, and each return by the line moving up and to the right. In the figure, a *window* with depth equal to 5 is defined. Only a sequence of calls and returns with a net movement of 6 in either direction causes the window to move. As can be seen, the executing program can remain within this window for quite long periods of time. The Berkeley results (for C and Pascal) showed that a window of depth 8 will need to shift only on less than 1 percent

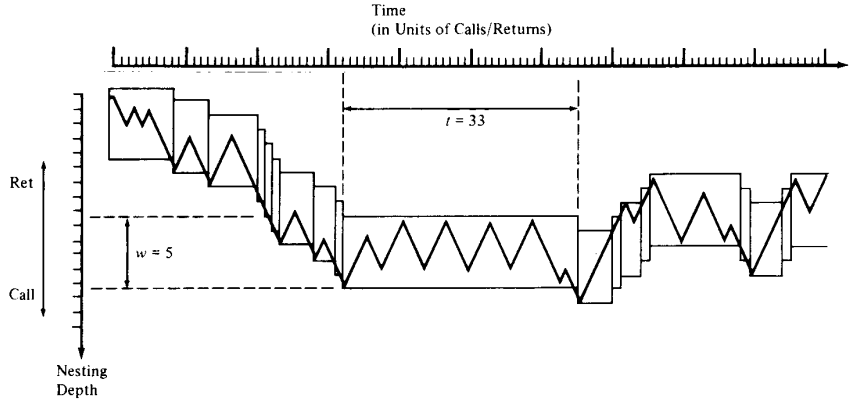


Fig. 1. The call-return behavior of programs.

of the calls or returns [23]. These results also suggest that operand references are highly localized.

D. Implications

A number of groups have looked at results such as those just reported and have concluded that the attempt to make the instruction set architecture close to HLLs is not the most effective design strategy. Rather, the HLLs can best be supported by optimizing performance of the most time-consuming features of typical HLL programs.

Generalizing from the work of a number of researchers, three elements emerge that, by and large, characterized RISC architectures. First, use a large number of registers. This is intended to optimize operand referencing. The studies just discussed show that there are several references per HLL instruction, and that there is a high proportion of move (assignment) statements. This, coupled with the locality and predominance of scalar references, suggests that performance can be improved by reducing memory references at the expense of more register references. Because of the locality of these references, an expanded register set seems practical.

Second, careful attention needs to be paid to the design of instruction pipelines. Because of the high proportion of conditional branch and procedure call instructions, a straightforward instruction pipeline will be inefficient. This manifests itself as a high proportion of instructions that are pre-fetched but never executed.

Finally, a simplified (reduced) instruction set is indicated. This point is not as obvious as the others, but should become clearer in the ensuing discussion. In addition, we will see that the desire to implement an entire CPU on a single chip leads to a reduced instruction set solution.

III. Optimized Register Usage

The results summarized above point out the desirability of quick access to operands that are referenced frequently. We have seen that there is a large proportion of assignment statements in HLL programs, and many of these are of the simple form $A = B$. Also, there are a significant number of operand accesses per HLL statement. If we couple these results with the fact that most accesses are to local scalars, heavy reliance on register storage is suggested.

The reason that register storage is indicated is that it is the fastest available storage device, faster than both main memory and cache. The register file is physically small, generally on the same chip as the ALU and control unit, and employs much shorter addresses than addresses for cache and memory. Thus a strategy is needed that will allow the most frequently accessed operands to be kept in registers and to minimize register-memory operations.

Two basic approaches are possible, one based on software and the other on hardware. The software approach is to rely on the compiler to maximize register usage. The compiler will attempt to allocate registers to those variables that will be used the most in a given time period. This approach requires the use of sophisticated program-analysis algorithms. The hardware approach is simply to use more registers so that more variables can be held in registers for longer periods of time. This section presents both approaches.

To provide some context for this section, the following subsections discuss design issues related to CPU registers.

A. Registers

To understand the role of registers in the CPU, let us consider the requirements placed on the CPU, the things that it must do:

- *Fetch instructions:* The CPU must read instructions from memory.
- *Interpret instructions:* The instruction must be decoded to determine what action is required.
- *Fetch data:* The execution of an instruction may require reading data from memory or an I/O module.
- *Process data:* The execution of an instruction may require performing some arithmetic or logical operation on data.
- *Write data:* The results of an execution may require writing data to memory or an I/O module.

To be able to do these things, it should be clear that the CPU needs to temporarily store some data. The CPU must remember the location of the last instruction so that it can know where to get the next instruction. It needs to store instructions and data temporarily while an instruction is being executed. In other words, the CPU needs a small internal memory. This memory consists of a set of high-speed registers. The registers in the CPU serve two functions:

- *User-visible registers:* These enable the machine- or assembly-language programmer to minimize main-memory references by optimizing use of registers.
- *Control and status registers:* These are used by the control unit to control the operation of the CPU and by privileged, operating system programs to control the execution of programs.

There is no clean separation of registers into these two categories. For example, on some machines the program counter is user-visible (e.g., VAX-11 architecture), but on many it is not. For purposes of the following discussion, however, we will use these categories.

B. User-Visible Registers

A user-visible register is one which may be referenced by means of the machine language that the CPU executes. Virtually all contemporary CPU designs provide for a number of user-visible registers, as opposed to a single accumulator. We can characterize these in the following categories:

- General Purpose
- Data
- Address
- Condition Codes.

General-purpose registers can be assigned to a variety of functions by the programmer. Sometimes, their use within the instruction set is orthogonal to the operation; that is, any general-purpose register can contain the operand for any opcode. This provides true general-purpose register use. Often, however, there are restrictions. For example, there may be dedicated registers for floating-point operations.

In some cases, general-purpose registers can be used for addressing functions (e.g., register indirect, displacement). In other cases, there is a partial or clean separation between data registers and address registers. *Data registers* may only be used to hold data and cannot be employed in the calculation of an operand address. *Address registers* may themselves be somewhat general-purpose, or they may be

devoted to a particular addressing mode. Examples of registers are as follows:

- *Segment pointers:* In a machine with segmented addressing, a segment register holds the address of the base of the segment. There may be multiple registers, for example, one for the operating system and one for the current process.
- *Index registers:* These are used for indexed addressing, and may be autoindexed.
- *Stack pointer:* If there is user-visible stack addressing, then typically the stack is in memory and there is a dedicated register that points to the top of the stack. This allows implicit addressing; that is, push, pop, and other stack instructions need not contain an explicit stack operand.

There are several design issues to be addressed here. An important one is whether to use completely general-purpose registers or to specialize their use. With the use of specialized registers, it can generally be implicit in the opcode which type of register a certain operand specifier refers to. The operand specifier must only identify one of a set of specialized registers rather than one out of all the registers, thus saving bits. On the other hand, this specialization limits the programmer's flexibility. There is no final and best solution to this design issue, but, the trend seems to be toward the use of specialized registers.

Another design issue is the number of registers, either general-purpose or data-plus-address, to be provided. Again, this affects instruction set design since more registers require more operand specifier bits. Somewhere between 8 and 32 registers appears optimum [20]. Fewer registers result in more memory references; more registers do not noticeably reduce memory references. However, a new approach, which finds advantage in the use of hundreds of registers, is exhibited in some RISC systems.

Finally, there is the issue of register length. Registers that must hold addresses obviously must be at least long enough to hold the largest address. Data registers should be able to hold values of most data types. Some machines allow two contiguous registers to be used as one for holding double-length values.

A final category of registers, which is at least partially visible to the user, holds *condition codes* (also referred to as flags). Condition codes are bits set by the CPU hardware as the result of operations. For example, an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set. The code may subsequently be tested as part of a conditional branch operation.

Condition code bits are collected into one or more registers. Usually, they form part of a control register. Generally, machine instructions allow these bits to be read by implicit reference, but they cannot be altered by the programmer.

In some machines, a subroutine call will result in the automatic saving of all user-visible registers, which are to be restored on return. The saving and restoring is performed by the CPU as part of the execution of call-and-return instructions. This allows each subroutine to use the user-visible registers independently. On other machines, it is the responsibility of the programmer to save the contents of the relevant user-visible registers prior to a subroutine call by including instructions for this purpose in the program.

C. The Hardware Approach

The hardware approach has been pioneered by the Berkeley RISC group [18] and is used in the first commercial RISC product, the Pyramid [24].

Register Windows: On the face of it, the use of a large set of registers should decrease the need to access memory. The design task is to organize the registers in such a fashion that this goal is realized.

Since most operand references are to local scalars, the obvious approach is to store these in registers, with perhaps a few registers reserved for global variables. The problem is that the definition of *local* changes with each procedure call and return, operations that occur frequently. On every call, local variables must be saved from the registers into memory, so that the registers can be reused by the called program. Furthermore, parameters must be passed. On return, the variables of the parent program must be restored (loaded back into registers) and results must be passed back to the parent program.

The solution is based on two other results reported above. First, a typical procedure employs only a few passed parameters and local variables. Second, the depth of procedure activation fluctuates within a relatively narrow range (Fig. 1). To exploit these properties, multiple small sets of registers are used, each assigned to a different procedure. A procedure call automatically switches the CPU to use a different fixed-size window of registers, rather than saving registers in memory. Windows for adjacent procedures are overlapped to allow parameter passing.

The concept is illustrated in Fig. 2. At any time, only one window of registers is visible and is addressable as if it were the only set of registers (e.g., address 0 through $N - 1$). The window is divided into three fixed-size areas. Parameter registers hold parameters passed down from the procedure that called the current procedure and results to be passed back up. Local registers are used for local variables, as assigned by the compiler. Temporary registers are used to

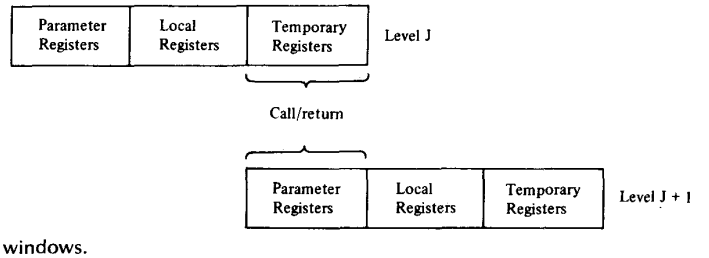


Fig. 2. Overlapping register windows.

exchange parameters and results with the next lower level (procedure called by current procedure). The temporary registers at one level are physically the same as the parameter registers at the next lower level. This overlap permits parameters to be passed without the actual movement of data.

To handle any possible pattern of calls and returns, the number of register windows would have to be unbounded. Instead, the register windows can be used to hold the few most recent procedure activations. Older activations must be saved in memory and later restored when the nesting depth decreases. Thus the actual organization of the register file is as a circular buffer of overlapping windows.

This organization is shown in Fig. 3 [21], which depicts a circular buffer of six windows. The buffer is filled to a depth

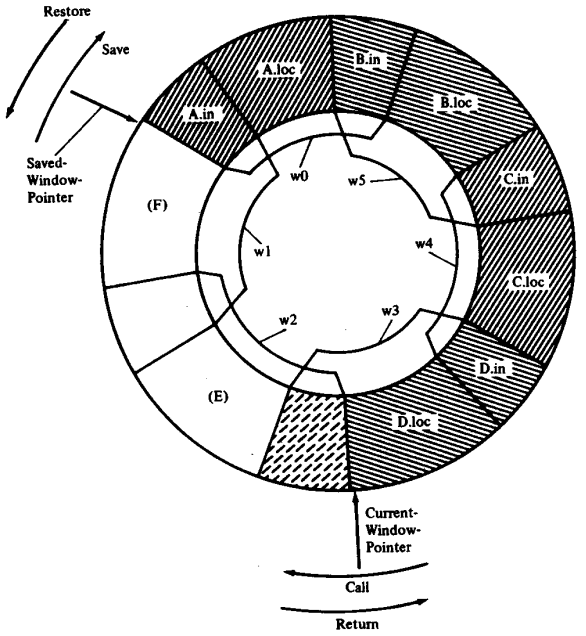


Fig. 3. Circular buffer organization of overlapped windows.

of 4 (*A* called *B*; *B* called *C*; *C* called *D*) with procedure *D* active. The current-window pointer (CWP) points to the window of the currently active procedure. Register references by a machine instruction are offset by this pointer to determine the actual physical register. The saved-window pointer identifies the window most recently saved in memory. If procedure *D* now calls procedure *E*, arguments for *E* are placed in *D*'s temporary registers (the overlap between *w*3 and *w*4) and the CWP is advanced by one window.

If procedure *E* then makes a call to procedure *F*, the call cannot be made with the current status of the buffer. This is because *F*'s window overlaps *A*'s window. If *F* begins to load its temporary registers, preparatory to a call, it will overwrite the parameter registers of *A* (A.param). Thus when CWP is incremented (modulo 6) so that it becomes equal to SWP, an interrupt occurs and *A*'s window is saved. Only the first two portions (A.param and A.loc) need be saved. Then the SWP is incremented and the call to *F* proceeds. A similar interrrupt can occur on returns. For example, subsequent to the activation of *F*, when *B* returns to *A*, CWP is decremented and becomes equal to SWP. This causes an interrupt that results in the restoral of *A*'s window.

From the preceding, it can be seen that an *N*-window register file can hold only $N - 1$ procedure activations. The value of *N* need not be large. As was mentioned earlier, one study [23] found that, with eight windows, a save or restore is needed on only 1 percent of the calls or returns. The Berkeley RISC computers use 8 windows of 16 registers each. The Pyramid computer employs 16 windows of 32 registers each.

Global Variables: The window scheme just described provides an efficient organization for storing local scalar variables in registers. However, this scheme does not address the need to store global variables, those accessed by more than one procedure (e.g., COMMON variables in Fortran). Two options suggest themselves. First, variables declared as global in an HLL can be assigned memory locations by the compiler, and all machine instructions that reference these variables will use memory-reference operands. This is straightforward, from both the hardware and software (compiler) points of view. However, for frequently accessed global variables, this scheme is inefficient.

An alternative is to incorporate a set of global registers in the CPU. These registers would be fixed in number and available to all procedures. A unified numbering scheme can be used to simplify the instruction format. For example, references to registers 0 through 7 could refer to unique global registers, and references to registers 8 through 31 could be offset to refer to physical registers in the current window. Thus there is an increased hardware burden to accommodate the split in register addressing. In addition, the compiler must decide which global variables should be assigned to registers.

Large Register File versus Cache: The register file, organized into windows, acts as a small, fast buffer for holding a subset of all variables that are likely to be used most heavily. From this point of view, the register file acts much like a cache memory. The question therefore arises as to whether it would be simpler and better to use a cache and a small traditional register file.

Table 5 compares characteristics of the two approaches. The window-based register file holds all of the local scalar variables (except in the rare case of window overflow) of the most recent $N - 1$ procedure activations. The cache holds a selection of recently used scalar variables. The register files should save time, since all local scalar variables are retained. On the other hand, the cache may make more efficient use of space, since it is reacting to the situation dynamically. Furthermore, caches generally treat all memory references alike, including instructions and other types

Table 5 Characteristics of Large Register File and Cache Organizations

Large Register File	Cache
All Local Scalars	Recently Used Local Scalars
Individual Variables	Blocks of Memory
Compiler-Assigned Global Variables	Recently Used Global Variables
Save/Restore Based on Procedure Nesting Depth	Save/Restore Based on Cache Replacement Algorithm
Register Addressing	Memory Addressing

of data. Thus savings in these other areas are possible with a cache and not a register file.

A register file may make inefficient use of space, since not all procedures will need the full window space allotted to them. On the other hand, the cache suffers from another sort of inefficiency: Data are read in blocks. Whereas the register file contains only those variables in use, the cache reads in a block of data, some or much of which will not be used.

The cache is capable of handling global as well as local variables. There are usually many global scalars, but only a few of them are heavily used [21]. A cache will dynamically discover these variables and hold them. If the window-based register file is supplemented with global registers, it too can hold some global scalars. However, it is difficult for a compiler to determine which globals will be heavily used.

With the register file, the movement of data between registers and memory is determined by the procedure nesting depth. Since this depth usually fluctuates within a narrow range, the use of memory is relatively infrequent. Most cache memories are set-associative with a small set size. Thus there is the danger that other data or instructions will displace, in the cache, frequently used variables.

Based on the discussion so far, the choice between a large window-based register file and a cache is not clear cut. There is one characteristic, however, in which the register approach is clearly superior and which suggests that a cache-based system will be noticeably slower. This distinction shows up in the amount of addressing overhead experienced by the two approaches.

Fig. 4 [25] illustrates the difference. To reference a local scalar in a window-based register file, a "virtual" register number and a window number are used. These can pass through a relatively simple decoder to select one of the physical registers. To reference a memory location in cache,

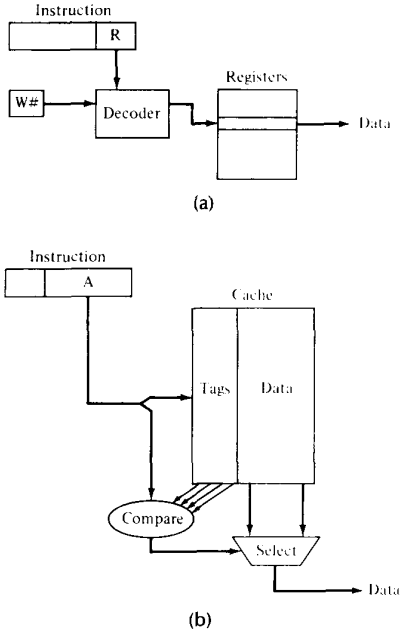


Fig. 4. Referencing a local scalar. (a) Window-based register file. (b) cache.

a full-width memory address must be generated. The complexity of this operation depends on the addressing mode. In a set-associative cache, a portion of the address is used to read a number of words and tags equal to the set size. Another portion of the address is compared to the tags, and one of the words that was read is selected. It should be clear that even if the cache is as fast as the register file, the access time will be considerably longer. Thus from the point of view of performance, the window-based register file is superior for local scalars. Further performance improvement could be achieved by the addition of a cache for instructions only.

D. The Compiler Approach

Let us assume now that only a small number (e.g., 16–32) of registers is available on the target RISC machine. In this case, optimized register usage is the responsibility of the compiler. A program written in a high-level language has, of course, no explicit references to register. Rather, program quantities are referred to symbolically. The objective of the compiler is to keep as many computations as possible in registers rather than main memory, and to minimize load-and-store operations.

In general, the approach taken is as follows. Each program quantity that is a candidate for residing in a register is assigned to a symbolic or virtual register. The compiler then maps the unlimited number of symbolic registers into a fixed number of real registers. Symbolic registers whose usage does not overlap can share the same real register. If, in a particular portion of the program, there are more quantities to deal with than real registers, then some of the quantities are assigned to memory locations. Load-and-store instructions are used to temporarily position quantities in registers for computational operations.

The essence of the optimization task is to decide which quantities are to be assigned to registers at any given point in the program. The technique most commonly used in RISC compilers is known as graph coloring, which is a technique borrowed from the discipline of topology [26]–[28].

The graph coloring problem is this. Given a graph consisting of nodes and edges, assign colors to nodes such that adjacent nodes have different colors, and do this in such a way as to minimize the number of different colors. This problem is adapted to the compiler problem in the following way. First, the program is analyzed to build a register interference graph. The nodes of the graph are the symbolic registers. If two symbolic registers are "live" during the same program fragment, then they are joined by an edge to depict interference. An attempt is then made to color the graph with N colors, where N is the number of registers. If this fails, then nodes that cannot be colored must be placed in memory, and loads and stores must be used to make space for the affected quantities when they are needed.

Fig. 5 is a simple example of the process. Assume a program with six symbolic registers to be compiled onto a machine with three active registers. Fig. 5(a) shows the time sequence of active use of each symbolic register, and Fig. 5(b) shows the register interference graph. A possible coloring with three colors is indicated. One symbolic register, F, is left uncolored and must be dealt with using loads and stores.

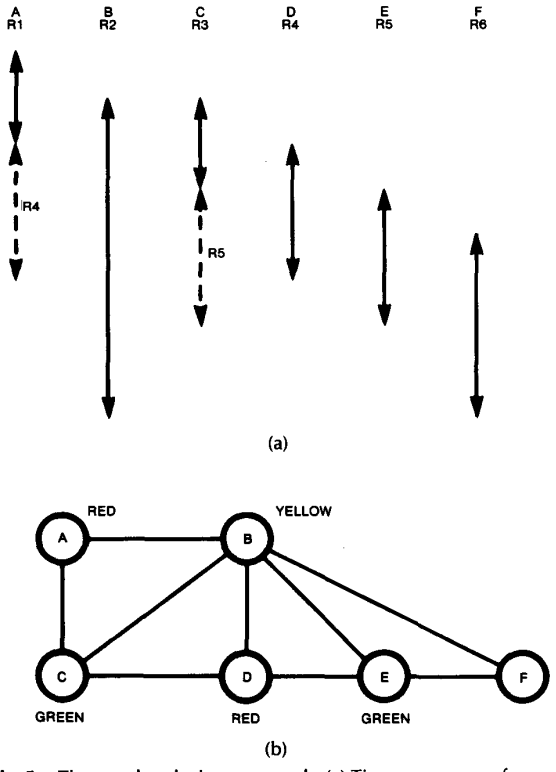


Fig. 5. The graph coloring approach. (a) Time sequence of active use of symbolic registers. (b) Register interference graph.

IV. Reduced Instruction Set Architecture

In this section, we look at some of the general characteristics of and the motivation for a reduced instruction set architecture. Specific examples will be seen later in this paper. We begin with a discussion of motivations for contemporary complex instruction set architectures.

A. The CISC Approach

We have noted the trend to richer instruction sets, which include a larger number of instructions and more-complex instructions. Two principal reasons have motivated this trend: a desire to simplify compilers and a desire to improve performance. Underlying both of these reasons was the shift to high-level languages (HLL) on the part of programmers; architect attempted to design machines that provided better support for HLLs.

It is not the intent of this paper to say that the CISC designers took the wrong direction. RISC technology is very new and so the CISC versus RISC debate cannot now be settled. Indeed, because technology continues to evolve and because architectures exist along a spectrum rather than in two neat categories, a black-and-white assessment is unlikely ever to emerge. Thus the comments that follow are simply meant to point out some of the potential pitfalls in the CISC approach and to provide some understanding of the motivation of the RISC adherents.

The first of the reasons cited, compiler simplification, seems obvious. The task of the compiler writer is to generate a sequence of machine instructions for each HLL statement. If there are machine instructions that resemble HLL statements, this task is simplified. This reasoning has been disputed by the RISC researchers [29]–[31]. They have found that complex machine instructions are often hard to exploit since the compiler must find those cases that exactly fit the construct. The task of optimizing the generated code to minimize code size, reduce instruction execution count, and enhance pipelining is much more difficult with a complex instruction set. As evidence of this, studies cited earlier in this paper indicate that most of the instructions in a compiled program are the relatively simple ones.

The other major reason cited is the expectation that a CISC will yield smaller, faster programs. Let us examine both aspects of this assertion: that programs will be smaller and that they will execute faster.

There are two advantages to smaller programs. First, because the program takes up less memory, there is a savings in that resource. With memory today being so inexpensive, this potential advantage is no longer compelling. More importantly, smaller programs should improve performance, and this will happen in two ways. First, fewer instructions means fewer instruction bytes to be fetched. And second, in a paging environment, smaller programs occupy fewer pages, reducing page faults.

The problem with this line of reasoning is that it is far from certain that a CISC program will be smaller than a corresponding RISC program. In many cases, the CISC program, expressed in symbolic machine language, may be *shorter* (i.e., fewer instructions), but the number of bits of memory occupied may not be noticeably *smaller*. Table 6 shows

Table 6 Code Size Relative to RISC I

	[18] 11 C Programs	[21] 12 C Programs	[38] 5 C Programs
RISC I	1.0	1.0	1.0
VAX-11/780	0.8	0.67	
M68000	0.9		0.9
Z8002	1.2		1.12
PDP-11/70	0.9	0.71	

results from three studies that compared the size of compiled C programs on a variety of machines, including RISC I, which has a reduced instruction set architecture. On the whole, the code savings of CISC versus RISC is less than one might expect. It is also interesting to note that the VAX-11, which has a much more complex instruction set than the PDP-11, achieves very little savings over the latter. These results were confirmed by IBM researchers [30], who found that the IBM 801 (a RISC) produced code that was 0.9 times the size of code on an IBM S/370. The study used a set of PL/I programs.

There are several reasons for these rather surprising results. We have already noted that compilers on CISCs tend to favor simpler instructions, so that the conciseness of the complex instructions seldom comes into play. Also, since there are more instructions on a CISC, longer opcodes are required, producing longer instructions. Finally, RISCs tend to emphasize register rather than memory references, and the former require fewer bits. An example of this last effect is discussed presently (see Fig. 6).

So, the expectation that a CISC will produce smaller programs, with the attendant advantages, may not be realized.

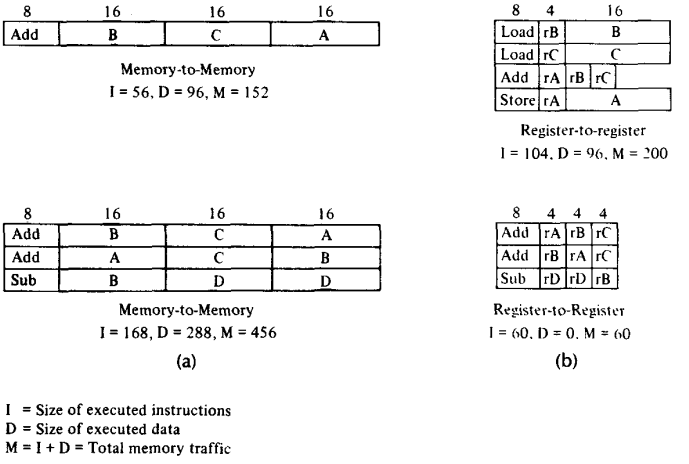


Fig. 6. (a) $A \leftarrow B + C$. (b) $A \leftarrow B + C$; $B \leftarrow A + C$; $D \leftarrow D - B$.

The second motivating factor for increasingly complex instruction sets was that instruction execution would be faster. It seems to make sense that a complex HLL operation will execute more quickly as a single machine instruction rather than as a series of more-primitive instructions. However, because of the bias towards the use of those simpler instructions, this may not be so. The entire control unit must be made more complex, and/or the microprogram control store must be made larger, to accommodate a richer instruction set. Either factor increases the execution time of the simple instructions.

In fact, some researchers have found that the speedup in the execution of complex functions is due not so much to the power of the complex machine instructions as to their residence in high-speed control store [30]. In effect, the control store acts as an instruction cache. Thus the hardware architecture is in the position of trying to determine which subroutines or functions will be used most frequently and assigning those to the control store by implementing them in microcode. The results have been less than encouraging. Thus on S/370 systems, instructions such as Translate and Extended-Precision-Floating-Point-Divide reside in high-speed storage, while the sequence involved in setting up procedure calls or initiating an interrupt handler are in slower main memory.

Thus it is far from clear that the trend to increasingly complex instruction sets is appropriate. This has led a number of groups to pursue the opposite path.

B. Characteristics of Reduced Instruction Set Architecture

Although a variety of different approaches to reduced instruction set architecture have been taken, certain characteristics are common to all of them. These characteristics are listed in Table 7 and described here. Specific examples are explored later in this section.

Table 7 Characteristics of Reduced Instruction Set Architectures

One Instruction Per Cycle
Register-to-Register Operations
Simple Address Modes
Simple Instruction Formats

The first characteristic listed in Table 7 is that one machine instruction is executed per machine cycle. A *machine cycle* is defined to be the time it takes to fetch two operands from registers, perform as ALU operation, and store the result in a register. Thus RISC machine instructions should be no more complicated, than, and execute about as fast as, microinstructions on CISC machines. With simple, one-cycle instructions, there is little or no need for microcode; the machine instructions can be hardwired. Such instructions should execute faster than comparable machine instructions on other machines, since it is not necessary to access a microprogram control store during instruction execution.

A second characteristic is that most operations should be register-to-register, with only simple LOAD and STORE operations accessing memory. This design feature simplifies the instruction set and therefore the control unit. For example, a RISC instruction set may include only one or two ADD instructions (e.g., integer add, add with carry); the VAX-11 has 25 different ADD instructions. Another benefit is that such an architecture encourages the optimization of register use, so that frequently accessed operands remain in high-speed storage.

This emphasis on register-to-register operations is unique to RISC designs. Other contemporary machines provide such instructions but also include memory-to-memory and mixed register/memory operations. Attempts to compare these approaches were made in the 1970s, before the appearance of RISCs. Fig. 6(a) illustrates the approach taken [22]. Hypothetical architectures were evaluated on program size and the number of bits of memory traffic. Results such as this one led one researcher to suggest that future architectures should contain no registers at all [32]. One wonders what he would have thought, at the time, of the RISC machine marketed by Pyramid, which contains no less than 528 registers!

What was missing from these studies was a recognition of the frequent access to a small number of local scalars and that, with a large bank of registers or an optimizing compiler, most operands could be kept in registers for long periods of time. Thus Fig. 6(b) may be a fairer comparison.

Returning to Table 7, a third characteristic is the use of simple addressing modes. Almost all instructions use sim-

ple register addressing. Several additional modes, such as displacement and PC-relative, may be included. Other, more-complex modes can be synthesized in software from the simple ones. Again, this design feature simplifies the instruction set and the control unit.

A final common characteristic is the use of simple instruction formats. Generally, only one or a few formats are used. Instruction length is fixed and aligned on word boundaries. Field locations, especially the opcode, are fixed. This design feature has a number of benefits. With fixed fields, opcode decoding and register operand accessing can occur simultaneously. Simplified formats simplify the control unit. Instruction fetching is optimized since word-length units are fetched. This also means that a single instruction does not cross page boundaries.

Taken together, these characteristics can be assessed to determine the potential benefits of the RISC approach. These benefits fall into two main categories: those related to performance and those related to VLSI implementation.

With respect to performance, a certain amount of "circumstantial evidence" can be presented. First, more-effective optimizing compilers can be developed. With more-primitive instructions, there are more opportunities for moving functions out of the loops, reorganizing code for efficiency, maximizing register utilization, and so forth. It is even possible to compute parts of complex instructions at compile time. For example, the S/370 Move Characters (MVC) instruction moves a string of characters from one location to another. Each time it is executed, the move will depend on the length of the string, whether and in which direction the locations overlap, and what the alignment characteristics are. In most cases, these will all be known at compile time. Thus the compiler could produce an optimized sequence of primitive instructions for this function.

A second point, already noted, is that most instructions generated by a compiler are relatively simple anyway. It would seem reasonable that a control unit built specifically for those instructions and using little or no microcode could execute them faster than a comparable CISC.

A third point relates to the use of instruction pipelining. RISC researchers feel that the instruction pipelining technique can be applied much more effectively with a reduced instruction set. We examine this point in some detail presently.

A final, and somewhat less significant point, is that RISC programs should be more responsive to interrupts since interrupts are checked between rather elementary operations. Architectures with complex instructions either restrict interrupts to instruction boundaries or must define specific interruptable points and implement mechanisms for restarting an instruction.

The case for improved performance for a reduced instruction set architecture is far from proven. A number of studies have been done but not on machines of comparable technology and power. Further, most studies have not attempted to separate the effects of a reduced instruction set and the effects of a large register file. The "circumstantial evidence" however, has been sufficient to encourage the RISC proponents.

The second area of potential benefit, which is more clear-cut, relates to VLSI implementation. When VLSI is used, the design and implementation of the CPU are fundamentally changed. Traditional CPUs, such as the IBM S/370 and the VAX-11, consist of one or more printed circuit boards containing standardized SSI and MSI packages. With the advent of LSI and VLSI, it is possible to put an entire CPU on a single chip. For a single-chip CPU, there are two motivations for following a RISC strategy. First, there is the issue of performance. On-chip delays are of much shorter duration than inter-chip delays. Thus it makes sense to devote scarce chip real estate to those activities that occur frequently. We have seen that simple instructions and access to local scalars are, in fact, the most frequent activities. The Berkeley RISC chips were designed with this consideration in mind. Whereas a typical single-chip microprocessor dedicates about half of its area to the microcode control store, the RISC I chip devotes only about 6 percent of its area to the control unit [33].

A second VLSI-related issue is design-and-implementation time. A VLSI processor is difficult to develop. Instead of relying on available SSI/MSI parts, the designer must perform circuit design, layout, and modeling at the device level. With a reduced instruction set architecture, this process is far easier, as evidenced by Table 8 [34]. If, in addition, the performance of the RISC chip is equivalent to comparable CISC microprocessors, then the advantages of the RISC approach become evident.

Table 8 Design and Layout Effort for Some Microprocessors

CPU	Transistors	Design (Person-Months)	Layout (Person-Months)
RISC I	44	15	12
RISC II	41	18	12
M68000	68	100	70
Z8000	18	60	70
Intel iAPx-432	110	170	90

V. RISC Pipelining

One of the traditional methods of enhancing processor performance is instruction pipelining. The use of a reduced instruction set architecture opens up new opportunities for the effective use of pipelining. To illustrate the significance of pipelining on a RISC machine, we begin with a general discussion.

A. Pipelining Strategy

Instruction pipelining is similar to the use of an assembly line in a manufacturing plant. An assembly line takes advantage of the fact that a product goes through various stages of production. By laying the production process out in an assembly line, products at various stages can be worked on simultaneously. This process is also referred to as *pipelining*, because, as in a pipeline, new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.

To apply this concept to instruction execution, we must recognize that, in fact, the execution of an instruction involves a number of stages. As a simple approach, consider subdividing instruction processing into two stages: fetch instruction and execute instruction. There are times during the execution of an instruction when main memory

is not being accessed. This time could be used to fetch the next instruction in parallel with the execution of the current one. Fig. 7(a) depicts this approach. The pipeline has two independent stages. The first stage fetches an instruction

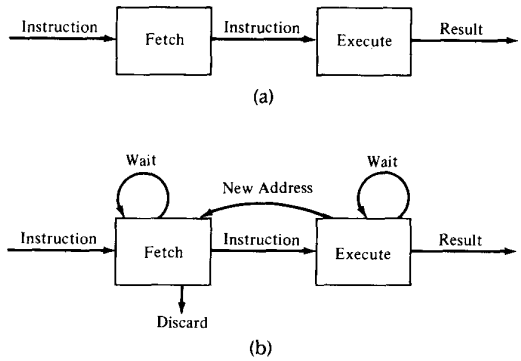


Fig. 7. Two-stage instruction pipeline. (a) Simplified view. (b) Expanded view.

and buffers it. When the second stage is free, the first stage passes the buffered instruction to the second stage. While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch and buffer the next instruction. This is called *instruction prefetch* or *fetch overlap*.

It should be clear that this process will speed up instruction execution. If the fetch and instruction stages were of equal duration, the instruction cycle time would be halved. However, if we look more closely at this pipeline (Fig. 7(b)), we will see that this doubling of execution rate is unlikely for two reasons:

1) The execution time will generally be longer than the fetch time. Execution will involve reading and storing operands and the performance of some operation. Thus the fetch stage may have to wait for some time before it can empty its buffer.

2) A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched.

The time loss from the second reason can be reduced by guessing. A simple rule is the following: When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction. Then, if the branch is not taken, no time is lost. If the branch is taken, the fetched instruction must be discarded and a new instruction fetched.

While these factors reduce the potential effectiveness of the two-stage pipeline, some speedup occurs. To gain further speedup, the pipeline must have more stages. Let us consider the following decomposition of the instruction processing.

- *Fetch Instruction (FI):* Read the next expected instruction into a buffer.
- *Decode Instruction (DI):* Determine the opcode and the operand specifiers.
- *Calculate Operands (CO):* Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of address calculation.
- *Fetch Operands (FO):* Fetch each operand from memory. Operands in registers need not be fetched.
- *Execute Instruction (EI):* Perform the indicated operation and store the result, if any, in the specified destination operand location.

With this decomposition, the various stages will be of more nearly equal duration. For the sake of illustration, let us assume equal duration and assume that only one stage that accesses memory may be active at a time. Then, Fig. 8 illustrates that a five-stage pipeline can reduce the execution time for four instructions from 20 time units to 13 time units. Note that the FI stage always involves a memory access. The FO and EI stages may or may not involve memory access, but the diagram is based on the assumption that they do.

Again, several factors serve to reduce the performance enhancement. If the five stages are not of equal length, there will be some waiting involved at various pipeline stages, as discussed before. A conditional branch instruction can invalidate several instruction fetches. A similar unpredictable event is an interrupt. Fig. 9 indicates the logic needed for pipelining to account for branches and interrupts.

Other problems arise that did not appear in our simple two-stage organization. The CO stage may depend on the contents of a register that could be altered by a previous instruction still in the pipeline. Other such register and memory conflicts could occur. The system must contain logic to account for this type of conflict.

The system should also contain additional logic to improve pipeline efficiency. For example, if an EI stage is not going to access memory, then another memory-accessing stage in another instruction can be performed in parallel.

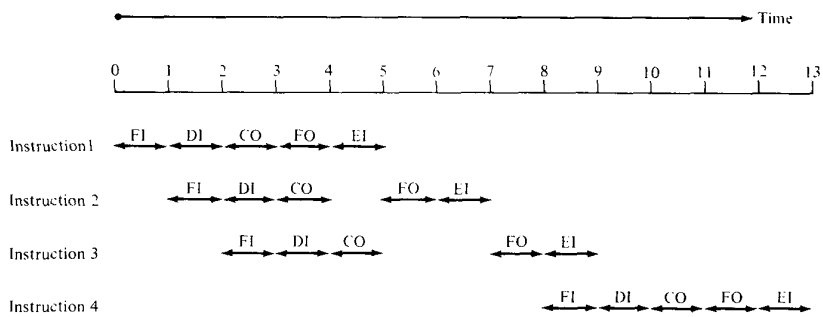


Fig. 8. Timing diagram for pipelined operation.

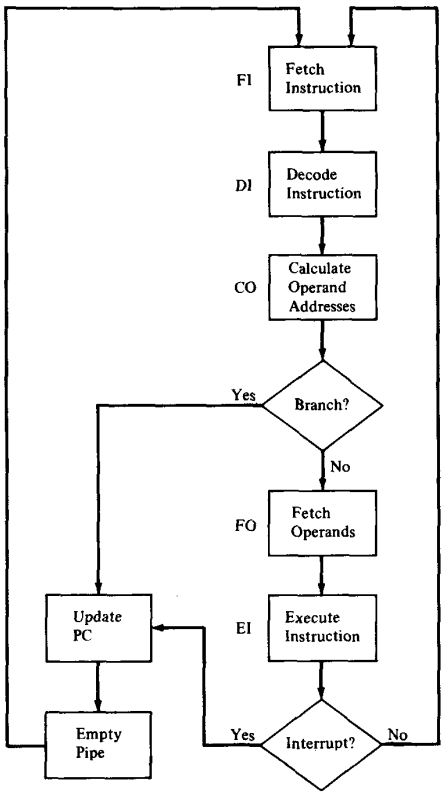


Fig. 9. Five-stage CPU pipeline.

From the preceding discussion, it might appear that the greater the number of stages in the pipeline, the faster the execution rate. However, two factors work against this conclusion:

1) At each stage of the pipeline, there is some overhead involved in moving data from buffer to buffer and in performing various preparation and delivery functions. This overhead can appreciably lengthen the total execution time of a single instruction, which can produce significant delays when the ideal pipeline pattern is not followed either through branching or memory access dependencies.

2) The amount of control logic required to handle memory and register dependencies and to optimize the use of the pipeline increases enormously with the number of stages. This can lead to a situation where the control logic controlling the gating between stages is more complex than the stages being controlled.

Thus instruction pipelining is a powerful technique for enhancing performance but requires careful design to achieve optimum results with reasonable complexity.

B. Dealing with Branches

One of the major problems in designing an instruction pipeline is assuring a steady flow of instructions to the initial stages of the pipeline. The primary impediment, as we have seen, is the conditional branch instruction. Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not.

In what follows, we briefly summarize some of the more common approaches to be taken for dealing with branches.

- *Multiple Streams:* A simple pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and may choose erroneously. A brute-force approach is to allow the pipeline to fetch both instructions, making use of multiple streams. One problem with this approach is that additional branch instructions may enter the pipeline (either stream) before the original branch is resolved. These instructions need their own multiple streams beyond what is supported in the hardware.
- *Prefetch Branch Target:* When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed. If the branch is taken, we have already prefetched the target.
- *Branch Prediction:* Various techniques can be used to predict whether a branch will be taken. These can be based on historical analysis of past executions (e.g., by opcode) or on some dynamic measure of the recent frequency of branching.
- *Delayed Branch:* It is possible to improve pipeline performance by automatically rearranging instructions within a program so that branch instructions occur later than actually desired.

The first three approaches are built into the hardware and are exercised at run time. The last approach listed above is performed at compile time and is used in most of the RISC compilers.

C. Pipelining with RISC Instructions

Let us now consider pipelining in the context of a RISC architecture. Most instructions are register-to-register, and an instruction cycle has the following two phases:

- I: Instruction fetch.
- E: Execute. Performs an ALU operation with register input and output.

For load and store operations, three phases are required:

- I: Instruction fetch.
- E: Execute. Calculates memory address.
- D: Memory. Register-to-memory or memory-to-register operation.

Fig. 10 depicts the timing of a sequence of instructions using no pipelining (Figs. 10–14 are based on [33]). Clearly,

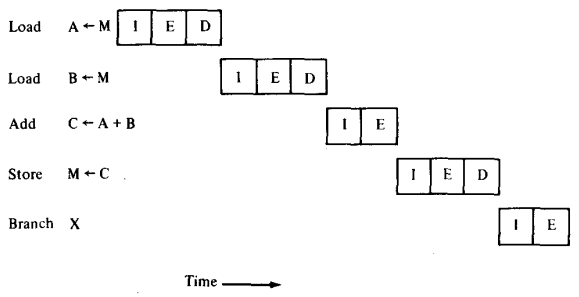


Fig. 10. Timing of sequential execution.

this is a wasteful process. Even very simple pipelining can substantially improve performance. Fig. 11 shows a two-way pipelining scheme, in which the I and E phases of two different instructions are performed simultaneously. This

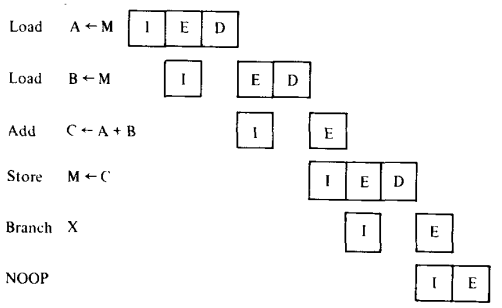


Fig. 11. Two-way pipelined timing.

scheme can yield up to twice the execution rate of a serial scheme. Two problems prevent the maximum speedup from being achieved. First, we assume that a single-port memory is used and that only one memory access is possible per phase. This requires the insertion of a wait state in some instructions. Second, a branch instruction interrupts the sequential flow of execution. To accommodate this with minimum circuitry, a NOOP instruction can be inserted into the instruction stream by the compiler or assembler.

Pipelining can be improved further by permitting two memory accesses per phase. This yields the sequence shown in Fig. 12. Now, up to three instructions can be over-

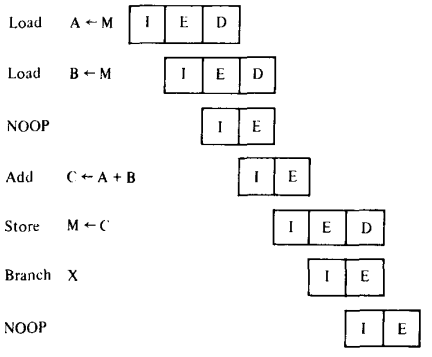


Fig. 12. Three-way pipelined timing.

lapped, and the improvement is as much as a factor of three. Again, branch instructions cause the speedup to fall short of the maximum possible. Also, note that data dependencies have an effect. If an instruction needs an operand that is altered by the preceding instruction, a delay is required. Again, this can be accomplished by a NOOP.

The pipelining discussed so far works best if the three phases are of approximately equal duration. Because the E phase usually involves an ALU operation, it may be longer. In this case, we can divide into two subphases:

- E_1: Register file read.
- E_2: ALU operation and register write.

Because of the simplicity and regularity of the instruction set, the design of the phasing into three or four phases is easily accomplished. Fig. 13 shows the result with a four-

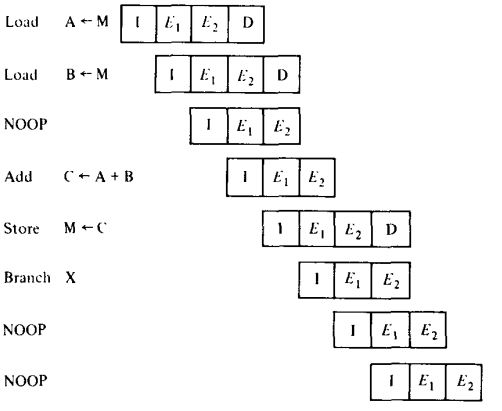


Fig. 13. Four-way pipelined timing.

way pipeline. Up to four instructions at a time can be under way, and the maximum potential speedup is a factor of four. Note again the use of NOOPs to account for data and branch delays.

D. Optimization of Pipelining

Because of the simple and regular nature of RISC instructions, pipelining schemes can be efficiently employed. There are few variations in instruction execution duration, and the pipeline can be tailored to reflect this. However, we have seen that data and branch dependencies reduce the overall execution rate.

To compensate for these dependencies, code reorganization techniques have been developed [35]. First, let us consider branching instructions. *Delayed branch*, a way of increasing the efficiency of the pipeline, makes use of a branch that does not take effect until after the following instruction. This strange procedure is illustrated in Table 9. In the first column, we see a normal symbolic instruction machine-language program. After 102 is executed, the next instruction to be executed is 105. In order to regularize the pipeline, a NOOP is inserted after this branch. However, increased performance is achieved if the instructions at 101 and 102 are interchanged. Fig. 14 shows the result. The JUMP instruction is fetched before the ADD instruction. Note,

Table 9 Normal and Delayed Branch

Address	Normal Branch	Delayed Branch	Optimized Delayed Branch
100	LOAD *X, A*	LOAD *X, A*	LOAD *X, A*
101	ADD 1, *A*	ADD 1, *A*	JUMP 105
102	JUMP 105	JUMP 106	ADD 1, *A*
103	ADD *A, B*	NOOP	ADD *A, B*
104	SUB *C, B*	ADD *A, B*	SUB *C, B*
105	STORE *A, Z*	SUB *C, B*	STORE *A, Z*
106		STORE *A, Z*	

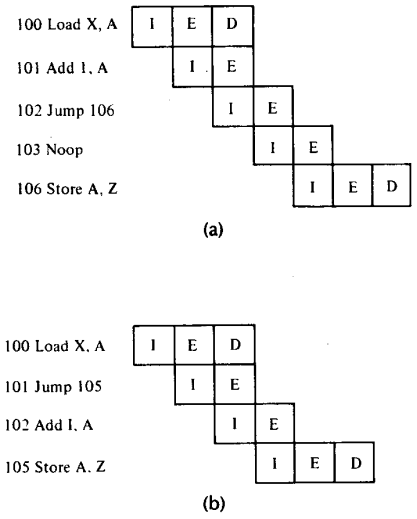


Fig. 14. Execution of delayed jump. (a) Inserted NOOP. (b) Reversed instructions.

however, that the ADD instruction is fetched before the execution of the JUMP instruction has a chance to alter the program counter. Thus the original semantics of the program are retained.

This interchange of instructions will work successfully for unconditional branches, calls, and returns. For conditional branches, this procedure cannot be blindly applied. If the condition that is tested for the branch can be altered by the immediately preceding instruction, then the compiler must refrain from doing the interchange and instead insert a NOOP. The experience of both the Berkeley RISC and IBM 801 systems is that the majority of conditional branch instructions can be optimized in this fashion [18], [30].

A similar sort of tactic, called the delayed load, can be used on LOAD instructions. On LOAD instructions, the register that is to be the target of the load is locked by the CPU. The CPU then continues execution of the instruction stream until it reaches an instruction requiring that register, at which point it idles until the load is complete. If the compiler can rearrange instructions so that useful work can be done while the load is in the pipeline, efficiency is increased.

VI. Example Systems

This section provides two concrete examples of RISC systems, the Berkeley RISC and the R2000 from MIPS Computer Systems. Unlike a number of other machines that present RISC characteristics mixed with CISC characteristics, these are both relatively "pure" RISC systems.

A. Berkeley RISC

The best documented RISC project is that conducted at the University of California at Berkeley. Two similar machines, RISC I and RISC II, were produced [18], [21]. The Berkeley RISC architecture was the inspiration for a commercially available product, the Pyramid [24].

Instruction Set: Table 10 lists the instructions for the Berkeley RISC computers.

As can be seen, most of the instructions reference only register operands. Register-to-register instructions have three operands and can be expressed in the form

$$Rd \leftarrow R_{S1} \text{ op } S2$$

Rd and R_{S1} are register references. S2 can refer either to a register or to a 13-bit immediate operand. Register zero (R_0) is hardwired with the value 0. This form is well suited to typical programs, which have a high proportion of local scalars and constants.

The available ALU operations can be grouped as follows:

- integer addition (with or without carry)
- integer subtraction (with or without carry)
- bitwise Boolean AND, OR, XOR
- shift left logical, right logical, or right arithmetic.

All of these instructions can optionally set the four condition codes (ZERO, NEGATIVE, OVERFLOW, CARRY). Integers are represented in 32-bit 2's-complement form.

Only simple load-and-store instructions reference memory. There are separate load-and-store instructions for word (32 bits), halfword, and byte. For the latter two cases, there are instructions for loading these quantities as signed or unsigned numbers. Signed numbers are sign-extended to fill out the 32-bit destination register. Unsigned numbers are padded with 0s.

On the RISC I, the only available addressing mode, other than register, is a displacement mode. That is, the effective address of an operand consists of a displacement from an address contained in a register:

$$EA = (R_{S1}) + S2$$

or

$$EA = (R_{S1}) + (R_{S2})$$

according as the second operand is immediate or a register reference. To perform a load or store, an extra phase is added to the instruction cycle. During the second phase, the address is calculated using the ALU; the load or store occurs in a third phase. This single addressing mode is quite versatile and can be used to synthesize other addressing modes, as indicated in Table 11.

The RISC II includes an additional version of each load and store instruction using relative address:

$$EA = (PC) = S2.$$

The remaining instructions include control-transfer instructions and some miscellaneous instruction. The control-transfer instructions include conditional jump, call, and conditional return instructions. Both forms of RISC II addressing can be used.

Instruction Format: One of the major factors in the complexity of instruction processing is instruction decoding, especially the task of extracting the various instruction fields. To minimize this chore, the ideal instruction set would use a single fixed-length format with fixed field positions. The RISC instruction set comes close to this goal.

All RISC instructions are a single word (32 bits) in length (Fig. 15). The first 7 bits are the opcode, allowing up to 128 different opcodes. RISC I and RISC II use only 31 and 39 codes, respectively. The SCC bit indicates whether to set the condition codes. The DEST field usually contains a 5-bit destination register reference. For conditional branch instructions, 4 bits of the field designate which condition or conditions are to be tested.

Table 10 RISC Instruction Set

Instruction	Operands	Comments	
ADD	Rs,S2,Rd	Rd←Rs + S2	integer add
ADDC	Rs,S2,Rd	Rd←Rs + S2 + carry	add with carry
SUB	Rs,S2,Rd	Rd←Rs − S2	integer subtract
SUBC	Rs,S2,Rd	Rd←Rs − S2 − carry	subtract with carry
SUBR	Rs,S2,Rd	Rd←S2 − Rs	integer subtract
SUBCR	Rs,S2,Rd	Rd←S2 − Rs − carry	subtract with carry
AND	Rs,S2,Rd	Rd←Rs | S2	logical and
OR	Rs,S2,Rd	Rd←Rs | S2	logical or
XOR	Rs,S2,Rd	Rd←Rs xor S2	logical exclusive or
SLL	Rs,S2,Rd	Rd←Rs shifted by S2	shift left
SRL	Rs,S2,Rd	Rd←Rs shifted by S2	shift right logical
SRA	Rs,S2,Rd	Rd←Rs shifted by S2	shift right arithmetic
LDXW	(Rx)S2,Rd	Rd←M[Rx + S2]	load word
LDXHU	(Rx)S2,Rd	Rd←M[Rx + S2]	load halfword unsigned
LDXHS	(Rx)S2,Rd	Rd←M[Rx + S2]	load halfword signed
LDXBU	(Rx)S2,Rd	Rd←M[Rx + S2]	load byte unsigned
LDYBS	(Rx)S2,Rd	Rd←M[Rx + S2]	load byte signed
STXW	Rm,(Rx)S2	M[Rx + S2]←Rm	store word
STXH	Rm,(Rx)S2	M[Rx + S2]←Rm	store halfword
STXB	Rm,(Rx)S2	M[Rx + S2]←Rm	store byte
LDRW	S2,Rd	Rd←M[PC + S2]	load word relative
LDRHU	S2,Rd	Rd←M[PC + S2]	load halfword unsigned relative
LDRHS	S2,Rd	Rd←M[PC + S2]	load halfword signed relative
LDRBU	S2,Rd	Rd←M[PC + S2]	load byte unsigned relative
LDRBS	S2,Rd	Rd←M[PC + S2]	load byte signed relative
STRW	Rm,S2	M[PC + S2]←Rm	store word
STRH	Rm,S2	M[PC + S2]←Rm	store halfword
STRB	Rm,S2	M[PC + S2]←Rm	store byte
JMP	COND,S2(Rx)	pc←Rx + S2	conditional jump
JMPR	COND,Y	pc←pc + Y	conditional relative
CALL	Rd,S2(Rx)	Rd←pc, next	call
		pc←Rx + S2,CWP←CWP − 1	and change window
CALLR	Rd,Y	Rd←pc, next	call relative
		pc←pc + Y,CWP←CWP − 1	and change window
RET	Rm,S2	pc←Rm + S2,CWP←CWP + 1	return and change window
CALLINT	Rd	Rd←last pc; next CWP←CWP − 1	disable interrupts
RETINT	Rm,S2	pc←Rm + S2; next CWP←CWP + 1	enable interrupts
LDHI	Rd,Y	Rd⟨31:13⟩←Y; Rd⟨12:0⟩←0	load immediate high
GTLPC	Rd	Rd←last pc	to restart delayed jump
GETPSW	Rd	Rd←PSW	load status word
PUTPSW	Rm	PSW←Rm	set status word

Table 11 Synthesizing Other Addressing Modes with RISC Addressing Modes

Mode	Algorithm	RISC Equivalent	Instruction Type
Immediate	operand = A	S2	register-register
Direct	EA = A	$R_0 + S_2$	load, store
Register	EA = R	Rs_1, Rs_2	register-register
Register indirect	EA = (R)	$Rs_1 + 0$	load, store
Displacement	EA = (R) + A	$Rs_1 + S_2$	load, store

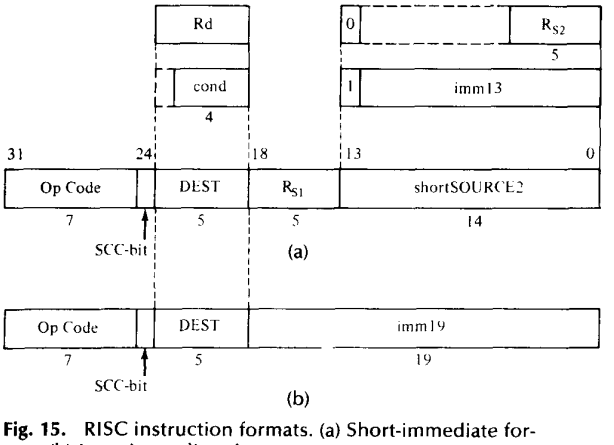


Fig. 15. RISC instruction formats. (a) Short-immediate format. (b) Long-immediate format.

The remaining 19 bits designate one or two operands, depending on opcode. A single 19-bit 2's-complement immediate operand is used for all PC-relative instructions. Otherwise, the first of the two operands is a register reference. The second operand is either a register reference or a 13-bit 2's-complement immediate operand.

Register File: The RISC register file contains 138 registers. Physical registers 0 through 9 are global registers shared by all procedures. The remaining registers are grouped into eight windows. Each process sees logical registers 0 through 31 (Fig. 16). Logical registers 26 through 31 are shared with the calling (parent) procedure, and logical registers 10 through 15 are shared with any called (child) procedure. These two portions overlap with other windows.

Pipelining: The RISC I processor uses a two-stage pipeline, dividing each instruction into fetch and execute states. RISC II uses the three stages. The second stage performs

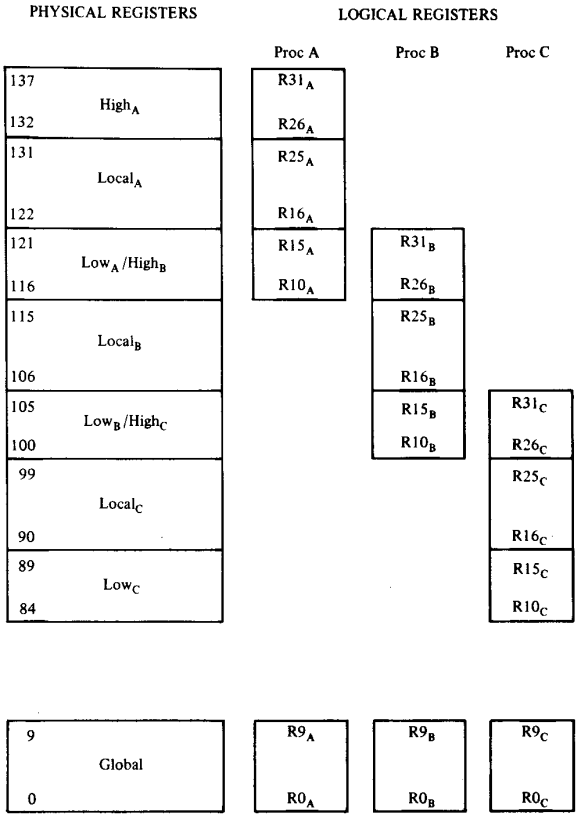


Fig. 16. Berkeley RISC register windows.

ALU operations. The third stage stores a result in Rd or accesses memory with an effective address computed in the second stage.

B. MIPS R2000

One of the first commercially available chip sets was developed by MIPS Computer Systems [36], [37]. The system was inspired by an experimental system, also using the name MIPS, developed at Stanford [38].

The RISC processor chip (called the R2000) is partitioned into two sections, one containing the CPU, and the other containing a coprocessor for memory management. The CPU has a very simple architecture. The intent was to design a system in which the instruction execution logic was as simple as possible, leaving space available for logic to enhance performance (e.g., the entire memory management unit).

The processor supports thirty-two 32-bit registers. It also provides for up to 128 kbytes of high-speed cache, half each for instructions and data. The relatively large cache (the IBM 3090 provides 128–256 kbytes of cache) enables the system to keep large sets of program code and data local to the processor, off-loading the main memory bus and avoiding the need for a large register file with the accompanying windowing logic. All processor instructions are encoded in a single 32-bit word. All data operations are register-to-register; the only memory references are pure load/store operations.

Several features found in other RISC designs are missing in the MIPS machine. As was mentioned, there are only 32 general-purpose registers, all of which are visible at all times; there are no hidden registers and no use of windowing. The optimizing compiler tailors each procedure's register usage. The 32 registers are used like a stack, with a virtual stack frame marking the top of the stack for the new procedure activation environment. By making the compiler do all the work, a procedure call or return may require as few as two instructions. In addition, the compiler uses a priority-based graph coloring algorithm to optimize register usage within and across procedures.

The R2000 makes no use of condition codes. If an instruction generates a condition, the corresponding flags are stored in a general-purpose register. This avoids the need for special logic to deal with condition codes as they affect the pipelining mechanism and the reordering of instructions by the compiler. Instead, the mechanisms already implemented to deal with register-value dependencies are employed. Further, conditions mapped onto the register file are subject to the same compile-time optimizations in allocation and reuse as other values stored in registers.

As with the Berkeley RISC, but unlike many other RISC-based machines, the MIPS uses a single instruction length. This single instruction length simplifies instruction fetch and decode, and also simplifies the interaction of instruction fetch with the virtual memory management unit (i.e., instructions do not cross word or page boundaries). The three instruction formats (Fig. 17) share common formating

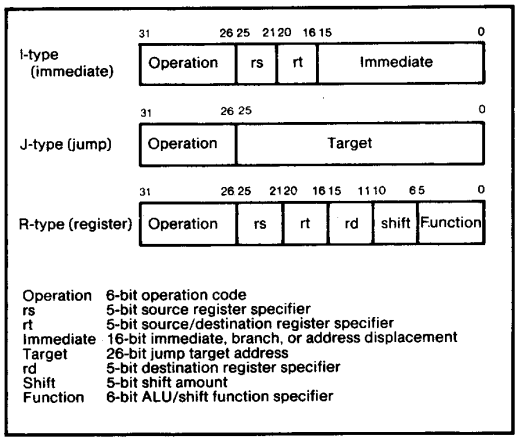


Fig. 17. MIPS instruction formats.

of opcodes and register references, simplifying instruction decode. The effect of longer instructions can be synthesized at compile time. For example, two I-type instructions can be concatenated to provide for operation on a 32-bit immediate quantity.

Only the simplest and most frequently used addressing mode is implemented in hardware. All addresses are of the form (contents of register plus immediate offset). Complex modes such as (base + index + offset) are synthesized in compile time, subject to optimizations that eliminate redundancy. This approach minimizes both hardware and pipeline latencies for loads and branches.

With its simplified instruction architecture, the MIPS can achieve very efficient pipelining. Instructions execute at a rate of almost one per cycle. The MIPS compiler is able to reorder instructions to fill delay slots with useful code 70–

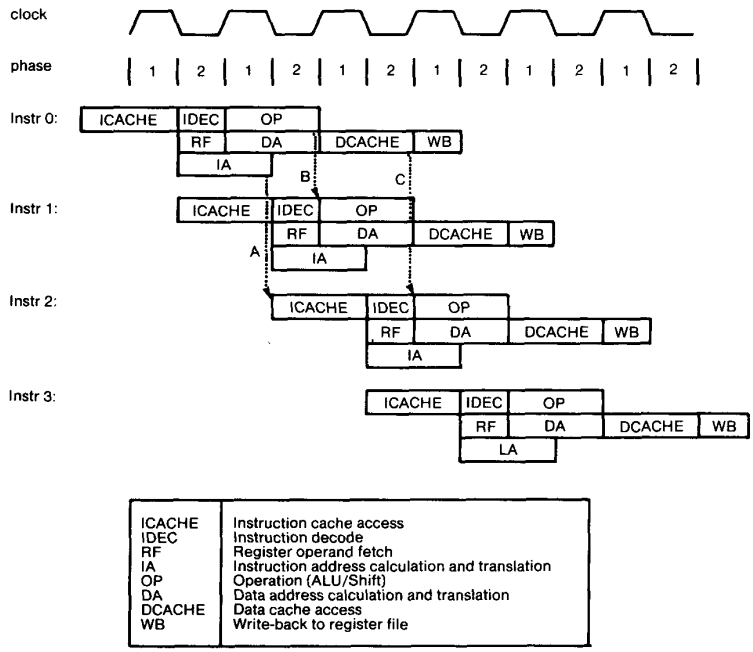


Fig. 18. MIPS instruction pipeline.

90 percent of the time. All instructions follow the same sequence of five pipeline stages: instruction fetch, source operand fetch from register file, ALU operation or data operand address generation, data memory reference, and write back into register file. As illustrated in Fig. 18, there is not only parallelism due to pipelining but also parallelism within the execution of a single instruction. The clock cycle is divided into two 30-ns phases. The external instruction and data access operations to the cache each require 60 ns, as do the major internal operations (OP, DA, IA). Instruction decode is a simpler operation, requiring only a single 30-ns phase, overlapped with register fetch in the same instruction. Calculation of an address for a branch instruction also overlaps instruction decode and register fetch, so that a branch at instruction 0 in Fig. 18 can address the ICACHE access of instruction 2 (see dotted line *A*). Similarly, a load at instruction 0 fetches data that are immediately used by the OP of instruction 2 (dotted line C), while an ALU/shift result gets passed directly into instruction 1 with no delay (dotted line *B*). This tight coupling between instructions makes for a highly efficient pipeline.

VII. The RISC versus CISC Controversy

For many years, the general trend in computer architecture and organization has been toward increasing CPU complexity: more instructions, more addressing modes, more specialized registers, and so on. The RISC movement represents a fundamental break with the philosophy behind that trend. Naturally the appearance of RISC systems, and the publication of papers by its proponents extolling RISC virtues, has led to a reaction from what might be called the mainstream of computer architecture.

The work that has been done on assessing merits of the RISC approach can be grouped into two categories:

Quantitative: attempts to compare program size and execution speed of programs on RISC and CISC machines that use comparable technology.

Qualitative: examination of issues such as high-level language support and optimum use of VLSI real estate.

Most of the work on quantitative assessment has been done by those working on RISC systems [31], [39], [40], and has been, by and large, favorable to the RISC approach. Others have examined the issue and come away unconvinced [41]. There are several problems with attempting such comparisons [42]:

- There is no pair of RISC and a CISC machine that are comparable in life-cycle cost, level of technology, gate complexity, sophistication of compiler, operating-system support, and so on.
- No definitive test set of programs exists. Performance varies with the program.
- It is difficult to sort out hardware effects from effects due to skill in compiler writing.
- Most of the comparative analysis on RISC has been done on "toy" machines: not commercial products. Furthermore, most commercially available machines advertised as RISC possess a mixture of RISC and CISC characteristics [13]. Thus a fair comparison with a commercial, "pure-play" CISC machine (e.g., VAX, Intel 432), is difficult.

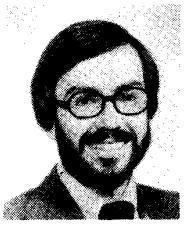
The qualitative assessment is, almost by definition, subjective. Several researchers have turned their attention to such an assessment [41], [43], [44], but the results are, at best, ambiguous, and certainly subject to rebuttal [45], and, of course, counter-rebuttal [46].

The success of the RISC approach in the marketplace is far from assured. As research, development, and product introduction continue, the assessment goes on.

References

[1] W. Stallings, *Reduced Instruction Set Computers*. Washington, DC: IEEE Computer Soc. Press, 1986.

[2] R. Weiss, "RISC processors: The new wave in computer systems," *Comput. Des.*, pp. 53-73, May 15, 1987.
[3] C. Bruno and S. Brady, "The RISC factor," *Datamation*, pp. y-DD, June 1, 1986.
[4] J. Moussouris *et al.*, "A CMOS RISC processor with integrated system functions," in *Proc. Compcon Spring 86* (Mar. 1986), pp. 126-131.
[5] L. Neff, "Clipper™ microprocessor architecture overview," in *Proc. Compcon Spring 86* (Mar. 1986), pp. 191-195.
[6] F. Waters, Ed., *IBM RT Personal Computer Technology*, IBM Publ. SA23-1057, 1986.
[7] E. Basart, "RISC design streamlines high-power CPUs," *Comput. Des.*, pp. 119-122, July 1, 1985.
[8] S. Gannes, "Back-to-basics computers with sports-car speed," *Fortune*, pp. 98-101, Sept. 30, 1985.
[9] M. Miller, "Simplicity is focus in efforts to increase computer power," *The Wall Street J.*, p. 17, Aug. 23, 1985.
[10] S. Ohr, "RISC machines," *Electron. Des.*, pp. 175-190, Jan. 10, 1985.
[11] M. Seither, "Pyramid challenges DEC with RISC supermini," *Mini-Micro Syst.*, pp. 33-36, Aug. 1985.
[12] R. Bernhard, "RISCs—Reduced instruction set computers—make leap," *Syst. Software*, pp. 81-84, Dec. 1984.
[13] N. Mokhoff, "New RISC machines appear as hybrids with both RISC and CISC features," *Comput. Des.*, pp. 22-25, Apr. 1, 1986.
[14] J. Birnbaum and W. Worley, "Beyond RISC: High-precision architecture," in *Proc. Compcon Spring 86* (Mar. 1986), pp. 40-47.
[15] J. Browne, "Understanding execution behavior of software systems," *Computer*, vol. 17, pp. 83-87, July 1984.
[16] D. Knuth, "An empirical study of FORTRAN programs," *Software Practice Exper.*, vol. 1, pp. 105-133, 1971.
[17] A. Tanenbaum, "Implications of structured programming for machine architecture," *Commun. ACM*, pp. 237-246, Mar. 1978.
[18] D. Patterson and C. Sequin, "A VLSI RISC," *Computer*, pp. 8-22, Sept. 1982.
[19] T. Huck, "Comparative analysis of computer architectures," Stanford Univ. Tech. Rep. 83-243, May 1983.
[20] A. Lunde, "Empirical evaluation of some features of instruction set processor architectures," *Commun. ACM*, pp. 143-153, Mar. 1972.
[21] M. Katevenis, "Reduced instruction set computer architectures for VLSI," Ph.D. dissertation, Computer Sci. Dep., Univ. of California at Berkeley, Oct. 1983. Reprinted by MIT Press, Cambridge, MA, 1985.
[22] D. Patterson, "Reduced instruction set computers," *Commun. ACM*, pp. 8-21, Jan. 1985.
[23] Y. Tamir and C. Sequin, "Strategies for managing the register file in RISC," *IEEE Trans. Comput.*, vol. C-30, pp. 977-988, Nov. 1983.
[24] R. Ragan-Kelley and R. Clark, "Applying RISC theory to a large computer," *Comput. Des.*, pp. 191-198, Nov. 1983.
[25] W. Stallings, *Computer Organization and Architecture*. New York, NY: Macmillan, 1987.
[26] G. Chaitin, "Register allocation and spilling via graph coloring," in *Proc. SIGPLAN Symp. on Compiler Construction* (June 1982), pp. 98-105.
[27] F. Chow, M. Himelstein, E. Killian, and L. Weber, "Engineering a RISC compiler system," in *Proc. Compcon Spring 86* (Mar. 1986), pp. 132-137.
[28] D. Coutant, C. Hammond, and J. Kelley, "Compilers for the new generation of Hewlett-Packard Computers," in *Proc. Compcon Spring 86* (Mar. 1986), pp. 182-195.
[29] J. Hennessy *et al.*, "Hardware/software tradeoffs for increased performance," in *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp. 2-11, Mar. 1982.
[30] G. Radin, "The 801 minicomputer," *IBM J. Res. Devel.*, pp. 237-246, May 1983.
[31] D. Patterson and R. Piepho, "Assessing RISCs in high-level language support," *IEEE Micro*, vol. 2, pp. 9-18, Nov. 1982.
[32] G. Myers, "The evaluation of expressions in a storage-to-storage architecture," *Comput. Arch. News*, June 1978.
[33] R. Sherburne, "Processor design tradeoffs in VLSI," Ph.D. dissertation, Rep. UCB/CSD 84/173, Univ. of California at Berkeley, Apr. 1984.
[34] D. Fitzpatrick *et al.*, "A RISCy approach to VLSI," *VLSI Des.*, pp. 14-20, 4th Quarter, 1981.
[35] J. Hennessy and T. Gross, "Postpass code optimization of pipeline constraints," *ACM Trans. Programming Languages Syst.*, July 1983.
[36] C. Rowen *et al.*, "RISC VLSI design for system-level performance," *VLSI Syst. Des.*, Mar. 1986.
[37] J. Moussouris *et al.*, "A CMOS RISC processor with integrated system functions," in *Proc. COMPCON Spring 86* (Mar. 1986), pp. 126-131.
[38] J. L. Hennessy, "VLSI processor architecture," *IEEE Trans. Comput.*, vol. C-33, pp. 1221-1246, Dec. 1984.
[39] J. Heath, "Re-evaluation of RISC I," *Comput. Arch. News*, Mar. 1984.
[40] D. Patterson, "RISC watch," *Comput. Arch. News*, Mar. 1984.
[41] R. P. Colwell, C. Y. Hitchcock, E. D. Jensen, H. M. B. Brinkley-Sprunt, and C. P. Kollar, "Instruction sets and beyond: Computers, complexity and controversy," *Computer*, vol. 18, pp. 8-19, Sept. 1985.
[42] O. Serlin, "MIPS, dhrystones, and other tales," *Datamation*, pp. 112-118, June 1, 1986.
[43] R. Bernhard, "More hardware means less software," *IEEE Spectrum*, December 1981.
[44] P. Wallich, "Toward simpler, faster computers." *IEEE Spectrum*, vol. 22, pp. 38-45, Aug. 1985.
[45] D. Patterson and J. Hennessy, "Comments, with reply on 'Computers, complexity, and controversy," by R. P. Colwell, *et al.*, *Computer*, vol. 18, pp. 142-143, Nov. 1985.
[46] R. Colwell, C. Hitchcock, E. Jensen, and H. Sprunt, "More controversy about 'computers, complexity, and controversy,'" *Computer*, p. 93, Dec. 1985.



William Stallings (Senior Member, IEEE) received the B.S. degree in electrical engineering from Notre Dame University, Notre Dame, IN, and the Ph.D. degree in computer science from the Massachusetts Institute of Technology, Cambridge, MA.

He is an independent consultant and president of Comp/Comm Consulting, London, England. He is also a frequent lecturer and the author of numerous technical papers and eleven books in the fields of data communications and computer science, including *Computer Organization and Architecture* (New York, NY: Macmillan, 1987) and *Data and Computer Communications, Second Edition* (New York, NY: Macmillan, 1988). His clients have included the Government of India, the International Monetary Fund, the National Security Agency, IBM, and Honeywell. Prior to forming his own consulting firm, he was Vice President of CSM Corp., a firm specializing in data processing and data communications for the health-care industry. He was also Director of Systems analysis and design for CTEC, Inc., a firm specializing in command, control, and communications systems.