Chapter 17 Reduced Instruction Set Computers

Learning Objectives

After studying this chapter, you should be able to:

- Provide an overview of the research results on instruction execution characteristics that motivated

the development of the RISC approach.
- Summarize the key characteristics of RISC machines.
- Understand the design and performance implications of using a large register file.
- Understand the use of compiler-based register optimization to improve performance.
- Discuss the implication of a RISC architecture for pipeline design and performance.
- List and explain key approaches to pipeline optimization on a RISC machine.

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. The following are some of the major advances since the birth of the computer:

- ***The family concept:*** *Introduced by IBM with its System/360 in 1964, followed shortly thereafter by DEC, with its PDP-8. The family concept decouples the architecture of a machine from its implementation. A set of computers is offered, with different price/performance characteristics, that presents the same architecture to the user. The differences in price and performance are due to different implementations of the same architecture.*
- ***Microprogrammed control unit:*** *Suggested by Wilkes in 1951 and introduced by IBM on the S/360 line in 1964. Microprogramming eases the task of designing and implementing the control unit and provides support for the family concept.*
- ***Cache memory:*** *First introduced commercially on IBM S/360 Model 85 in 1968. The insertion of this element into the memory hierarchy dramatically improves performance.*
- ***Pipelining:*** *A means of introducing parallelism into the essentially sequential nature of a machine-instruction program. Examples are instruction pipelining and vector processing.*
- ***Multiple processors:*** *This category covers a number of different organizations and objectives.*
- ***Reduced instruction set computer (RISC) architecture:*** *This is the focus of this chapter.*

When it appeared, RISC architecture was a dramatic departure from the historical trend in processor architecture. An analysis of the RISC architecture brings into focus many of the important issues in computer organization and architecture.

Although RISC architectures have been defined and designed in a variety of ways by different groups, the key elements shared by most designs are these:

- *A large number of general-purpose registers, and/or the use of compiler technology to optimize register usage.*
- *A limited and simple instruction set.*
- *An emphasis on optimizing the instruction pipeline.*

Table 17.1 *compares several RISC and non-RISC systems.*

Table 17.1 Characteristics of Some CISCs, RISCs, and Superscalar Processors

	Complex Instruction Set (CISC)Computer	Reduced Instruction Set (RISC) Computer

Characteristic	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000
Year developed	1973	1978	1989	1987	1991
Number of instructions	208	303	235	69	94
Instruction size (bytes)	2–6	2–57	1–11	4	4
Addressing modes	4	22	11	1	1
Number of general-purpose registers	16	16	8	40–520	32
Control memory size (kbits)	420	480	246	—	—
Cache size (kB)	64	64	8	32	128

	Superscalar		
Characteristic	PowerPC	Ultra SPARC	MIPS R10000
Year developed	1993	1996	1996
Number of instructions	225		
Instruction size (bytes)	4	4	4
Addressing modes	2	1	1

Number of general-purpose registers	*32*	*40–520*	*32*
Control memory size (kbits)	—	—	—
Cache size (kB)	*16–32*	*32*	*64*

We begin this chapter with a brief survey of some results on instruction sets, and then examine each of the three topics just listed. This is followed by a description of two of the best-documented RISC designs.

17.1 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. These **high-level languages (HLLs)**: (1) allow the programmer to express algorithms more concisely; (2) allow the compiler to take care of details that are not important in the programmer's expression of algorithms; and (3) often support naturally the use of structured programming and/or object-oriented design.

Alas, this solution gave rise to a perceived 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 machine program size, and compiler complexity. Designers responded with architectures intended to close this gap. Key features include large 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. Such complex instruction sets are intended to:

- Ease the task of the compiler writer.
- Improve execution efficiency, because 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 a different approach: namely, to make the architecture that supports the HLL simpler, rather than more complex.

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 as follows:

- **Operations performed:** These determine the functions to be performed by the processor 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. 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.

Operations

A variety of studies have been made to analyze the behavior of HLL programs, with the following

general conclusions. 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, we want to answer the question: Given a compiled machine-language program, which statements in the source language cause the execution of the most machine-language instructions and what is the execution time of these instructions?

To get at this underlying phenomenon, Patterson and Sequin [PATT82a] analyzed a set of measurements taken from compilers and programs for typesetting, computer-aided design (CAD), sorting, and file comparison. The programming languages C and Pascal compiled on the VAX, PDP-11, and Motorola 68000 to determine the average number of machine instructions and memory references per statement type. The second and third columns in **Table 17.2** show the relative frequency of occurrence of various HLL statements in a variety of programs; the data were obtained by observing the occurrences in running programs rather than just the number of times that statements occur in the source code. Hence these metrics capture dynamic behavior. To obtain the data in columns four and five (machine-instruction weighted), each value in the second and third columns is multiplied by the number of machine instructions produced by the compiler. These results are then normalized so that columns four and five show the relative frequency of occurrence, weighted by the number of machine instructions per HLL statement. Similarly, the sixth and seventh columns are obtained by multiplying the frequency of occurrence of each statement type by the relative number of memory references caused by each statement. The data in columns four through seven 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 17.2 Weighted Relative Dynamic Frequency of HLL Operations [PATT82a]

	Dynamic Occurrence		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%

The reader should be clear on the significance of **Table 17.2**. This table indicates the relative

performance impact 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 computer (CISC)** architectures. Thus, they can provide guidance to those looking for more efficient ways to support HLLs.

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 [PATT82a] also looked at the dynamic frequency of occurrence of classes of variables (**Table 17.3**). The results, consistent between Pascal and C programs, show that most references are to simple scalar variables. Further, more than 80% of the scalars were local (to the procedure) variables. In addition, each reference to an array or a structure requires a reference to an 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 17.3 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 before, it is necessary to deal with actual architectures to examine program behavior more deeply. One study, [LUND77], examined DEC-10 instructions dynamically and found that each instruction on the average references 0.5 operand in memory and 1.4 registers. Similar results are reported in [HUCK83] for C, Pascal, and FORTRAN programs on S/370, PDP-11, and VAX. 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, because 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.

Procedure Calls

We have seen that procedure calls and returns are an important aspect of HLL programs. The evidence (**Table 17.2**) suggests that these are the most time-consuming operations in 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.

A study by Tanenbaum [TANE78] found that 98% of dynamically called procedures were passed fewer than six arguments and that 92% of them used fewer than six local scalar variables. Similar

results were reported by the Berkeley RISC team [KATE83], as shown in **Table 17.4**. 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 is to local scalar variables. These studies show that those references are in fact confined to relatively few variables.

Table 17.4 Procedure Arguments and Local Scalar Variables

Percentage of Executed Procedure Calls With	Compiler, Interpreter, and Typesetter	Small Nonnumeric Programs
>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. These results reinforce the conclusion that operand references are highly localized.

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, characterize RISC architectures. First, use a large number of registers or use a compiler to optimize register usage. This is intended to optimize operand referencing. The studies just discussed show that there are several references per HLL statement 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 prefetched but never executed.

Finally, an instruction set consisting of high-performance primitives is indicated. Instructions should

have predictable costs (measured in execution time, code size, and increasingly, in energy dissipation) and be consistent with a high-performance implementation (which harmonizes with predictable execution-time cost).

17.2 The Use of a Large Register File

The results summarized in **Section 17.1** point out the desirability of quick access to operands. We have seen that there is a large proportion of assignment statements in HLL programs, and many of these are of the simple form $A \leftarrow B$. 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, 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 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 assign 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.

In this section, we will discuss the hardware approach. This approach has been pioneered by the Berkeley RISC group [PATT82a]; was used in the first commercial RISC product, the Pyramid [RAGA83]; and is currently used in the popular **SPARC** architecture.

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.

Because 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 procedure. Furthermore, parameters must be passed. On return, the variables of the calling procedure must be restored (loaded back into registers) and results must be passed back to the calling procedure.

The solution is based on two other results reported in **Section 17.1**. First, a typical procedure employs only a few passed parameters and local variables (**Table 17.4**). Second, the depth of procedure activation fluctuates within a relatively narrow range. To exploit these properties, multiple small sets of registers are used, each assigned to a different procedure. A procedure call automatically switches the processor 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 **Figure 17.1**. At any time, only one window of registers is visible and is addressable as if it were the only set of registers (e.g., addresses 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 hold results to be passed back up. Local registers are used for local variables, as assigned by the compiler. Temporary registers are used to 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. Keep in mind that, except for the overlap, the registers at two different levels are physically distinct. That is, the parameter and local registers at level *J* are disjoint from the local and temporary registers at level $J + 1$.

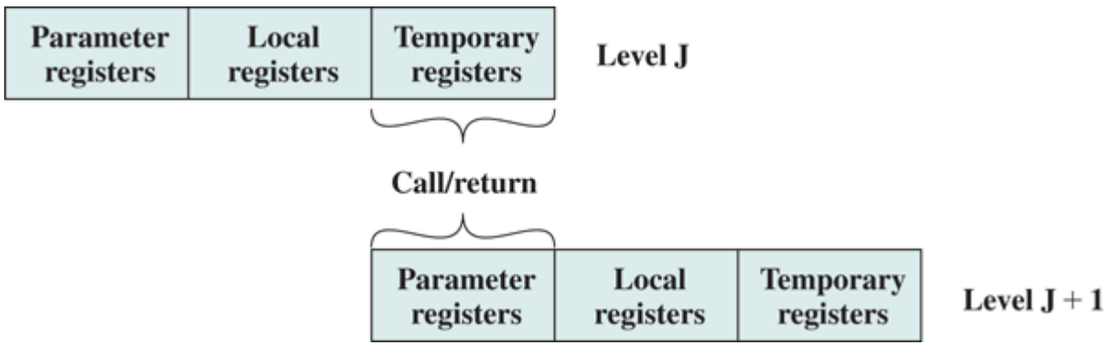


Figure 17.1 Overlapping Register Windows

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. Two notable examples of this approach are Sun's SPARC architecture, described in **Section 17.7**, and the IA-64 architecture used in Intel's Itanium processor.

The circular organization is shown in **Figure 17.2**, which depicts a circular buffer of six windows. The buffer is filled to a depth 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 (SWP) 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 w3 and w4) and the CWP is advanced by one window.

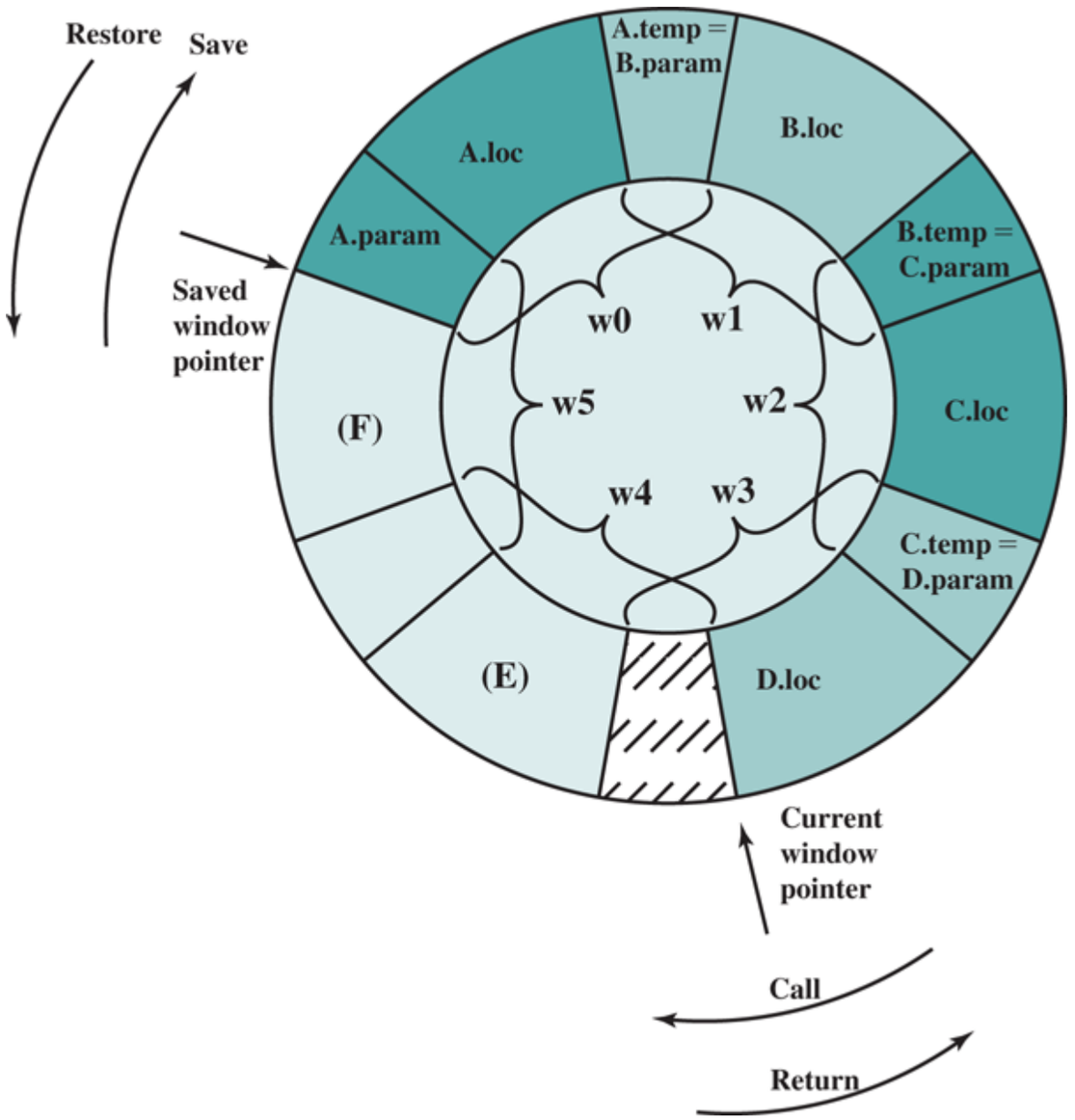


Figure 17.2 Circular-Buffer Organization of Overlapped Windows

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.in). 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.in and A.loc) need be saved. Then, the SWP is incremented and the call to F proceeds. A similar interrupt 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 restoration 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. One study [TAMI83] found that, with 8 windows, a save or restore is needed on only 1% 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. 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 processor. 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. There is an increased hardware burden to accommodate the split in register addressing. In addition, the linker 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 the most heavily. From this point of view, the register file acts much like a cache memory, although a much faster 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 17.5 compares characteristics of the two approaches. The window-based register file holds all 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 file should save time, because all local scalar variables are retained. On the other hand, the cache may make more efficient use of space, because it is reacting to the situation dynamically. Furthermore, caches generally treat all memory references alike, including instructions and other types of data. Thus, savings in these other areas are possible with a cache and not a register file.

Table 17.5 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
Multiple operands addressed and accessed in one cycle	One operand addressed and accessed per cycle

A register file may make inefficient use of space, because not all procedures will need the full window

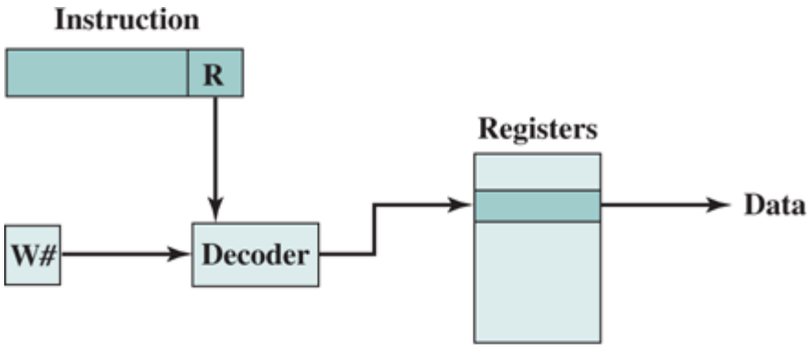
space allotted to them. On the other hand, the cache suffers from another sort of inefficiency: Data are read into the cache in blocks. Whereas the register file contains only those variables in use, the cache reads in a block of data, some or much of which will not be used.

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 [KATE83]. 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, when program modules are separately compiled, it is impossible for the compiler to assign global values to registers; the linker must perform this task.

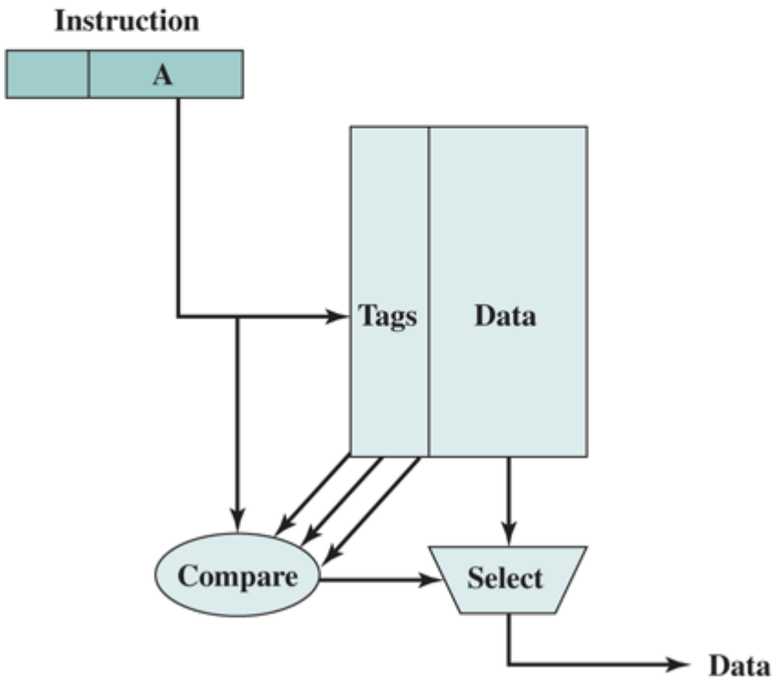
With the register file, the movement of data between registers and memory is determined by the procedure nesting depth. Because this depth usually fluctuates within a narrow range, the use of memory is relatively infrequent. Most cache memories are set associative with a small set size. Thus, there is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

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.

Figure 17.3 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, 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 with the tags, and one of the words that were 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.



(a) Window-based register file



(b) Cache

Figure 17.3 Referencing a Scalar

17.3 Compiler-Based Register Optimization

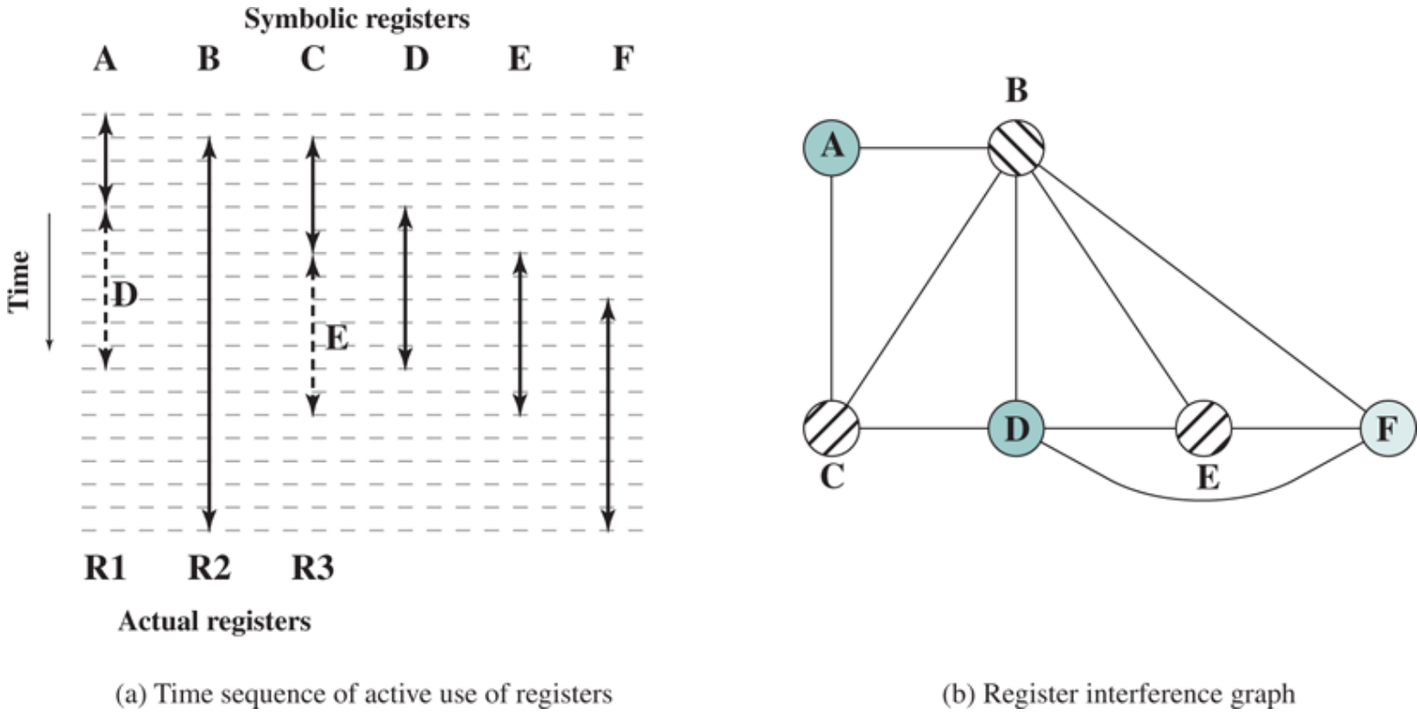
Let us assume now that only a small number (e.g., 16–32) of registers is available on the target RISC machine. In this case, optimized register usage is the responsibility of the compiler. A program written in a high-level language has, of course, no explicit references to registers (the C-language keyword register notwithstanding). Rather, program quantities are referred to symbolically. The objective of the compiler is to keep the operands for 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 position quantities in registers temporarily 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 [CHAI82, CHOW86, COUT86, CHOW90].

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. Nodes that share the same color can be assigned to the same register. If this process does not fully succeed, then those 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.

Figure 17.4 is a simple example of the process. Assume a program with six symbolic registers to be compiled into three actual registers. **Figure 17.4a** shows the time sequence of active use of each symbolic register. The dashed horizontal lines indicate successive instruction executions. **Figure 17.4b** shows the register interference graph (shading and stripes are used instead of colors). A possible coloring with three colors is indicated. Because symbolic registers A and D do not interfere, the compiler can assign both of these to physical register R1. Similarly, symbolic registers C and E can be assigned to register R3. One symbolic register, F, is left uncolored and must be dealt with using loads and stores.



(a) Time sequence of active use of registers

(b) Register interference graph

Figure 17.4 Graph Coloring Approach

In general, there is a trade-off between the use of a large set of registers and compiler-based register optimization. For example, [BRAD91a] reports on a study that modeled a RISC architecture with features similar to the Motorola 88000 and the MIPS R2000. The researchers varied the number of registers from 16 to 128, and they considered both the use of all general-purpose registers and registers split between integer and floating-point use. Their study showed that with even simple register optimization, there is little benefit to the use of more than 64 registers. With reasonably sophisticated register optimization techniques, there is only marginal performance improvement with more than 32 registers. Finally, they noted that with a small number of registers (e.g., 16), a machine with a shared register organization executes faster than one with a split organization. Similar conclusions can be drawn from [HUGU91], which reports on a study that is primarily concerned with optimizing the use of a small number of registers rather than comparing the use of large register sets with optimization efforts.

17.4 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 chapter. We begin with a discussion of motivations for contemporary complex instruction set architectures.

Why CISC

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 HLLs on the part of programmers; architects attempted to design machines that provided better support for HLLs.

It is not the intent of this chapter to say that the CISC designers took the wrong direction. 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, but it is not. The task of the compiler writer is to build a compiler that generates good (fast, small, fast and small) sequences of machine instructions for HLL programs (i.e., the compiler views individual HLL statements in the context of surrounding HLL statements). If there are machine instructions that resemble HLL statements, this task is simplified. This reasoning has been disputed by the RISC researchers ([HENN82], [RADI83], [PATT82b]). They have found that complex machine instructions are often hard to exploit because 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 chapter 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. 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 three ways. First, fewer instructions means fewer instruction bytes to be fetched. Second, in a paging environment, smaller programs occupy fewer pages, reducing page faults. Third, more instructions fit in cache(s).

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 17.6** shows 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. Note that there is little or no savings using a CISC over a RISC. It is also interesting to note that the VAX, 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 [RADI83], 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.

Table 17.6 Code Size Relative to RISC I

	[PATT82a] 11 C Programs	[KATE83] 12 C Programs	[HEAT84] 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	

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, because 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.

So the expectation that a CISC will produce smaller programs, with the attendant advantages, may not be realized. 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 toward 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 [RADI83]. In effect, the control store acts as an instruction cache. Thus, the hardware architect 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. On S/390 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 a trend to increasingly complex instruction sets is appropriate. This has led a number of groups to pursue the opposite path.

Characteristics of Reduced Instruction Set Architectures

Although a variety of different approaches to reduced instruction set architecture have been taken, certain characteristics are common to all of them:

- One instruction per cycle
- Register-to-register operations
- Simple addressing modes
- Simple instruction formats

Here, we provide a brief discussion of these characteristics. Specific examples are explored later in this chapter.

The first characteristic listed is that there is **one machine instruction per machine cycle**. A *machine cycle* is defined to be the time it takes to fetch two operands from registers, perform an 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 (discussed in Part Four). 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, because 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 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 notable for RISC designs. Contemporary CISC 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. **Figure 17.5a** illustrates the approach taken. 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 [MYER78]. One wonders what he would have thought, at the time, of the RISC machine once produced by Pyramid, which contained no less than 528 registers!

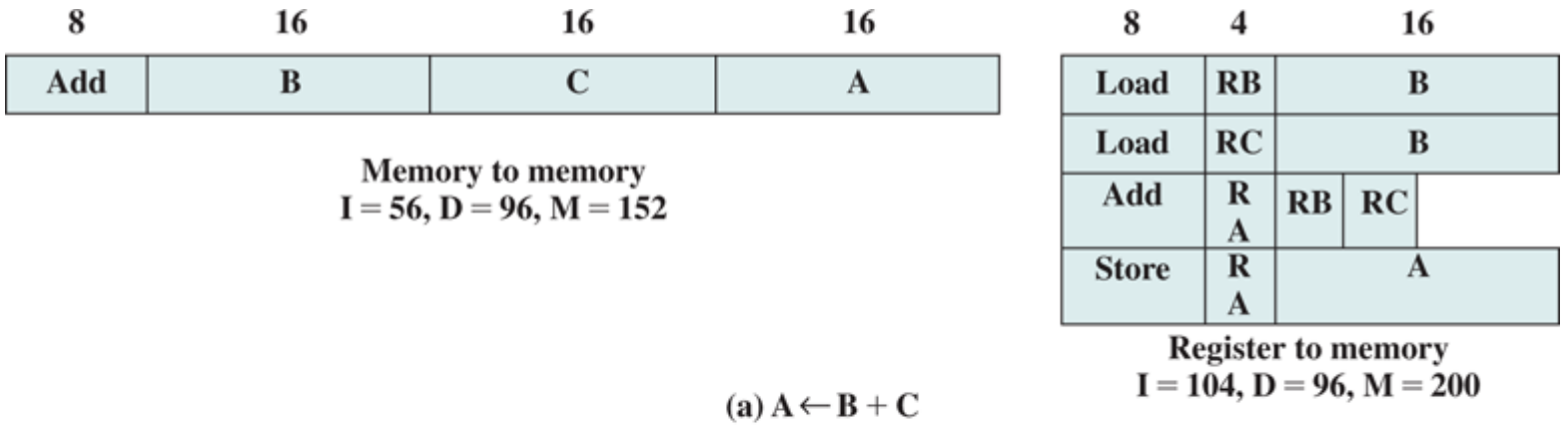


Figure 17.5 Two Comparisons of Register-to-Register and Memory-to-Memory Approaches

What was missing from those 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, **Figure 17.5b** may be a fairer comparison.

A third characteristic is the use of **simple addressing modes**. Almost all RISC instructions use simple 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 because word-length units are fetched. Alignment on a word boundary also means that a single instruction does not cross page boundaries.

Taken together, these characteristics can be assessed to determine the potential performance benefits of the RISC approach. 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 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/390

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 processors are more responsive to interrupts because interrupts are checked between rather elementary operations. Architectures with complex instructions either restrict interrupts to instruction boundaries or must define specific interruptible points and implement mechanisms for restarting an instruction.

The case for improved performance for a reduced instruction set architecture is strong, but one could perhaps still make an argument for CISC. 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, is suggestive.

CISC versus RISC Characteristics

After the initial enthusiasm for RISC machines, there has been a growing realization that (1) RISC designs may benefit from the inclusion of some CISC features and that (2) CISC designs may benefit from the inclusion of some RISC features. The result is that the more recent RISC designs, notably the PowerPC, are no longer “pure” RISC and the more recent CISC designs, notably the Pentium II and later Pentium models, do incorporate some RISC characteristics.

An interesting comparison in [MASH95] provides some insight into this issue. **Table 17.7** lists a number of processors and compares them across a number of characteristics. For purposes of this comparison, the following are considered typical of a classic RISC:

1. A single instruction size.
2. That size is typically 4 bytes.
3. A small number of data addressing modes, typically less than five. This parameter is difficult to pin down. In the table, register and literal modes are not counted and different formats with different offset sizes are counted separately.
4. No indirect addressing that requires you to make one memory access to get the address of another operand in memory.
5. No operations that combine load/store with arithmetic (e.g., add from memory, add to memory).
6. No more than one memory-addressed operand per instruction.
7. Does not support arbitrary alignment of data for load/store operations.
8. Maximum number of uses of the memory management unit (MMU) for a data address in an instruction.
9. Number of bits for integer register specifier equal to five or more. This means that at least 32 integer registers can be explicitly referenced at a time.
10. Number of bits for floating-point register specifier equal to four or more. This means that at least 16 floating-point registers can be explicitly referenced at a time.

Table 17.7 Characteristics of Some Processors

Notes:

[a] RISC that does not conform to this characteristic.

[b] CISC that does not conform to this characteristic.

Processor	Number of	Max instruction	Number of addressing	Indirect addressing	Load/store combined	Max number	Unaligned addressing	Max number	Number of bits	Number of bits

	instruction sizes	size in bytes	modes		with arithmetic	of memory operands	allowed	of MMU uses	for integer register specifier	for FP register specifier
AMD29000	1	4	1	no	no	1	no	1	8	3[a]
MIPS R2000	1	4	1	no	no	1	no	1	5	4
SPARC	1	4	2	no	no	1	no	1	5	4
MC88000	1	4	3	no	no	1	no	1	5	4
HP PA	1	4	10[a]	no	no	1	no	1	5	4
IBM RT/PC	2[a]	4	1	no	no	1	no	1	4[a]	3[a]
IBM RS/6000	1	4	4	no	no	1	yes	1	5	5
Intel i860	1	4	4	no	no	1	no	1	5	4
IBM 3090	4	8	2[b]	no[b]	yes	2	yes	4	4	2
Intel 80486	12	12	15	no[b]	yes	2	yes	4	3	3
NSC 32016	21	21	23	yes	yes	2	yes	4	3	3
MC68040	11	22	44	yes	yes	2	yes	8	4	3
VAX	56	56	22	yes	yes	6	yes	24	4	0
Clipper	4[a]	8[a]	9[a]	no	no	1	0	2	4[a]	3[a]
Intel 80960	2[a]	8[a]	9[a]	no	no	1	yes[a]	—	5	3[a]

Items 1 through 3 are an indication of instruction decode complexity. Items 4 through 8 suggest the ease or difficulty of pipelining, especially in the presence of virtual memory requirements. Items 9 and 10 are related to the ability to take good advantage of compilers.

In the table, the first eight processors are clearly RISC architectures, the next five are clearly CISC, and the last two are processors often thought of as RISC that in fact have many CISC characteristics.

17.5 Risc Pipelining

Pipelining with Regular Instructions

As we discussed in **Section 16.4**, instruction pipelining is often used to enhance performance. Let us reconsider this in the context of a RISC architecture. Most instructions are register to register, and an instruction cycle has the following two stages:

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

For load and store operations, three stages are required:

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

Figure 17.6a depicts the timing of a sequence of instructions using no pipelining. Clearly, this is a wasteful process. Even very simple pipelining can substantially improve performance. **Figure 17.6b** shows a two-stage pipelining scheme, in which the I and E stages of two different instructions are performed simultaneously. The two stages of the pipeline are an instruction fetch stage, and an execute/memory stage that executes the instruction, including register-to-memory and memory-to-register operations. Thus we see that the instruction fetch stage of the second instruction can be performed in parallel with the first part of the execute/memory stage. However, the execute/memory stage of the second instruction must be delayed until the first instruction clears the second stage of the pipeline. This 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 stage. 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.

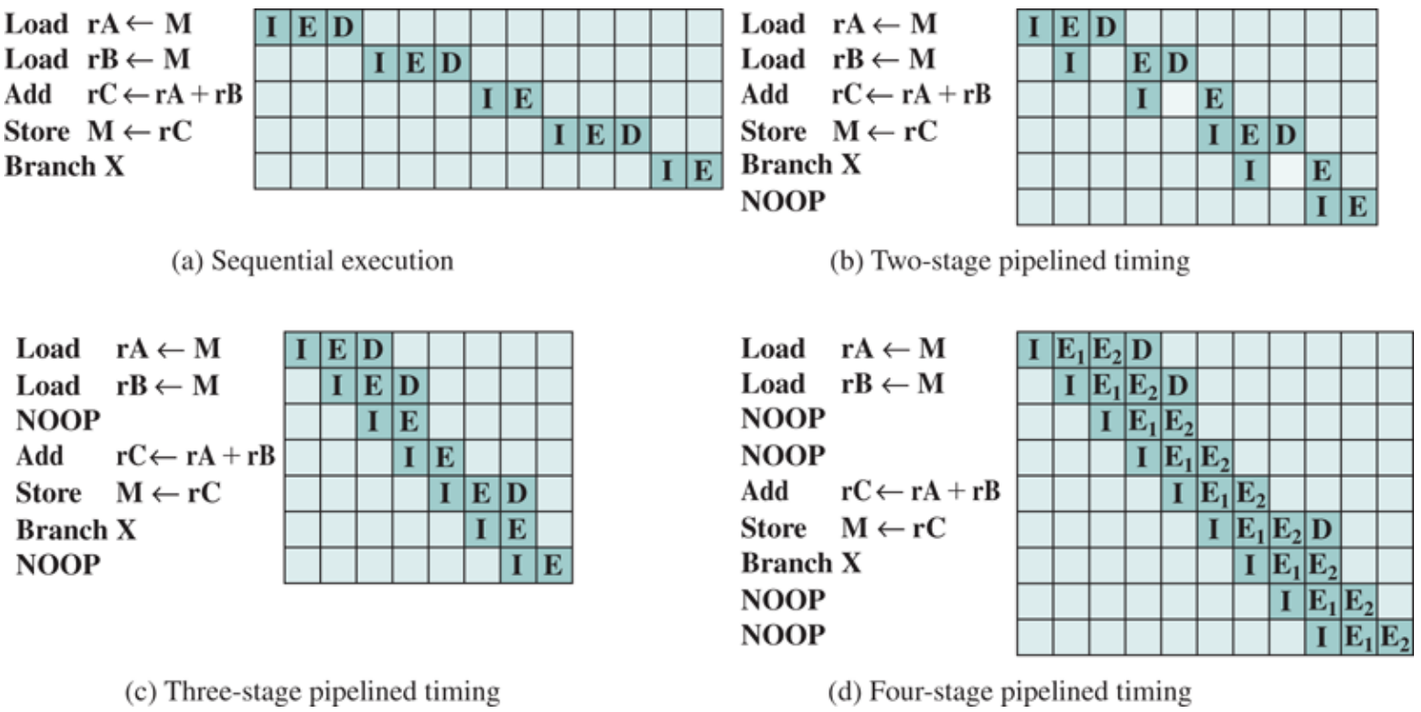


Figure 17.6 The Effects of Pipelining

Pipelining can be improved further by permitting two memory accesses per stage. This yields the sequence shown in **Figure 17.6c**. Now, up to three instructions can be overlapped, and the improvement is as much as a factor of 3. 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 stages are of approximately equal duration. Because the E stage usually involves an ALU operation, it may be longer. In this case, we can divide into two substages:

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

Because of the simplicity and regularity of a RISC instruction set, the design of the phasing into three or four stages is easily accomplished. **Figure 17.6d** shows the result with a four-stage pipeline. Up to four instructions at a time can be under way, and the maximum potential speedup is a factor of 4. Note again the use of NOOPs to account for data and branch delays.

Optimization of Pipelining

Because of the simple and regular nature of RISC instructions, it is easier for a hardware designer to implement a simple, fast pipeline. 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.

DELAYED BRANCH

To compensate for these dependencies, code reorganization techniques have been developed. 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 execution of the following instruction (hence the term *delayed*). The instruction location immediately following the branch is referred to as the *delay slot.* This strange procedure is illustrated in **Table 17.8**. In the column labeled “normal branch,” we see a normal symbolic instruction machine-language program. After 102 is executed, the next instruction to be executed is 105. 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.

Table 17.8 Normal and Delayed Branch

Address	Normal Branch		Delayed Branch		Optimized Delayed Branch	
100	LOAD	X, rA	LOAD	X, rA	LOAD	X, rA
101	ADD	1, rA	ADD	1, rA	JUMP	105
102	JUMP	105	JUMP	106	ADD	1, rA
103	ADD	rA, rB	NOOP		ADD	rA, rB

104	SUB	rC, rB	ADD	rA, rB	SUB	rC, rB
105	STORE	rA, Z	SUB	rC, rB	STORE	rA, Z
106			STORE	rA, Z		

Figure 17.7 shows the result. **Figure 17.7a** shows the traditional approach to pipelining, of the type discussed in **Chapter 16** (e.g., see **Figures 16.11** and **16.12**). The JUMP instruction is fetched at time 4. At time 5, the JUMP instruction is executed at the same time that instruction 103 (ADD instruction) is fetched. Because a JUMP occurs, which updates the program counter, the pipeline must be cleared of instruction 103; at time 6, instruction 105, which is the target of the JUMP, is loaded. **Figure 17.7b** shows the same pipeline handled by a typical RISC organization. The timing is the same. However, because of the insertion of the NOOP instruction, we do not need special circuitry to clear the pipeline; the NOOP simply executes with no effect. **Figure 17.7c** shows the use of the delayed branch. The JUMP instruction is fetched at time 2, before the ADD instruction, which is fetched at time 3. Note, however, that the ADD instruction is fetched before the execution of the JUMP instruction has a chance to alter the program counter. Therefore, during time 4, the ADD instruction is executed at the same time that instruction 105 is fetched. Thus, the original semantics of the program are retained but two fewer clock cycles are required for execution.

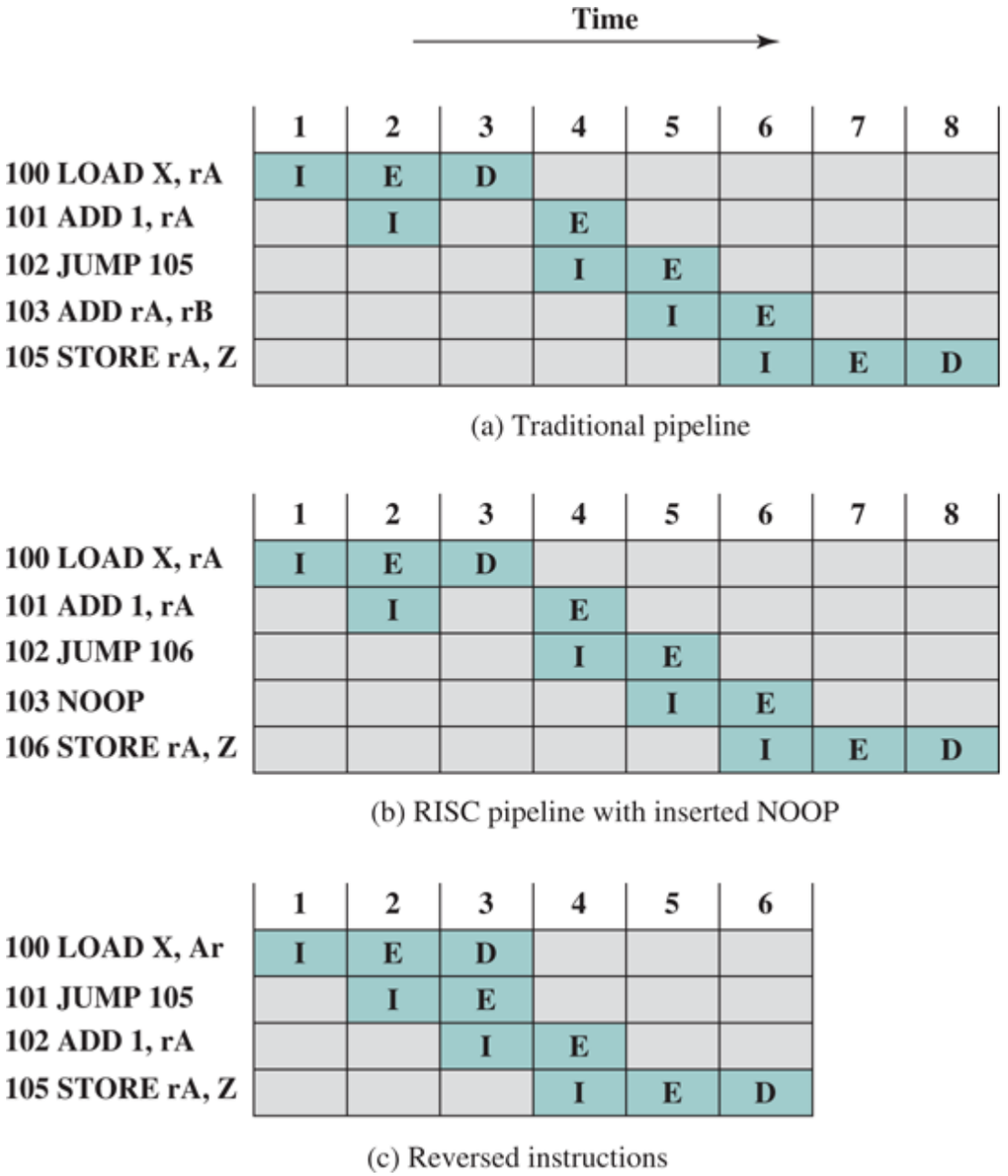


Figure 17.7 Use of the Delayed Branch

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. Otherwise, the compiler can seek to insert a useful instruction after the branch. The experience with both the Berkeley RISC and IBM 801 systems is that the majority of conditional branch instructions can be optimized in this fashion ([PATT82a], [RADI83]).

DELAYED LOAD

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 processor. The processor 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.

Aleksandr Lukin/123RF

Loop Unrolling Simulator

LOOP UNROLLING

Another compiler technique to improve instruction parallelism is loop unrolling [BACO94]. Unrolling replicates the body of a loop some number of times called the unrolling factor (*u*) and iterates by step *u* instead of step 1.

Unrolling can improve the performance by

- reducing loop overhead
- increasing instruction parallelism by improving pipeline performance
- improving register, data cache, or TLB locality

Figure 17.8 illustrates all three of these improvements in an example. Loop overhead is cut in half because two iterations are performed before the test and branch at the end of the loop. Instruction parallelism is increased because the second assignment can be performed while the results of the first are being stored and the loop variables are being updated. If array elements are assigned to registers, register locality will improve because a[*i*] and $a[i+1]$ are used twice in the loop body, reducing the number of loads per iteration from three to two.

```
do i=2, n-1
        a[i] = a[i] + a[i-1] * a[i+1]
end do
```

(a) Original loop

```
do i=2, n-2, 2
        a[i] = a[i] + a[i-1] * a[i+1]
        a[i+1] = a[i+1] + a[i] * a[i+2]
end do

if (mod(n-2, 2) = i) then
    a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

(b) Loop unrolled twice

Figure 17.8 Loop Unrolling

As a final note, we should point out that the design of the instruction pipeline should not be carried out in isolation from other optimization techniques applied to the system. For example, [BRAD91b] shows that the scheduling of instructions for the pipeline and the dynamic allocation of registers should be considered together to achieve the greatest efficiency.

17.6 MIPS R4000

One of the first commercially available RISC chip sets was developed by MIPS Technology Inc. The system was inspired by an experimental system, also using the name MIPS, developed at Stanford [HENN84]. In this section we look at the MIPS R4000. It has substantially the same architecture and instruction set of the earlier MIPS designs: the R2000 and R3000. The most significant difference is that the R4000 uses 64 rather than 32 bits for all internal and external data paths and for addresses, registers, and the ALU.

The use of 64 bits has a number of advantages over a 32-bit architecture. It allows a bigger address space—large enough for an operating system to map more than a terabyte of files directly into virtual memory for easy access. With 1-terabyte and larger disk drives now common, the 4-gigabyte address space of a 32-bit machine becomes limiting. Also, the 64-bit capacity allows the R4000 to process data such as IEEE double-precision floating-point numbers and character strings, up to eight characters in a single action.

The R4000 processor chip is partitioned into two sections, one containing the CPU and the other containing a coprocessor for memory management. The processor 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).

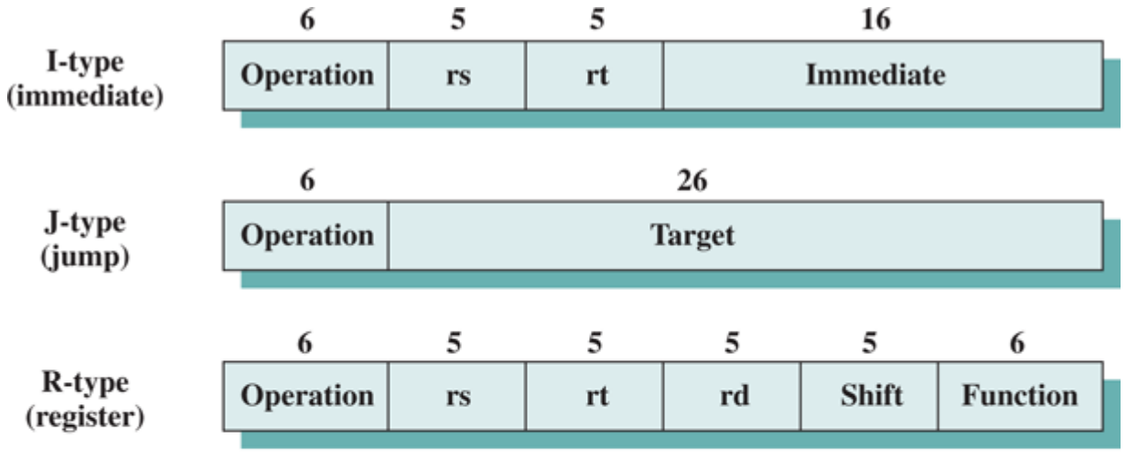
The processor supports thirty-two 64-bit registers. It also provides for up to 128 Kbytes of high-speed cache, half each for instructions and data. The relatively large cache (the IBM 3090 provides 128 to 256 Kbytes of cache) enables the system to keep large sets of program code and data local to the processor, off-loading the main memory bus and avoiding the need for a large register file with the accompanying windowing logic.

Instruction Set

All MIPS R series instructions are encoded in a single 32-bit word format. All data operations are register to register; the only memory references are pure load/store operations.

The R4000 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 files are subject to the same compile-time optimizations in allocation and reuse as other values stored in registers.

As with most RISC-based machines, the MIPS uses a single 32-bit instruction length. This single instruction length simplifies instruction fetch and decode, and it 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 (**Figure 17.9**) share common formatting of opcodes and register references, simplifying instruction decode. The effect of more complex instructions can be synthesized at compile time.



Operation	Operation code
rs	Source register specifier
rt	Source/destination register specifier
Immediate	Immediate, branch, or address displacement
Target	Jump target address
rd	Destination register specifier
Shift	Shift amount
Function	ALU/shift function specifier

Figure 17.9 MIPS Instruction Formats

Only the simplest and most frequently used memory-addressing mode is implemented in hardware. All memory references consist of a 16-bit offset from a 32-bit register. For example, the “load word” instruction is of the form

`lw r2, 128(r3)`	/* load word at address 128 offset from register 3 into register 2

Each of the 32 general-purpose registers can be used as the base register. One register, r0, always contains 0.

The compiler makes use of multiple machine instructions to synthesize typical addressing modes in conventional machines. Here is an example from [CHOW87], which uses the instruction lui (load upper immediate). This instruction loads the upper half of a register with a 16-bit immediate value, setting the lower half to zero. Consider an assembly-language instruction that uses a 32-bit immediate argument

`lw r2, #imm(r4)`	/* load word at address using a 32-bit immediate offset #imm /* offset from register 4 into register 2

This instruction can be compiled into the following MIPS instructions

`lui r1, #imm-hi`	/* where #imm-hi is the high-order 16 bits of #imm
`addu r1, r1, r4`	/* add unsigned #imm-hi to r4 and put in r1
`lw r2, #imm-lo(r1)`	/* where #imm-lo is the low-order 16 bits of #imm

Instruction Pipeline

With its simplified instruction architecture, the MIPS can achieve very efficient pipelining. It is instructive to look at the evolution of the MIPS pipeline, as it illustrates the evolution of RISC pipelining in general.

The initial experimental RISC systems and the first generation of commercial RISC processors achieve execution speeds that approach one instruction per system clock cycle. To improve on this performance, two classes of processors have evolved to offer execution of multiple instructions per clock cycle: superscalar and superpipelined architectures. In essence, a superscalar architecture replicates each of the pipeline stages so that two or more instructions at the same stage of the pipeline can be processed simultaneously. A superpipelined architecture is one that makes use of more, and more fine-grained, pipeline stages. With more stages, more instructions can be in the pipeline at the same time, increasing parallelism.

Both approaches have limitations. With superscalar pipelining, dependencies between instructions in different pipelines can slow down the system. Also, over-head logic is required to coordinate these dependencies. With superpipelining, there is overhead associated with transferring instructions from one stage to the next.

Chapter 18 is devoted to a study of superscalar architecture. The MIPS R4000 is a good example of a RISC-based superpipeline architecture.

Aleksandr Lukin/123RF

MIPS R3000 Five-Stage Pipeline Simulator

Figure 17.10a shows the instruction pipeline of the R3000. In the R3000, the pipeline advances once per clock cycle. The MIPS compiler is able to reorder instructions to fill delay slots with code 70 to 90% 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;
- Write back into register file.

As illustrated in **Figure 17.10a**, there is not only parallelism due to pipelining, but also parallelism within the execution of a single instruction. The 60-ns clock cycle is divided into two 30-ns stages. 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 stage, 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 *i* can address the ICACHE access of instruction $i+2$. Similarly, a load at instruction *i* fetches data that are immediately used by the OP of instruction $i+1$, while an ALU/shift result gets passed directly into instruction $i+1$ with no delay. This tight coupling between instructions makes for a highly efficient pipeline.

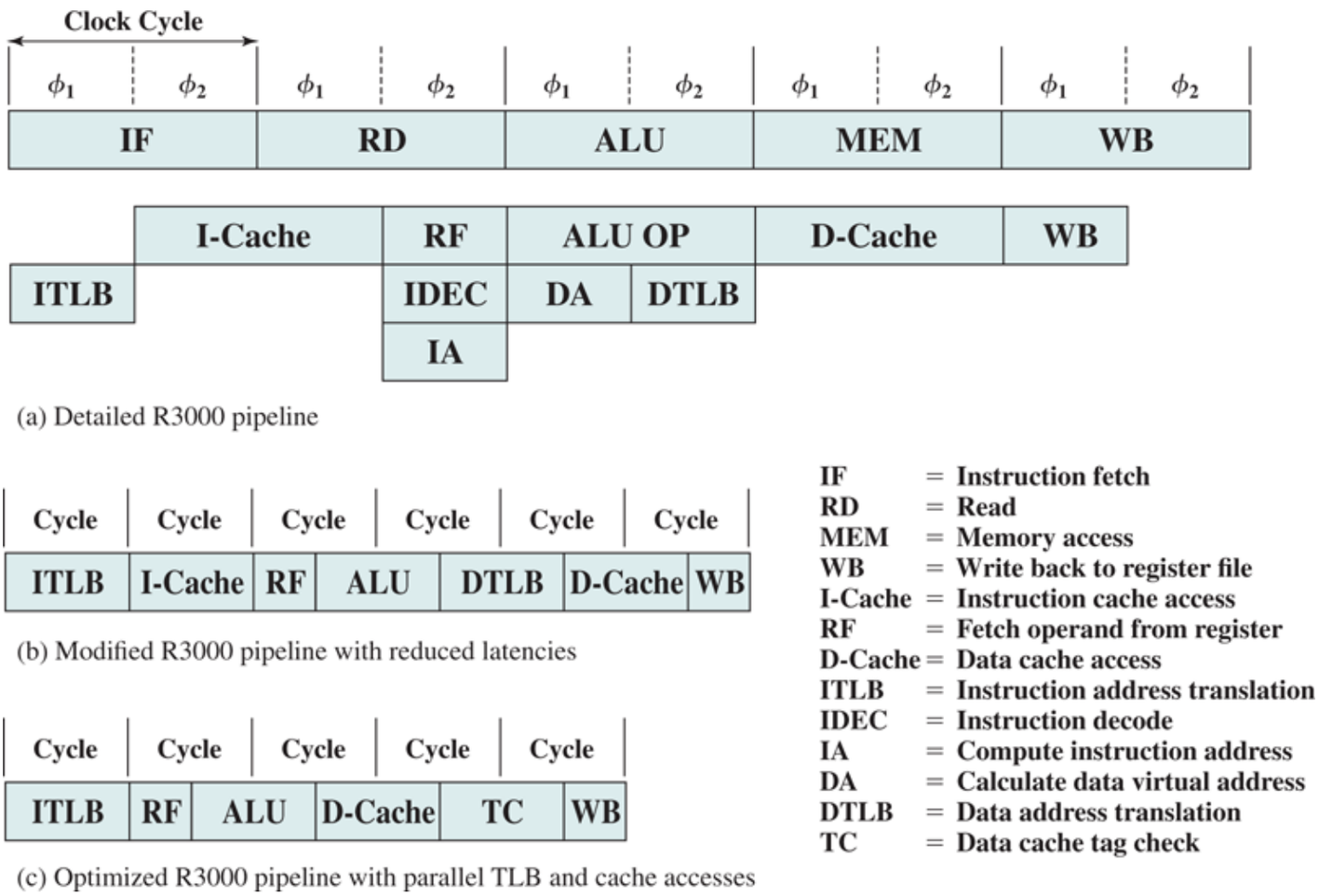


Figure 17.10 Enhancing the R3000 Pipeline

In detail, then, each clock cycle is divided into separate stages, denoted as $\phi 1$ and $\phi 2$. The functions performed in each stage are summarized in **Table 17.9**.

Table 17.9 R3000 Pipeline Stages

Pipeline	Phase	Function

Stage		
IF	$\phi 1$	Using the TLB, translate an instruction virtual address to a physical address (after a branching decision).
IF	$\phi 2$	Send the physical address to the instruction address.
RD	$\phi 1$	Return instruction from instruction cache.
		Compare tags and validity of fetched instruction.
RD	$\phi 2$	Decode instruction.
		Read register file.
		If branch, calculate branch target address.
ALU	$\phi 1 + \phi 2$	If register-to-register operation, the arithmetic or logical operation is performed.
ALU	$\phi 1$	If a branch, decide whether the branch is to be taken or not.
		If a memory reference (load or store), calculate data virtual address.
ALU	$\phi 2$	If a memory reference, translate data virtual address to physical using TLB.
MEM	$\phi 1$	If a memory reference, send physical address to data cache.
MEM	$\phi 2$	If a memory reference, return data from data cache, and check tags.
WB	$\phi 1$	Write to register file.

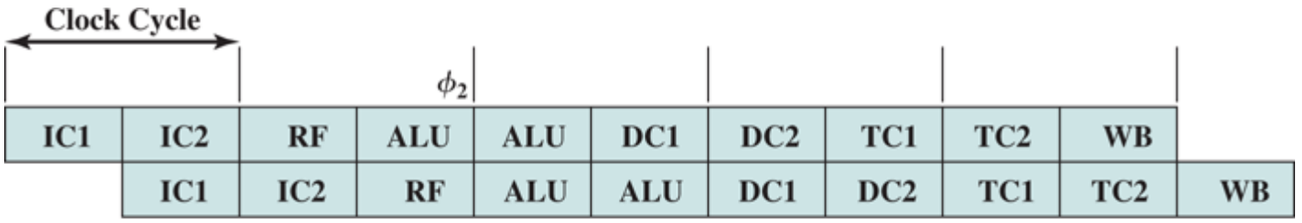
The R4000 incorporates a number of technical advances over the R3000. The use of more advanced technology allows the clock cycle time to be cut in half, to 30 ns, and for the access time to the register file to be cut in half. In addition, there is greater density on the chip, which enables the instruction and data caches to be incorporated on the chip. Before looking at the final R4000 pipeline, let us consider how the R3000 pipeline can be modified to improve performance using R4000 technology.

Figure 17.10b shows a first step. Remember that the cycles in this figure are half as long as those in **Figure 17.10a**. Because they are on the same chip, the instruction and data cache stages take only half as long; so they still occupy only one clock cycle. Again, because of the speedup of the register file access, register read and write still occupy only half of a clock cycle.

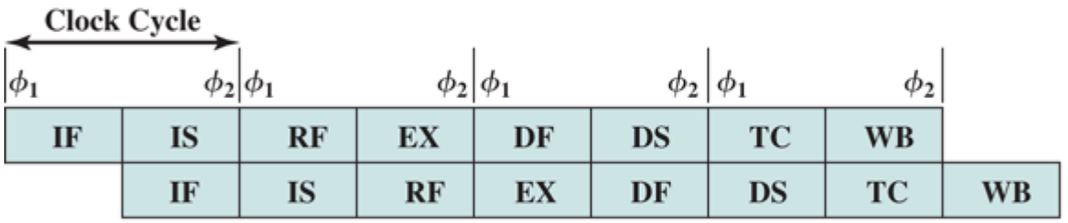
Because the R4000 caches are on-chip, the virtual-to-physical address translation can delay the cache access. This delay is reduced by implementing virtually indexed caches and going to a parallel cache access and address translation. **Figure 17.10c** shows the optimized R3000 pipeline with this

improvement. Because of the compression of events, the data cache tag check is performed separately on the next cycle after cache access. This check determines whether the data item is in the cache.

In a superpipelined system, existing hardware is used several times per cycle by inserting pipeline registers to split up each pipe stage. Essentially, each superpipeline stage operates at a multiple of the base clock frequency, the multiple depending on the degree of superpipelining. The R4000 technology has the speed and density to permit superpipelining of degree 2. **Figure 17.11a** shows the optimized R3000 pipeline using this superpipelining. Note that this is essentially the same dynamic structure as **Figure 17.10c**.



(a) Superpipelined implementation of the optimized R3000 pipeline



(b) R4000 pipeline

IF = Instruction fetch first half
IS = Instruction fetch second half
RF = Fetch operands from register
EX = Instruction execute
IC = Instruction cache
DC = Data cache
DF = Data cache first half
DS = Data cache second half
TC = Tag check
WB = Write back to register file

Figure 17.11 Theoretical R3000 and Actual R4000 Superpipelines

Further improvements can be made. For the R4000, a much larger and specialized adder was designed. This makes it possible to execute ALU operations at twice the rate. Other improvements allow the execution of loads and stores at twice the rate. The resulting pipeline is shown in **Figure 17.11b**.

The R4000 has eight pipeline stages, meaning that as many as eight instructions can be in the pipeline at the same time. The pipeline advances at the rate of two stages per clock cycle. The eight pipeline stages are as follows:

- **Instruction fetch first half:** Virtual address is presented to the instruction cache and the translation lookaside buffer.
- **Instruction fetch second half:** Instruction cache outputs the instruction and the TLB generates the physical address.
- **Register file:** Three activities occur in parallel:
  —Instruction is decoded and check made for interlock conditions (i.e., this instruction depends on

the result of a preceding instruction).

—Instruction cache tag check is made.

—Operands are fetched from the register file.

- **Instruction execute:** One of three activities can occur:
  —If the instruction is a register-to-register operation, the ALU performs the arithmetic or logical operation.

  —If the instruction is a load or store, the data virtual address is calculated.

  —If the instruction is a branch, the branch target virtual address is calculated and branch conditions are checked.

- **Data cache first:** Virtual address is presented to the data cache and TLB.
- **Data cache second:** The TLB generates the physical address, and the data cache outputs the data.
- **Tag check:** Cache tag checks are performed for loads and stores.
- **Write back:** Instruction result is written back to register file.

17.7 SPARC

SPARC (Scalable Processor Architecture) refers to an architecture defined by Sun Microsystems. Sun developed its own SPARC implementation but also licenses the architecture to other vendors to produce SPARC-compatible machines. The SPARC architecture is inspired by the Berkeley RISC I machine, and its instruction set and register organization is based closely on the Berkeley RISC model.

SPARC Register Set

As with the Berkeley RISC, the SPARC makes use of register windows. Each window gives addressability to 24 registers, and the total number of windows is implementation dependent and ranges from 2 to 32 windows. **Figure 17.12** illustrates an implementation that supports 8 windows, using a total of 136 physical registers; as the discussion in **Section 17.2** indicates, this seems a reasonable number of windows. Physical registers 0 through 7 are global registers shared by all procedures. Each procedure sees logical registers 0 through 31. Logical registers 24 through 31, referred to as *ins,* are shared with the calling (parent) procedure; and logical registers 8 through 15, referred to as *outs*, are shared with any called (child) procedure. These two portions overlap with other windows. Logical registers 16 through 23, referred to as *locals,* are not shared and do not overlap with other windows. Again, as the discussion of **Section 16.1** indicates, the availability of 8 registers for parameter passing should be adequate in most cases (e.g., see **Table 17.4**).

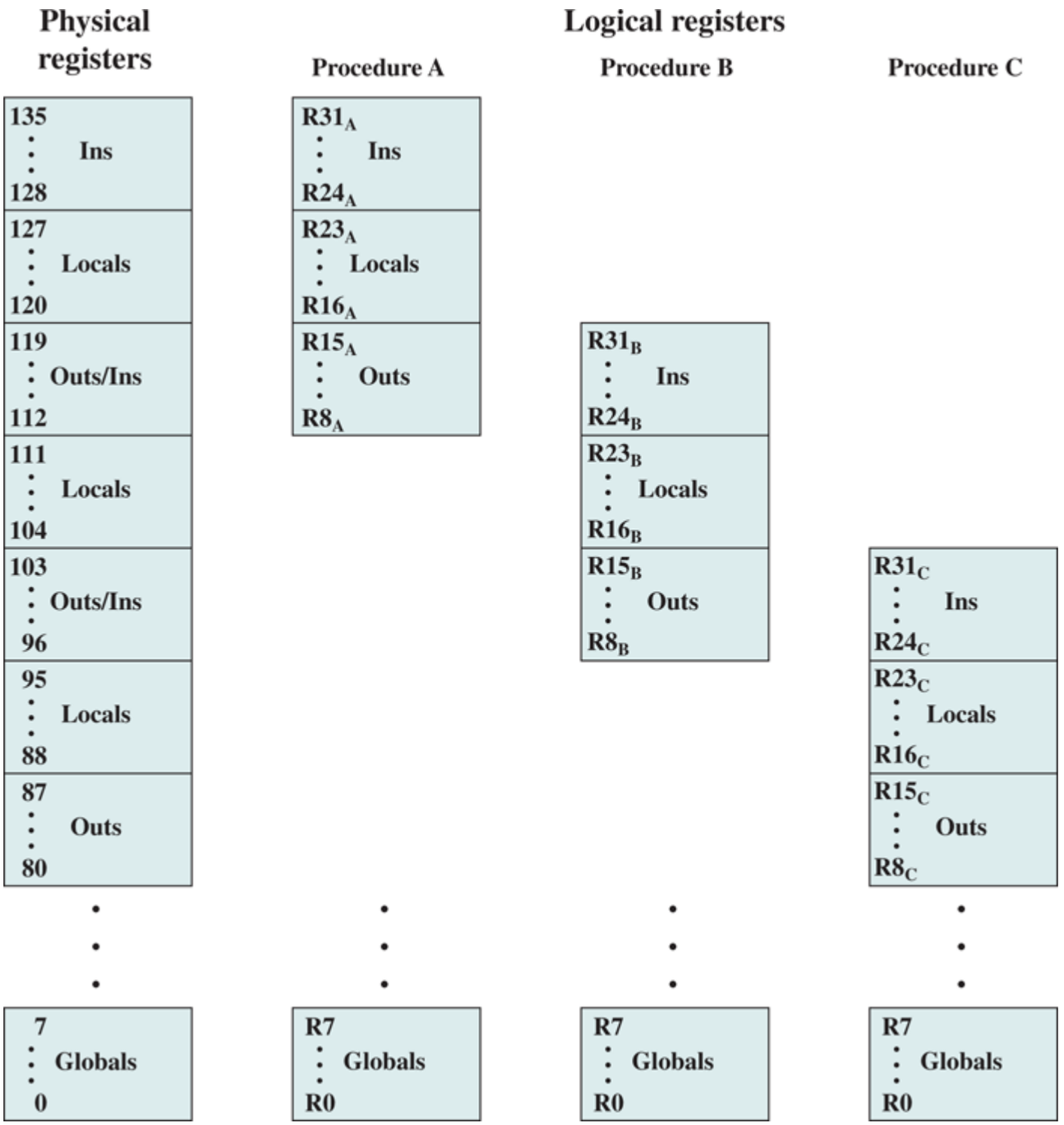


Figure 17.12 SPARC Register Window Layout with Three Procedures

Figure 17.13 is another view of the register overlap. The calling procedure places any parameters to be passed in its *outs* registers; the called procedure treats these same physical registers as its *ins* registers. The processor maintains a current window pointer (CWP), located in the processor status register (PSR), that points to the window of the currently executing procedure. The window invalid mask (WIM), also in the PSR, indicates which windows are invalid.

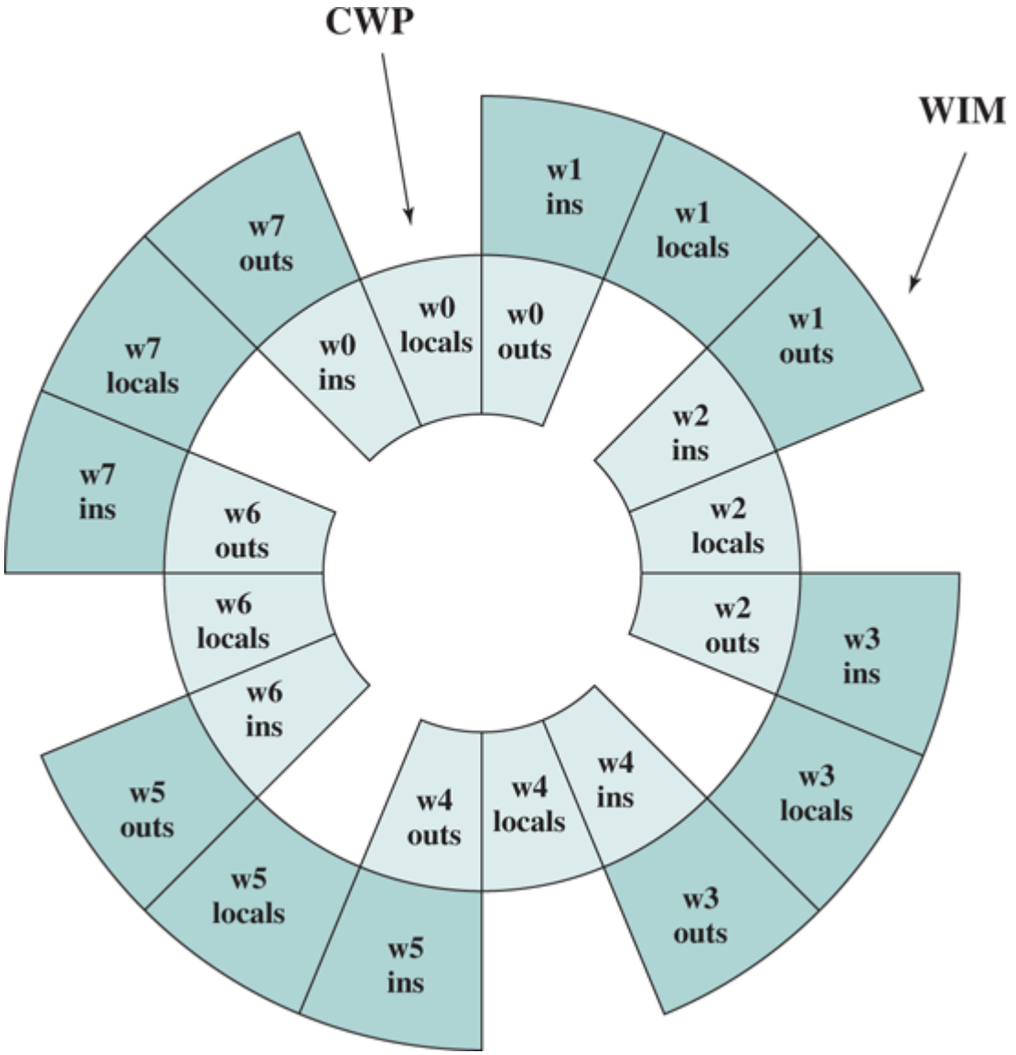


Figure 17.13 Eight Register Windows Forming a Circular Stack in SPARC

With the SPARC register architecture, it is usually not necessary to save and restore registers for a procedure call. The compiler is simplified because the compiler need be concerned only with allocating the local registers for a procedure in an efficient manner, and need not be concerned with register allocation between procedures.

Instruction Set

Most of the SPARC instructions reference only register operands. Register-to-register instructions have three operands and can be expressed in the form

$$R_d \rightarrow R_{S1} \text{op } S2$$

where R_d 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 and their negations.
- Shift left logical, right logical, or right arithmetic.

All of these instructions, except the shifts, can optionally set the four condition codes (ZERO, NEGATIVE, OVERFLOW, CARRY). Signed integers are represented in 32-bit twos complement form.

Only simple load and store instructions reference memory. There are separate load and store instructions for word (32 bits), doubleword, 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 zeros.

The only available addressing mode, other than register, is a displacement mode. That is, the effective address (EA) of an operand consists of a displacement from an address contained in a register:

$$\begin{aligned} \text{EA} &= (\text{R}_{\text{S1}}) + \text{S2} \\ \text{or EA} &= (\text{R}_{\text{S1}}) + (\text{R}_{\text{S2}}) \end{aligned}$$

depending on whether the second operand is immediate or a register reference. To perform a load or store, an extra stage is added to the instruction cycle. During the second stage, the memory address is calculated using the ALU; the load or store occurs in a third stage. This single addressing mode is quite versatile and can be used to synthesize other addressing modes, as indicated in **Table 17.10**.

Table 17.10 Synthesizing Other Addressing Modes with SPARC Addressing Modes
Note: $\text{S2} = \text{either}$ a register operand or a 13-bit immediate operand.

Instruction Type	Addressing Mode	Algorithm	SPARC Equivalent
Register-to-register	Immediate	$\text{operand} = \text{A}$	S2
Load, store	Direct	$\text{EA} = \text{A}$	$\text{R}_0 + \text{S}_2$
Register-to-register	Register	$\text{EA} = \text{R}$	$\text{R}_{\text{S1}}, \text{S}_{\text{S2}}$
Load, store	Register Indirect	$\text{EA} = (\text{R})$	$\text{R}_{\text{S1}} + 0$
Load, store	Displacement	$\text{EA} = (\text{R}) + \text{A}$	$\text{R}_{\text{S1}} + \text{S}_2$

It is instructive to compare the SPARC addressing capability with that of the MIPS. The MIPS makes use of a 16-bit offset, compared with a 13-bit offset on the SPARC. On the other hand, the MIPS does not permit an address to be constructed from the contents of two registers.

Instruction Format

As with the MIPS R4000, SPARC uses a simple set of 32-bit instruction formats (**Figure 17.14**). All instructions begin with a 2-bit opcode. For most instructions, this is extended with additional opcode bits elsewhere in the format. For the Call instruction, a 30-bit immediate operand is extended with two zero bits to the right to form a 32-bit PC-relative address in twos complement form. Instructions are aligned on a 32-bit boundary so that this form of addressing suffices.

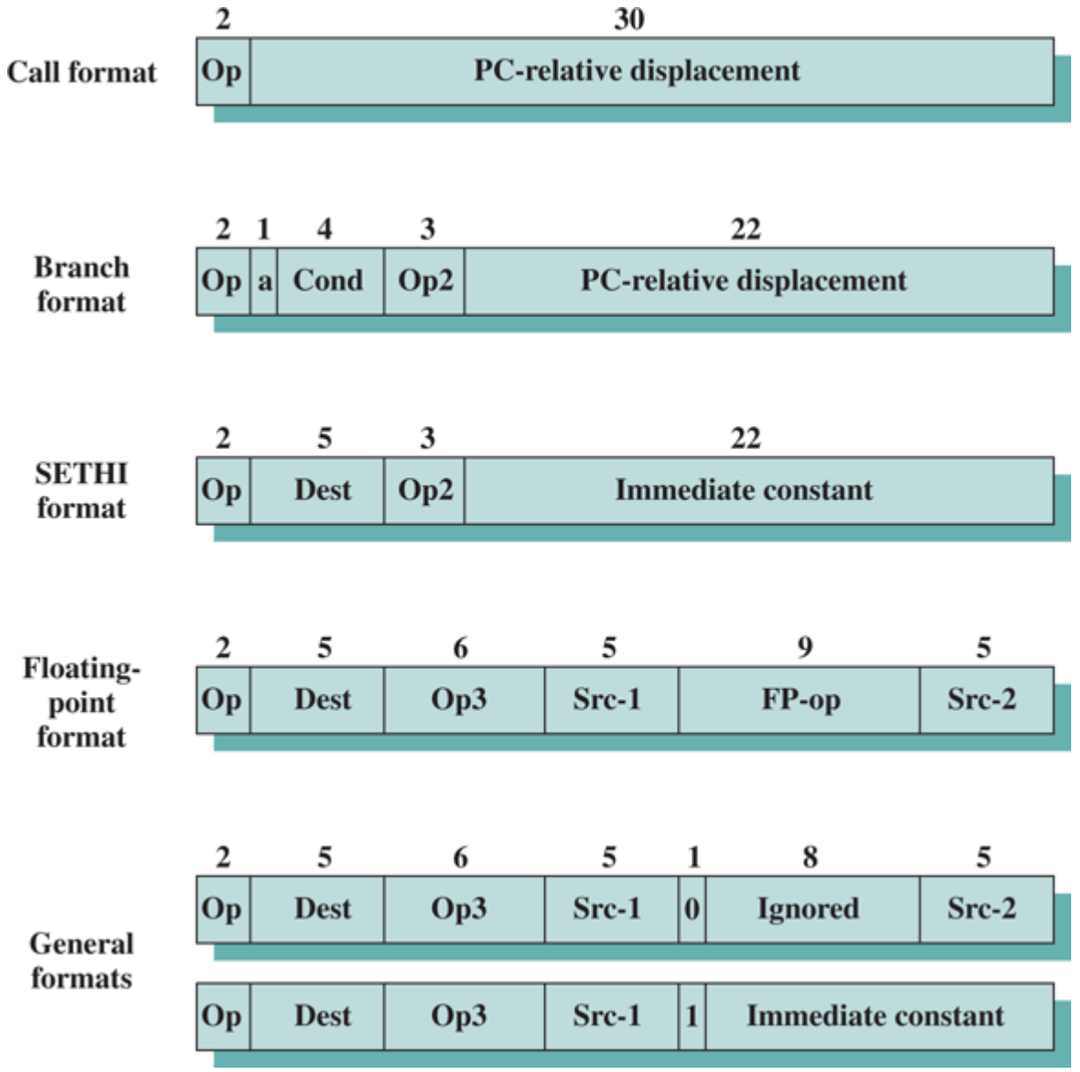


Figure 17.14 SPARC Instruction Formats

The Branch instruction includes a 4-bit condition field that corresponds to the four standard condition code bits, so that any combination of conditions can be tested. The 22-bit PC-relative address is extended with two zero bits on the right to form a 24-bit twos complement relative address. An unusual feature of the Branch instruction is the annul bit. When the annul bit is not set, the instruction after the branch is always executed, regardless of whether the branch is taken. This is the typical delayed branch operation found on many RISC machines and described in **Section 17.5** (see **Figure 17.7**). However, when the annul bit is set, the instruction following the branch is executed only if the branch is taken. The processor suppresses the effect of that instruction even though it is already in the pipeline. This annul bit is useful because it makes it easier for the compiler to fill the delay slot following a conditional branch. The instruction that is the target of the branch can always be put in the delay slot, because if the branch is not taken, the instruction can be annulled. The reason this technique is desirable is that conditional branches are generally taken more than half the time.

The SETHI instruction is a special instruction used to form a 32-bit constant. This feature is needed to form large data constants; for example, it can be used to form a large offset for a load or store instruction. The SETHI instruction sets the 22 high-order bits of a register with its 22-bit immediate operand, and zeros out the low-order 10 bits. An immediate constant of up to 13 bits can be specified in one of the general formats, and such an instruction could be used to fill in the remaining 10 bits of the register. A load or store instruction can also be used to achieve a direct addressing mode. To load a value from location K in memory, we could use the following SPARC instructions:

sethi	%hi(K), %r8	;load high-order 22 bits of address of location
		;K into register r8
Ld	[%r8 + %lo (K)] , %r8	;load contents of location K into r8

The macros %hi and %lo are used to define immediate operands consisting of the appropriate address bits of a location. This use of SETHI is similar to the use of the lui instruction on the MIPS.

The floating-point format is used for floating-point operations. Two source and one destination registers are designated.

Finally, all other operations, including loads, stores, arithmetic, and logical operations use one of the last two formats shown in **Figure 17.14**. One of the formats makes use of two source registers and a destination register, while the other uses one source register, one 13-bit immediate operand, and one destination register.

17.8 Processor Organization For Pipelining

This section looks at some of the enhancements to the pipeline illustrated in Figure 16.23 that can be used to improve performance. To begin, we merge the instruction decode (ID) and operand fetch (OF) stages into a single ID stage, with the ID stage responsible for instruction decode and register operand fetch. This is appropriate for a RISC machine, in which most operands are register. It can also be used in CISC machines. In either case, memory operand fetch is deferred to the load/store unit (LSU).

Figure 17.15 shows the addition of three enhancements: the instruction buffer, the store buffer, and the predecoder. All are designed to smooth out and enhance the flow of instructions through the pipeline. The **instruction buffer** supports the function of instruction prefetching. The objective is to prevent, or at least minimize, delays in issuing instructions due to L1 instruction cache misses. Without instruction prefetching, when a cache miss occurs in the L1 instruction cache, the pipeline freezes until new instructions are brought from the L2 cache into the L1 cache. To counter this, the IF stage can fetch multiple instructions to keep the instruction buffer full, so that when a miss occurs, the ID stage still has instructions to draw upon from the instruction buffer. Occasionally, when branches occur, instructions will need to be flushed from the instruction buffer. But overall performance is improved.

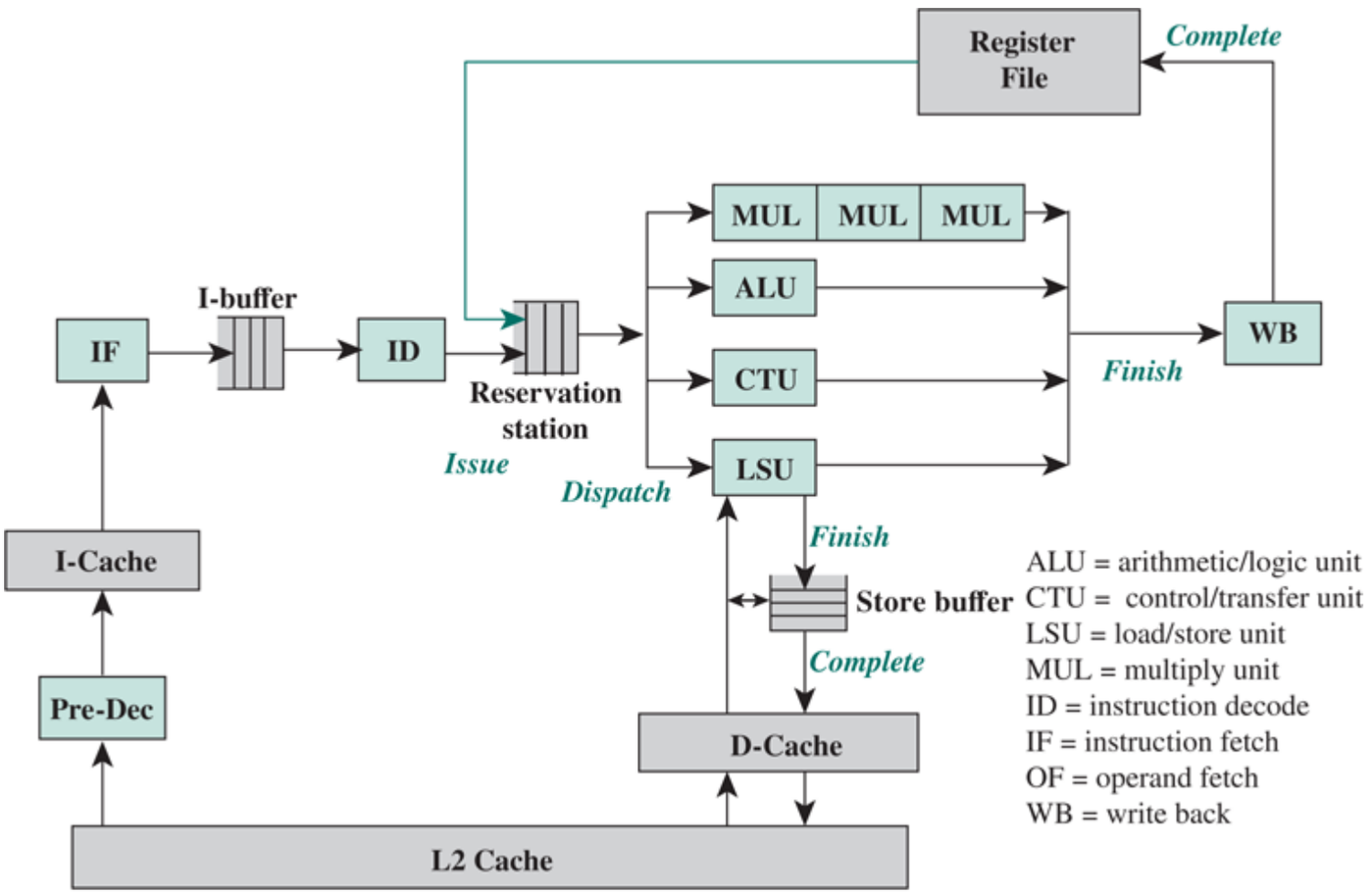


Figure 17.15 Pipeline Organization with Buffers and Pre-Decoding

The **predecoder (PD)** off-loads some of the tasks of the instruction decode (ID) stage to avoid the ID stage becoming a bottleneck. As previously discussed, the functions of the instruction decoder include decoding the opcode and operand fields and evaluating dependencies and hazards. As the pipeline

structure becomes more complex, these functions take more time, especially for a CISC machine such as the x86 architecture. However, as we move to a superscalar architecture in the next chapter, in which multiple instructions are decoded in parallel, even RISC architectures require substantial decoding time. The objective with the predecoder is to perform some of the decoding ahead of time to reduce the burden on the ID stage. The PD stage is inserted between the L2 cache and the L1 instruction cache. Because of the relatively slow L2 cache time, there is spare time here to perform the PD function. The PD may add a few bits to the instruction to designate the type of instruction and the resources required. For a CISC instruction architecture the PD may also determine instruction length and decode instruction prefixes.

Another buffer, the **store buffer**, improves performance on store operations. In essence, the store buffer allows a load to bypass the completion of a store to access a memory location. Thus, the load instruction can make use of a data item as soon as it is created (finished) and does not have to wait for the data item to be stored in the data cache (completed). This feature is especially useful for loop instructions, in which data created in one iteration is used immediately in the next iteration. On loads, the LSU examines the store buffer as a lookaside buffer and extracts the data from there if available; otherwise, it queries the data cache as normal.

Figure 17.15 shows the pipeline structure with the addition of the instruction buffer, the store buffer, and the predecoder to the organization shown in Figure 16.23. Next, we consider three more features to enhance performance: multiple reservation stations, forwarding, and the reorder buffer.

The reservation station was described in **Section 16.5** (see **Figure 16.24**). It overcomes the bottleneck problem that the ID cannot receive a new instruction until the preceding instruction is issued. If the corresponding functional unit (ALU, CTU, LSU, etc.) is not available, then the ID stage stalls. The reservation station allows the ID to issue the instruction into the reservation station, which can buffer multiple instructions pending their dispatch to the appropriate functional unit (FU). Figure 16.23 shows a single reservation station that serves all the FUs. The control of such an arrangement is relatively complex and is rarely used. The Pentium is an example of a system that used a single reservation station. An improvement both in performance and simplicity is the use of a **dedicated reservation station** for each individual FU. The process of dispatching an instruction to a functional unit proceeds in two parts:

- **Issue from ID to reservation station:** Each slot in the reservation station serves the role of a virtual FU, to which the ID issues an instruction. There is no stall unless all the slots in the reservation station for a given FU are in use (buffer full).
- **Dispatch from reservation station to FU:** Dispatch occurs when the corresponding FU is available and all operand values are available. However, dispatch does not have to be FIFO, but rather different priorities can be assigned to different instructions.

The reservation station is also referred to as an **instruction window**, particularly in the superscalar literature, and we use this latter term in **Chapter 18**.

Data forwarding addresses the problem of read-after-write (RAW) delays due to WB delays. As with the store buffer, data forwarding makes data available as soon as it is created. The forwarded data becomes input to the reservation stations, going to an operand field.

The **reorder buffer** supports out-of order execution. Out-of-order execution (OoOE) is an approach to processing that allows instructions for high-performance microprocessors to begin execution as soon as their operands are ready. Although instructions are issued in order, they can proceed out-of-order (OoO) with respect to each other. The goal of OoO processing is to allow the processor to avoid a class of stalls that occur when the data needed to perform an operation are unavailable. OoOE is discussed in **Chapter 18**. The reorder buffer ensures that instructions complete in order. Reorder buffers are discussed in **Appendix G**.

Figure 17.16 adds multiple reservation stations, forwarding, and the reorder buffer to the organization shown in **Figure 17.15**. Note that the forwarding function occurs at finishing time, prior to the reorder buffer and completion of write back.

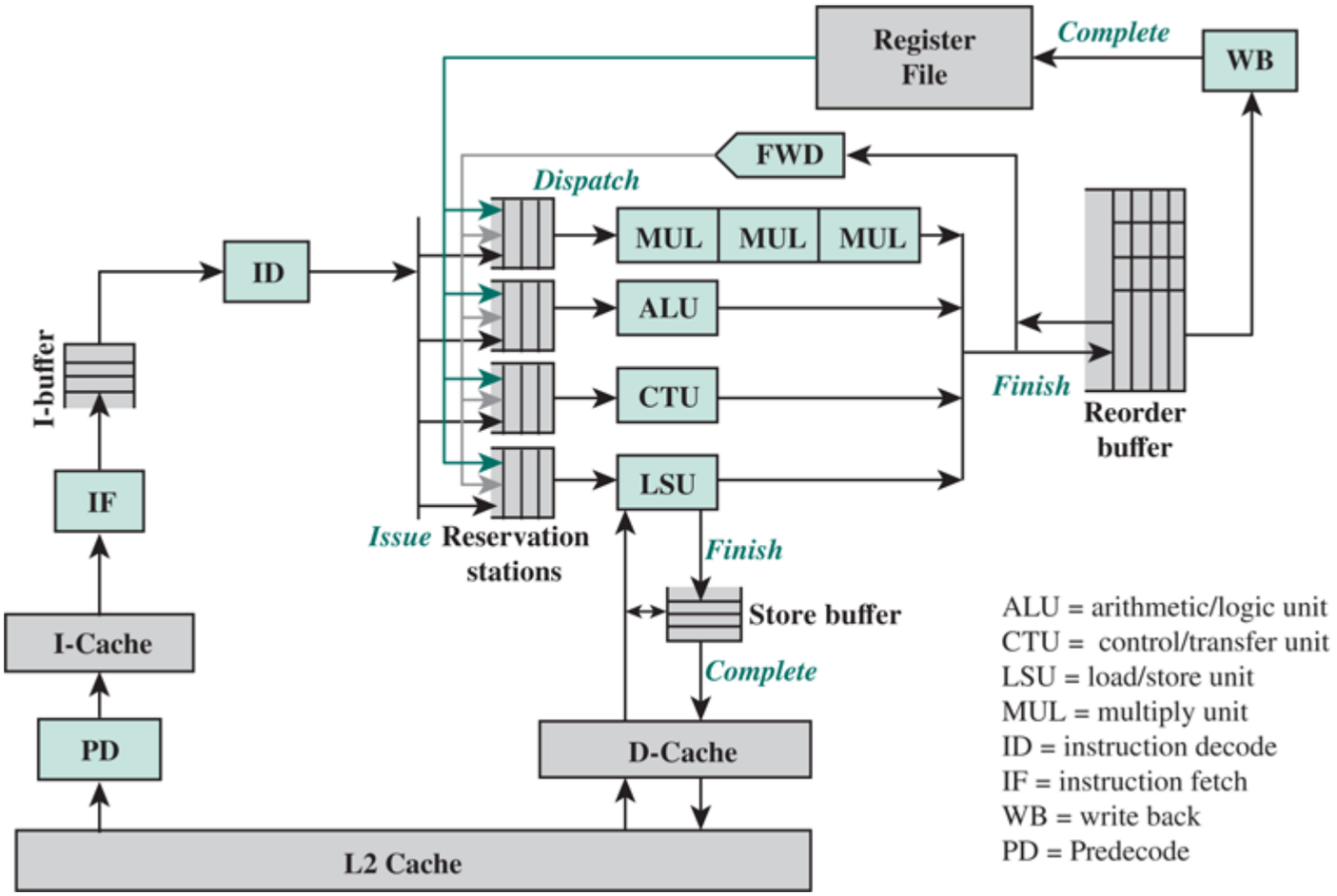


Figure 17.16 Pipeline Organization with Forwarding, Reorder Buffer, and Multiple Reservation Stations

17.9 CISC, RISC, And Contemporary Systems

During the 1980s, there was quite a RISC versus CISC controversy concerning the relative performance of each approach, with no clear-cut resolution of the issue. In more recent years, the RISC versus CISC controversy has died down to a great extent. This is because there has been a gradual convergence of the technologies. As chip densities and raw hardware speeds increase, RISC systems have become more complex. At the same time, in an effort to squeeze out maximum performance, CISC designs have focused on issues traditionally associated with RISC, such as an increased number of general-purpose registers and increased emphasis on instruction pipeline design.

At the time of the introduction of RISC, the focus of computer architecture design were desktops and servers, the primary design objective was performance, and the primary design constraints were chip area and processor design complexity. This situation has dramatically changed with the proliferation of embedded devices built on ARM, primarily a RISC-based architecture, while the primarily CISC x86 continues to dominate larger systems, including laptops, desktops, and servers. There is much more emphasis on power consumption as a design constraint. A study reported in [BLEM15] found that the details of implementation plus the presence or absence of specializations such as floating point and SIMD support were major factors in both performance and power consumption, but that the use of a CISC or RISC instruction set architecture was not a significant factor.

A major conclusion that can be deduced is that differences in the instruction set architecture (RISC vs. CISC) do affect implementation choices, but because of modern microarchitecture techniques, these differences do not drive performance or power consumption. An example of modern microarchitecture techniques is the translation of machine instructions into microinstructions on contemporary x86 implementations.

17.10 Key Terms, Review Questions, and Problems

Key Terms

complex instruction set computer (CISC)

data forwarding

dedicated reservation station

delayed branch

delayed load

high-level language (HLL)

instruction buffer

predecoder

reduced instruction set computer (RISC)

register file

register window

reorder buffer

SPARC

store buffer

Review Questions

17.1 What are some typical distinguishing characteristics of RISC organization?
17.2 Briefly explain the two basic approaches used to minimize register-memory operations on RISC machines.
17.3 If a circular register buffer is used to handle local variables for nested procedures, describe two approaches for handling global variables.
17.4 What are some typical characteristics of a RISC instruction set architecture?
17.5 What is a delayed branch?

Problems

17.1 In the discussion of **Figure 17.2** , it was stated that only the first two portions of a window are saved or restored. Why is it not necessary to save the temporary registers?
17.2 We wish to determine the execution time for a given program using the various pipelining schemes discussed in **Section 17.5** . Let

$$\begin{aligned} N &= \text{number of executed instructions} \\ D &= \text{number of memory accesses} \\ J &= \text{number of jump instructions} \end{aligned}$$

For the simple sequential scheme (**Figure 17.6a**), the execution time is $2N + D$ stages. Derive formulas for two-stage, three-stage, and four-stage pipelining.

17.3 Reorganize the code sequence in **Figure 17.6d** to reduce the number of NOOPs.
17.4 Consider the following code fragment in a high-level language:

```
for I  in 1…100  loop
    S ← S + Q(I). VAL
end loop;
```

Assume that Q is an array of 32-byte records and the VAL field is in the first 4 bytes of each record. Using x86 code, we can compile this program fragment as follows:

	MOV	ECX,1	;use register ECX to hold I
LP:	IMUL	EAX, ECX, 32	;get offset in EAX
	MOV	EBX, Q[EAX]	;load VAL field
	ADD	S, EBX	;add to S
	INC	ECX	;increment I
	CMP	ECX, 101	:compare to 101
	JNE	LP	;loop until $I = 100$

This program makes use of the IMUL instruction, which multiplies the second operand by the immediate value in the third operand and places the result in the first operand (see Problem 10.13). A RISC advocate would like to demonstrate that a clever compiler can eliminate unnecessarily complex instructions such as IMUL. Provide the demonstration by rewriting the above x86 program without using the IMUL instruction.
17.5 Consider the following loop:

```
S := 0;
for K:=1 to 100 do
      S:=S - K;
```

A straightforward translation of this into a generic assembly language would look something like this:

	LD	R1, 0	;keep value of S in R1
	LD	R2,1	;keep value of K in R2
LP	SUB	R1, R1, R2	;S : = S – K
	BEQ	R2, 100, EXIT	;done if $K = 100$
	ADD	R2, R2, 1	;else increment K

	JMP	LP	;back to start of loop

A compiler for a RISC machine will introduce delay slots into this code so that the processor can employ the delayed branch mechanism. The JMP instruction is easy to deal with, because this instruction is always followed by the SUB instruction; therefore, we can simply place a copy of the SUB instruction in the delay slot after the JMP. The BEQ presents a difficulty. We can't leave the code as is, because the ADD instruction would then be executed one too many times. Therefore, a NOP instruction is needed. Show the resulting code.

17.6 A RISC machine's compiler may do both a mapping of symbolic registers to actual registers and a rearrangement of instructions for pipeline efficiency. An interesting question arises as to the order in which these two operations should be done. Consider the following program fragment:

LD	SR1, A	;load A into symbolic register 1
LD	SR2, B	;load B into symbolic register 2
ADD	SR3, SR1, SR2	;add contents of SR1 and SR2 and store in SR3
LD	SR4, C	
LD	SR5, D	
ADD	SR6, SR4, SR5	

a. First do the register mapping and then any possible instruction reordering. How many machine registers are used? Has there been any pipeline improvement?
b. Starting with the original program, now do instruction reordering and then any possible mapping. How many machine registers are used? Has there been any pipeline improvement?

17.7 Add entries for the following processors to **Table 17.7** :

a. Pentium II
b. ARM

17.8 In many cases, common machine instructions that are not listed as part of the MIPS instruction set can be synthesized with a single MIPS instruction. Show this for the following:

a. Register-to-register move
b. Increment, decrement
c. Complement
d. Negate
e. Clear

17.9 A SPARC implementation has *K* register windows. What is the number *N* of physical registers?

17.10 SPARC is lacking a number of instructions commonly found on CISC machines. Some of these are easily simulated using either register R0, which is always set to 0, or a constant operand. These simulated instructions are called pseudoinstructions and are recognized by the SPARC assembler. Show how to simulate the following pseudoinstructions, each with a single

SPARC instruction. In all of these, src and dst refer to registers. (*Hint*: A store to R0 has no effect.)

a. MOV src, dst
b. COMPARE src1, src2
c. TEST src1
d. NOT dst
e. NEG dst
f. INC dst
g. DEC dst
h. CLR dst
i. NOP

17.11 Consider the following code fragment:

```
if K > 10
     L := K + 1
else
     L := K - 1
```

A straightforward translation of this statement into SPARC assembler could take the following form:

	sethi	%hi(K), %r8	;load high-order 22 bits of address of location
			;K into register r8
	ld	[%r8 + %lo(K)], %r8	;load contents of location K into r8
	cmp	%r8, 10	;compare contents of r8 with 10
	ble	L1	;branch if $(r8) \leq 10$
	nop		
	sethi	%hi(K), %r9	
	ld	[%r9 + %lo(K)], %r9	;load contents of location K into r9
	inc	%r9	;add 1 to (r9)
	sethi	%hi(L), %r10	
	st	%r9, [%r10 + %lo(L)]	;store (r9) into location L
	b	L2	
	nop		

L1:	sethi	%hi(K), %r11	
	ld	[%r11 + %lo(K)], %r12	;load contents of location K into r12
	dec	%r12	;subtract 1 from (r12)
	sethi	%hi(L), %r13	
	st	%r12, [%r13 + %lo(L)]	;store (r12) into location L
L2:			

The code contains a nop after each branch instruction to permit delayed branch operation.

a. Standard compiler optimizations that have nothing to do with RISC machines are generally effective in being able to perform two transformations on the foregoing code. Notice that two of the loads are unnecessary and that the two stores can be merged if the store is moved to a different place in the code. Show the program after making these two changes.
b. It is now possible to perform some optimizations peculiar to SPARC. The nop after the ble can be replaced by moving another instruction into that delay slot and setting the annul bit on the ble instruction (expressed as ble,a L1). Show the program after this change.
c. There are now two unnecessary instructions. Remove these and show the resulting program.