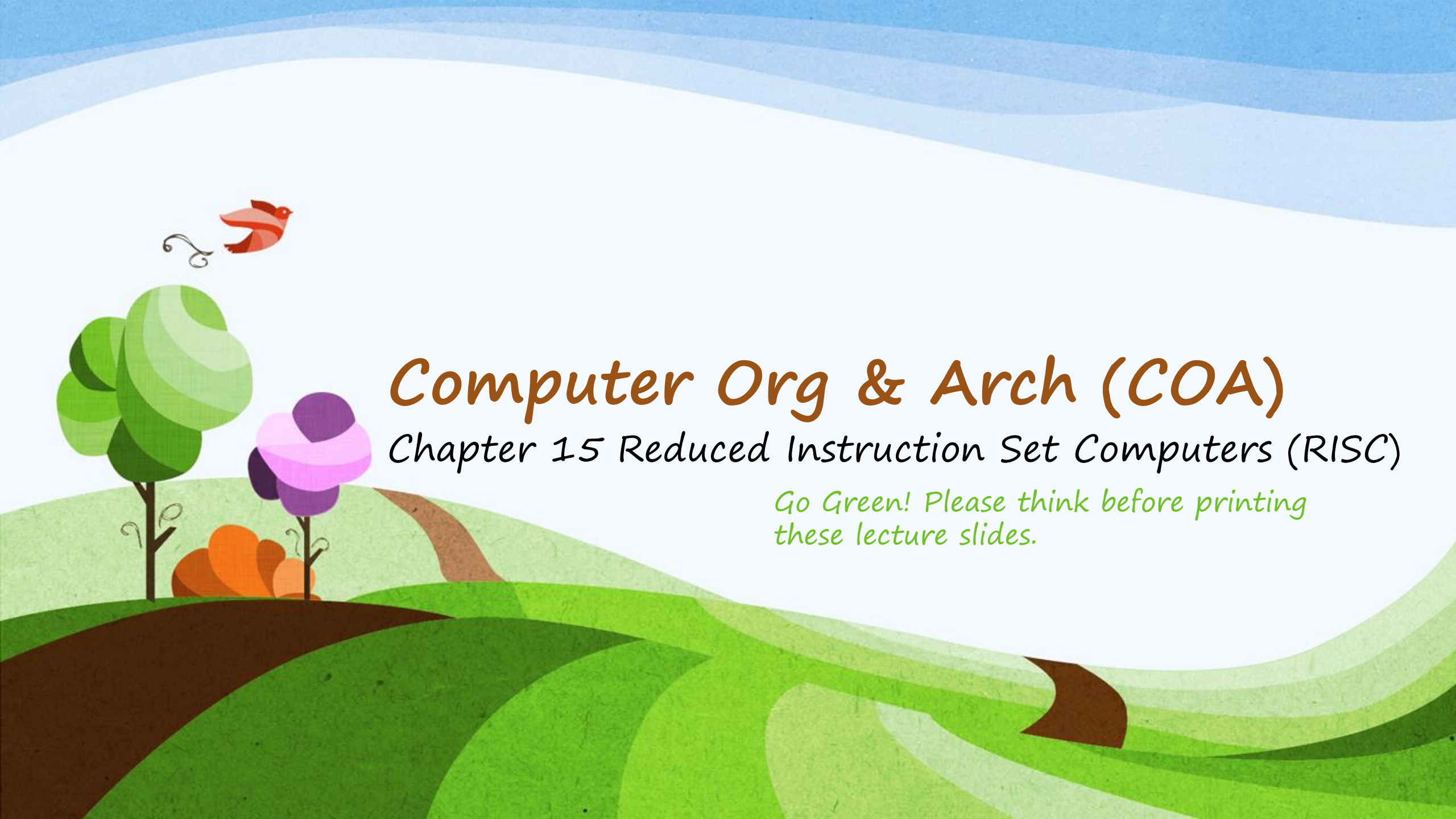


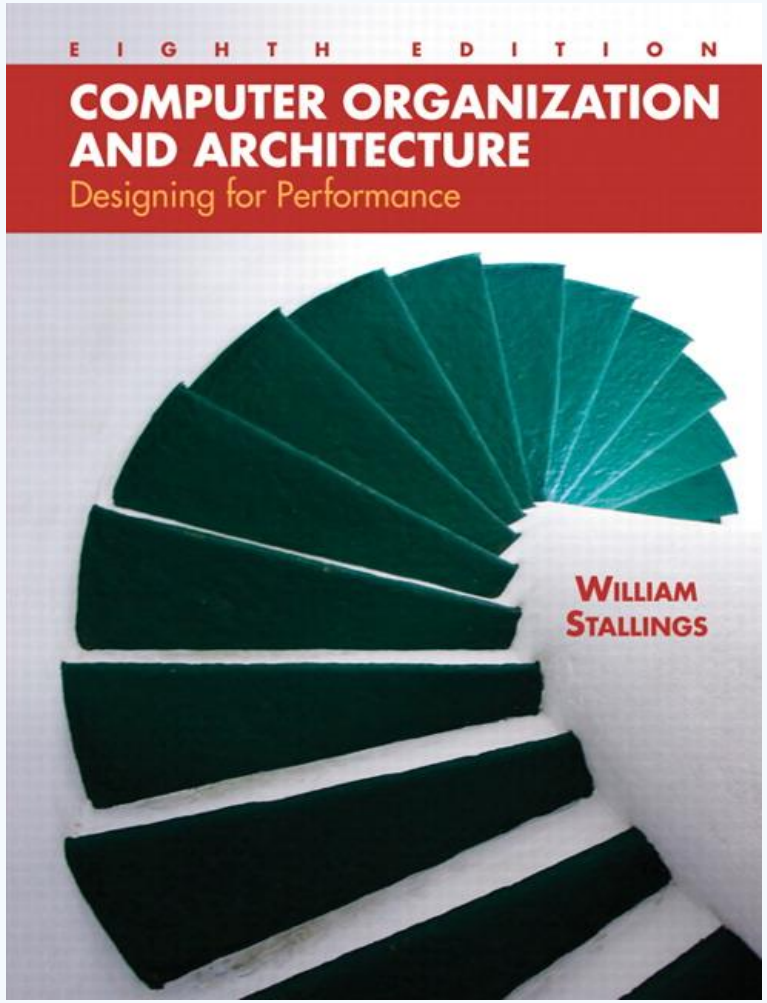
Computer Org & Arch (COA)

Chapter 15 Reduced Instruction Set Computers (RISC)

Go Green! Please think before printing these lecture slides.

Text Book

- **Title:** Computer Organization and Architecture: Designing for Performance
- **Author:** William Stallings
- **Edition:** 8th
- **Publisher:** Prentice-Hall India



Top level view Chapter 15 Reduced Instruction Set Computers (RISC)

- Instruction execution characteristics
  - Operations
  - Operands
  - Procedure calls
  - Implications
- The use of a large register file
  - Register windows
  - Global variables
  - Large register file versus cache
- Reduced instruction set architecture
  - Characteristics of RISC
  - CISC versus RISC characteristics
- RISC pipelining
  - Pipelining with regular instructions
  - Optimization of pipelining
- MIPS R4000
  - Instruction set
  - Instruction pipeline
- SPARC
  - SPARC register set
  - Instruction set
  - Instruction format
- Compiler-based register optimization
- RISC versus CISC controversy

PART FOUR: THE CENTRAL PROCESSING UNIT

Chapter 15
Reduced Instruction Set Computers (RISC)
(Assignment # 11 included in the last)

Please read slides to write elaborate answers to secure maximum marks in the exams.

Introduction

Since the development of the stored-program computer around 1950, there have been remarkably few true innovations in the areas of computer organization and architecture. 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.

Introduction

- **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

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.

Table 15.1
Characteristics of Some CISCs, RISCs, and Superscalar Processors

	Complex Instruction Set (CISC)Computer			Reduced Instruction Set (RISC) Computer		Superscalar		
Characteristic	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	PowerPC	Ultra SPARC	MIPS R10000
Year developed	1973	1978	1989	1987	1991	1993	1996	1996
Number of instructions	208	303	235	69	94	225		
Instruction size (bytes)	2–6	2–57	1–11	4	4	4	4	4
Addressing modes	4	22	11	1	1	2	1	1
Number of general-purpose registers	16	16	8	40 - 520	32	32	40 - 520	32
Control memory size (Kbits)	420	480	246	—	—	—	—	—
Cache size (KBytes)	64	64	8	32	128	16-32	32	64

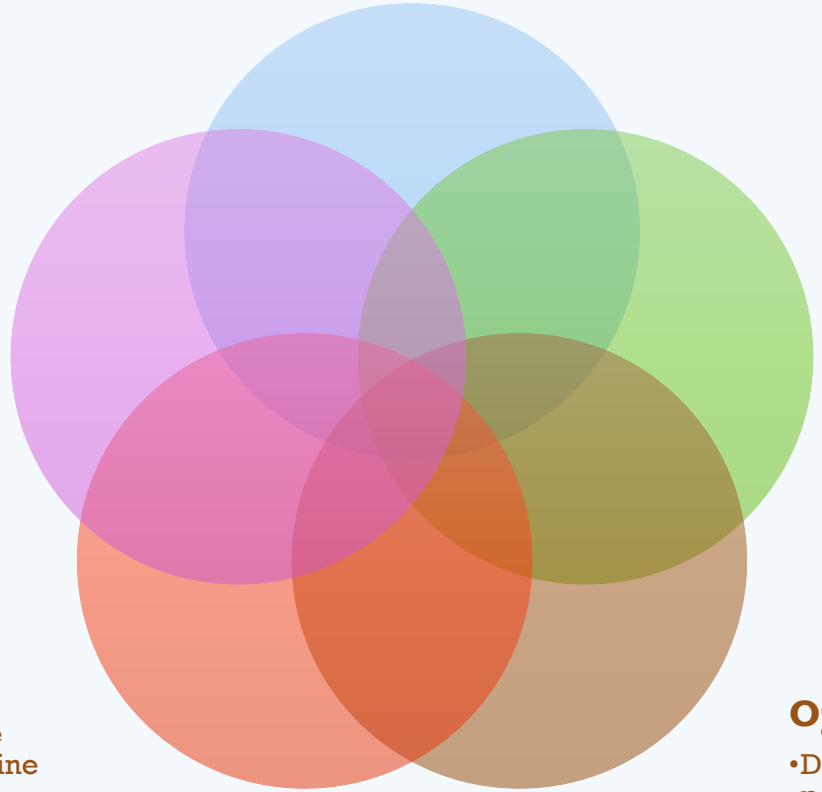
Table 15.1 Characteristics of Some CISCs, RISCs, and Superscalar Processors

Table 15.1 compares several RISC and non-RISC systems.

Instruction Execution Characteristics

High-level languages (HLLs)

- Allow the programmer to express algorithms more concisely
- Allow the compiler to take care of details that are not important in the programmer's expression of algorithms
- Often support naturally the use of structured programming and/or object-oriented design



Execution sequencing

- Determines the control and pipeline organization

Semantic gap

- The difference between the operations provided in HLLs and those provided in computer architecture

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

Operations performed

- Determine the functions to be performed by the processor and its interaction with memory

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.

Instruction Execution Characteristics

- 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.

Instruction Execution Characteristics

- 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 deter- mine 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.

Table 15.2 Weighted Relative Dynamic Frequency of HLL Operations

	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%

Table 15.2 Weighted Relative Dynamic Frequency of HLL Operations [PATT82a]

Table 15.2
Weighted Relative Dynamic Frequency of HLL Operations

- To get at this underlying phenomenon, the Patterson programs [PATT82a], described in Appendix 4A, were 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 15.2 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(proxy) 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 15.2
Weighted Relative Dynamic Frequency of HLL Operations

- The reader should be clear on the significance of Table 15.2. 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.

Table 15.3 Operands

	Pascal	C	Average
Integer Constant	16%	23%	20%
Scalar Variable	58%	53%	55%
Array/Structure	26%	24%	25%

Table 15.3 Dynamic Percentage of Operands

Table 15.3 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 15.3). 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 (dominance) of references to scalars, and these are highly localized.

Table 15.3 Operands

- 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.

Table 15.4
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%

Table 15.4 Procedure Arguments and Local Scalar Variables

Table 15.4
Procedure Arguments and Local Scalar Variables

- We have seen that procedure calls and returns are an important aspect of HLL programs. The evidence (Table 15.2) 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.

- Tanenbaum's study [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 15.4. 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 15.4
Procedure Arguments and Local Scalar Variables

- The same Berkeley group also looked at the pattern of procedure calls and returns in HLL programs. They found that it is rare to have a long uninterrupted sequence of procedure calls followed by the corresponding sequence of returns. Rather, they found that a program remains confined to a rather narrow window of procedure-invocation depth. This is illustrated in Figure 4.21, which was discussed in Chapter 4. These results reinforce the conclusion that operand references are highly localized.

Implications

- HLLs can best be supported by optimizing performance of the most time-consuming features of typical HLL programs
- Three elements characterize RISC architectures:
  - Use a large number of registers or use a compiler to optimize register usage
  - Careful attention needs to be paid to the design of instruction pipelines
  - Instructions should have predictable costs and be consistent with a high-performance implementation

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.

Implications

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

The Use of a Large Register File

Software Solution

- Requires compiler to allocate registers
- Allocates based on most used variables in a given time
- Requires sophisticated program analysis

Hardware Solution

- More registers
- Thus more variables will be in registers

The Use of a Large Register File

- 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 to minimize register-memory operations.
- Two basic approaches are possible, one based on software and the other on hardware. The software approach is to rely on the compiler to maximize register usage. The compiler will attempt to 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.

Overlapping Register Windows

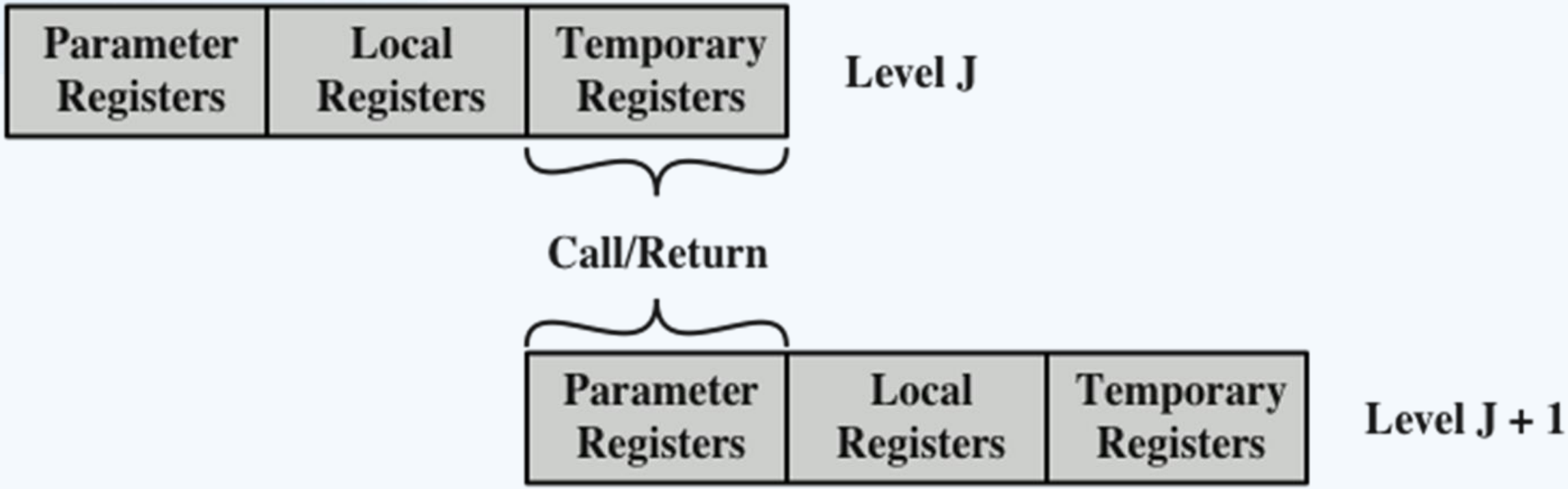


Figure 15.1 Overlapping Register Windows

Overlapping 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.

Overlapping Register Windows

- The solution is based on two other results reported in Section 15.1. First, a typical procedure employs only a few passed parameters and local variables (Table 15.4). Second, the depth of procedure activation fluctuates within a relatively narrow range (Figure 4.21). 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 15.1. 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 from the local and temporary registers at level J + 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 15.7, and the IA-64 architecture used in Intel's Itanium processor.

Circular Buffer Organization of Overlapped Windows

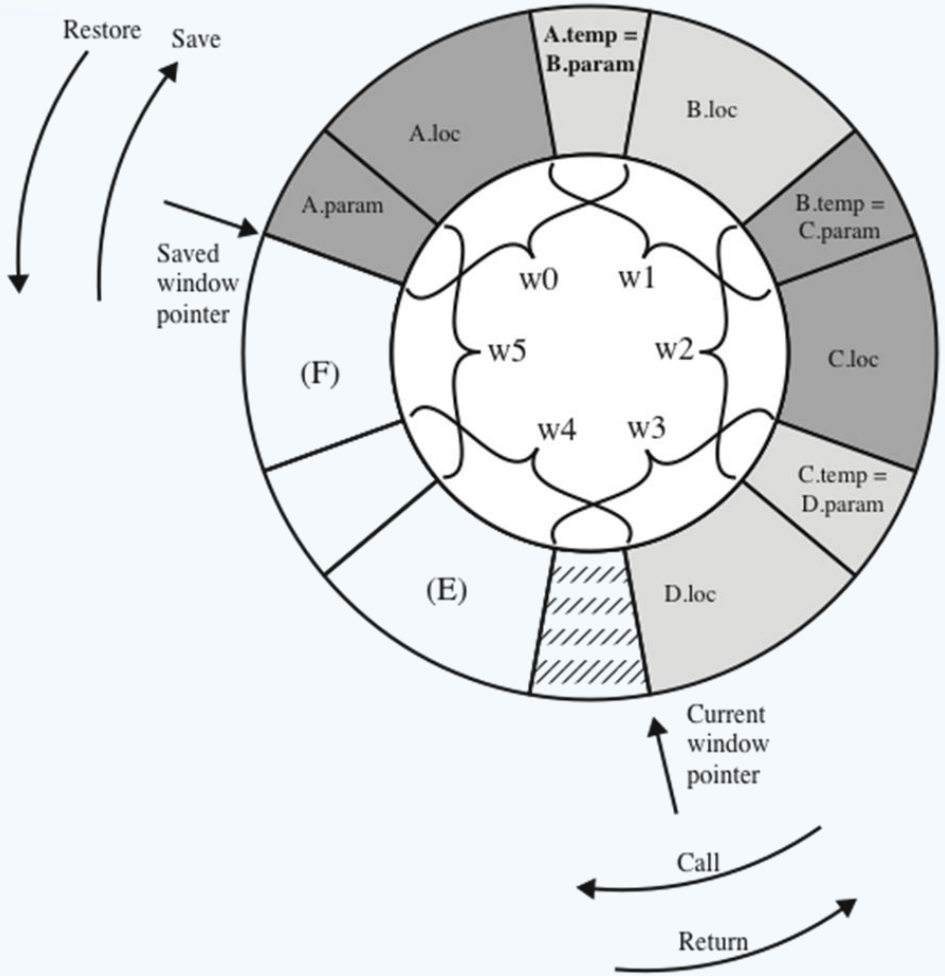


Figure 15.2 Circular-Buffer Organization of Overlapped Windows

Circular Buffer Organization of Overlapped Windows

- The circular organization is shown in Figure 15.2, 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.

- 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.

Circular Buffer Organization of Overlapped Windows

- From the preceding, it can be seen that an *N-window* register file can hold only *N* – 1 procedure activations. The value of *N* need not be large. As was mentioned in Appendix 4A, 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

- 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
  - However, for frequently accessed global variables this scheme is inefficient
- 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
- 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

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.

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

Table 15.5 Characteristics of Large-Register-File and Cache Organizations

Characteristics of Large-Register-File and Cache Organizations

- 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 15.5 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.

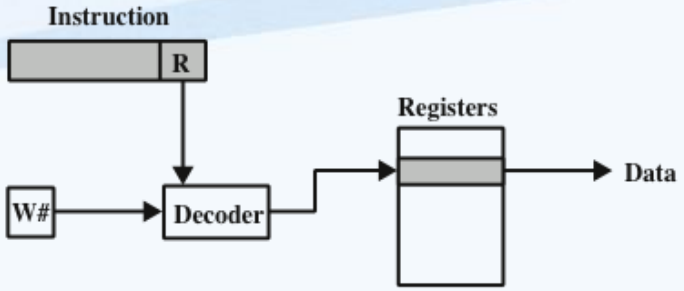
Characteristics of Large-Register-File and Cache Organizations

- 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.

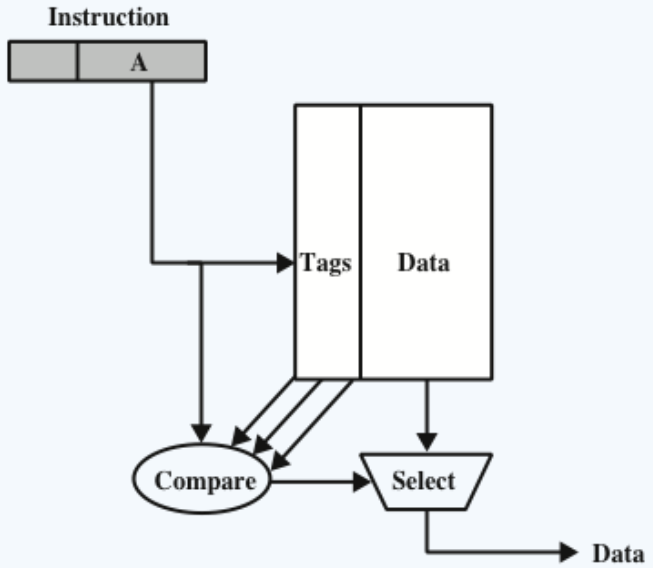
- The cache is capable of handling global as well as local variables. There are usually many global scalars, but only a few of them are heavily used [KATE83]. A cache will dynamically discover these variables and hold them. If the window- based register file is supplemented with global registers, it 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.

Referencing a Scalar



(a) Windows-based register file



(b) Cache

Figure 15.3 Referencing a Scalar

Referencing a Scalar

- 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.

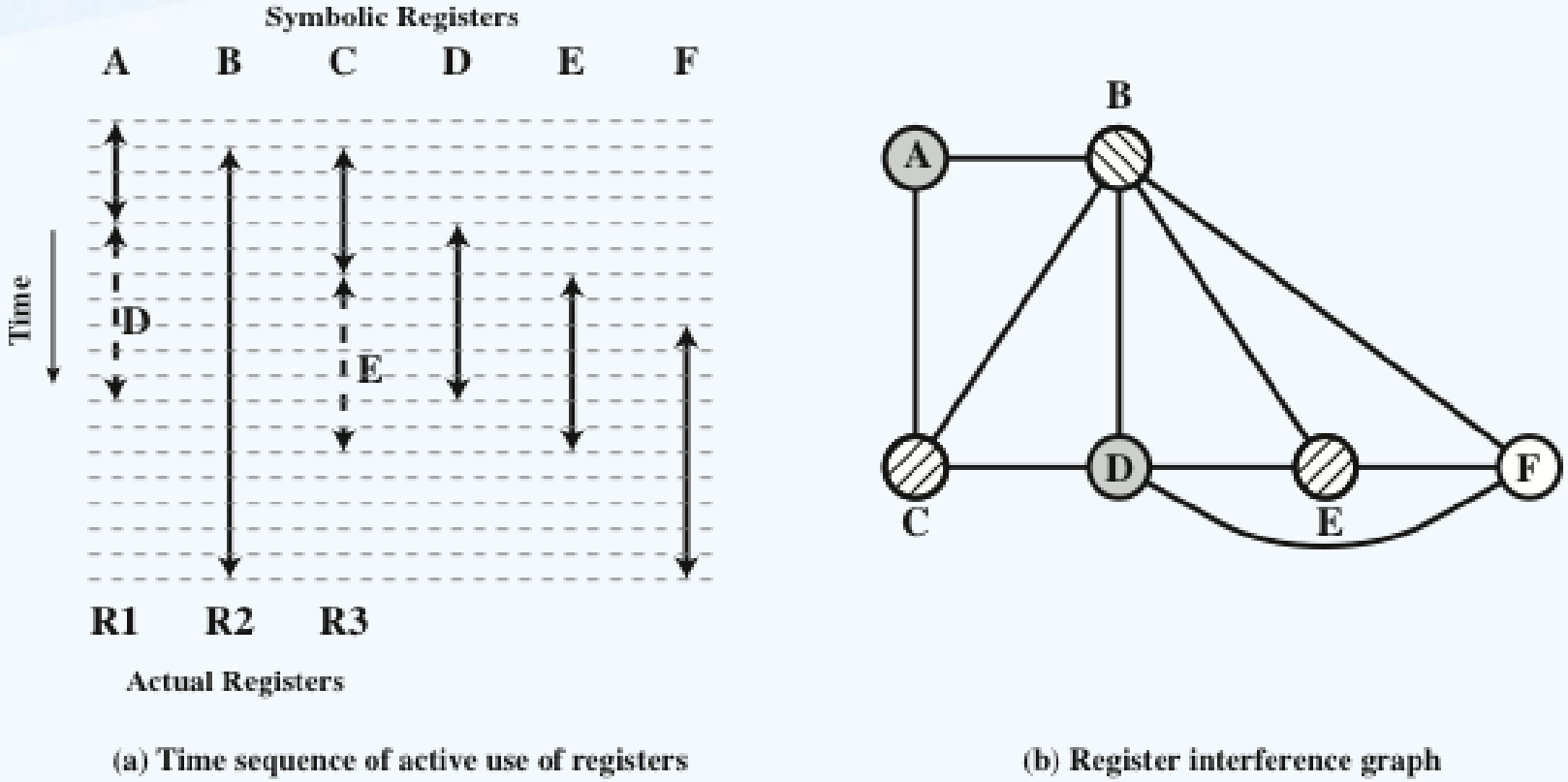
- Figure 15.3 illustrates the difference. To reference a local scalar in a window-based register file, a "virtual" register number and a window number are used. These can pass through a relatively simple decoder to select one of the physical registers. To reference a memory location in cache, a full-width 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.

Referencing a Scalar

- 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.

Graph Coloring Approach



(a) Time sequence of active use of registers

(b) Register interference graph

Figure 15.4 Graph Coloring Approach

Graph Coloring Approach

- Figure 15.4 is a simple example of the process. Assume a program with six symbolic registers to be compiled into three actual registers. Figure 15.4a shows the time sequence of active use of each symbolic register. The dashed horizontal lines indicate successive instruction executions. Figure 15.4b shows the register interference graph (shading and cross-hatching are used instead of colors). A possible coloring with three colors is indicated. Because symbolic registers A and D do not interfere, the compile 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. (Continued)

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.

Why CISC ?

(Complex Instruction Set Computer)

- There is a trend to richer instruction sets which include a larger and more complex number of instructions
- Two principal reasons for this trend:
  - A desire to simplify compilers
  - A desire to improve performance
- There are two advantages to smaller programs:
  - The program takes up less memory
  - Should improve performance
    - Fewer instructions means fewer instruction bytes to be fetched
    - In a paging environment smaller programs occupy fewer pages, reducing page faults
    - More instructions fit in cache(s)

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.

Why CISC ?

- 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], [RAD183], [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.

Why CISC ?

- There are two advantages to smaller programs. First, because the program takes up less memory, there is a savings in that resource. With memory today being so inexpensive, this potential advantage is no longer compelling. More important, 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).

Table 15.6
Code Size Relative to RISC 1

	[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	

Table 15.6 Code Size Relative to RISC I

Table 15.6
Code Size Relative to RISC 1

- 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 15.6 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 [RAD183], who found that the IBM 801 (a RISC) produced code that was 0.9 times the size of code on an IBM S/370. The study used a set of PL/I programs.

- There are several reasons for these rather surprising results. We have already noted that compilers on CISCs tend to favor simpler instructions, so that the conciseness of the complex instructions seldom comes into play. Also, 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.

Table 15.6
Code Size Relative to RISC 1

- 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 [RAD183]. 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

One machine instruction per machine cycle	• *Machine cycle* --- the time it takes to fetch two operands from registers, perform an ALU operation, and store the result in a register
Register-to-register operations	• Only simple LOAD and STORE operations accessing memory • This simplifies the instruction set and therefore the control unit
Simple addressing modes	• Simplifies the instruction set and the control unit
Simple instruction formats	• Generally only one or a few formats are used • Instruction length is fixed and aligned on word boundaries • Opcode decoding and register operand accessing can occur simultaneously

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.

Characteristics of Reduced Instruction Set Architectures

- 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.
- 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.

Characteristics of Reduced Instruction Set Architectures

- 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.

Comparison of Register-to-Register and Memory-to-Memory Approaches

8	16	16	16
Add	B	C	A

Memory to memory

I = 56, D= 96, M = 152

8	4	16
Load	RB	B
Load	RC	B
Add	RA	RB RC
Store	RA	A

Register to memory

I = 104, D = 96, M = 200

(a) A ← B + C

8	16	16	16
Add	B	C	A
Add	A	C	B
Sub	B	D	D

Memory to memory

I = 168, D= 288, M = 456

8	4	4	4
Add	RA	RB	RC
Add	RB	RA	RC
Sub	RD	RD	RB

Register to register

I = 60, D = 0, M = 60

(b) A ← B + C; B ← A + C; D ← D – B

I = number of bytes occupied by executed instructions
D = number of bytes occupied by data
M = total memory traffic = I + D

Figure 15.5 Two Comparisons of Register-to-Register and Memory-to-Memory Approaches

Comparison of Register-to-Register and Memory-to-Memory Approaches

- 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 15.5a 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!

- 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 15.5b may be a fairer comparison.

Table 15.7
Characteristics of Some Processors

Processor	Number of instruc-tion sizes	Max instruc-tion size in bytes	Number of addressing modes	Indirect addressing	Load/store combined with arithmetic	Max number of memory operands	Unaligned addressing allowed	Max Number of MMU uses	Number of bits for integer register specifier	Number of bits 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]

a RISC that does not conform to this characteristic.
b CISC that does not conform to this characteristic.

Table 15.7
Characteristics of Some Processors

- An interesting comparison in [MASH95] provides some insight into this issue. Table 15.7 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).

Table 15.7
Characteristics of Some Processors

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.

- 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.

The Effects of Pipelining

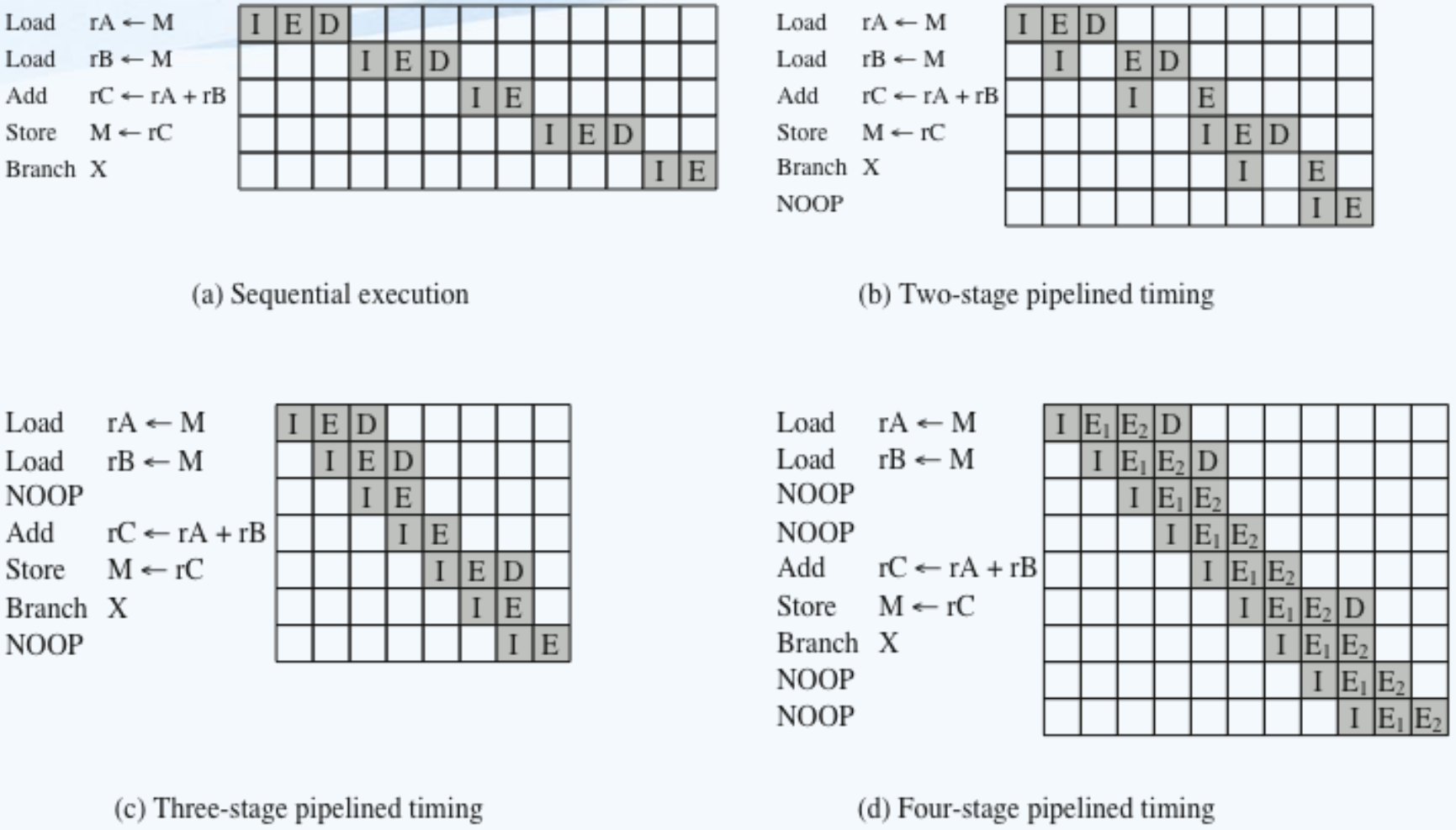


Figure 15.6 The Effects of Pipelining

The Effects of Pipelining

- As we discussed in Section 12.4, 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.

The Effects of Pipelining

- Figure 15.6a 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 15.6b 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 exe- cute/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.

The Effects of Pipelining

- Pipelining can be improved further by permitting two memory accesses per stage. This yields the sequence shown in Figure 15.6c. 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 15.6d 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

- Delayed branch
  - Does not take effect until after execution of following instruction
  - This following instruction is the delay slot
- Delayed Load
  - Register to be target is locked by processor
  - Continue execution of instruction stream until register required
  - Idle until load is complete
  - Re-arranging instructions can allow useful work while loading
- Loop Unrolling
  - Replicate body of loop a number of times
  - Iterate loop fewer times
  - Reduces loop overhead
  - Increases instruction parallelism
  - Improved register, data cache, or TLB locality

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.
- To compensate for these dependencies, code reorganization techniques have been developed.

Table 15.8
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		

Table 15.8 Normal And Delayed Branch

Table 15.8
Normal and Delayed Branch

- 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 15.8. 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.

Use of the Delayed Branch

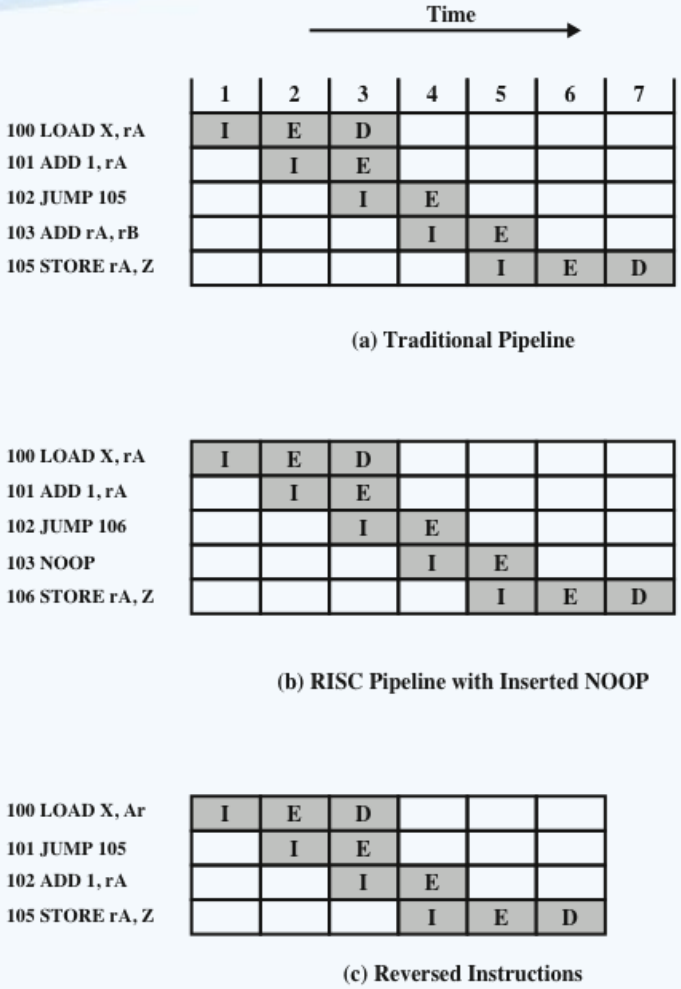


Figure 15.7 Use of the Delayed Branch

Use of the Delayed Branch

- Figure 15.7 shows the result. Figure 15.7a shows the traditional approach to pipelining, of the type discussed in Chapter 14 (e.g., see Figures 14.11 and 14.12). The JUMP instruction is fetched at time 3. At time 4, 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 5, instruction 105, which is the target of the JUMP, is loaded. Figure 15.7b 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 15.7c 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.

Use of the Delayed Branch

- Thus, the original semantics of the program are retained but one less clock cycle is required for execution.
- 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], [RAD183]).
- 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.

Loop Unrolling Twice Example

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

Becomes

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

Loop Unrolling Twice Example

- 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 15.8 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.

MIPS R4000

One of the first commercially available RISC chip sets was developed by MIPS Technology Inc.

Inspired by an experimental system developed at Stanford

Has substantially the same architecture and instruction set of the earlier MIPS designs (R2000 and R3000)

Uses 64 bits for all internal and external data paths and for addresses, registers, and the ALU

Is partitioned into two sections, one containing the CPU and the other containing a coprocessor for memory management

Supports thirty-two 64-bit registers

Provides for up to 128 Kbytes of high-speed cache, half each for instructions and data

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.

MIPS R4000

- 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.

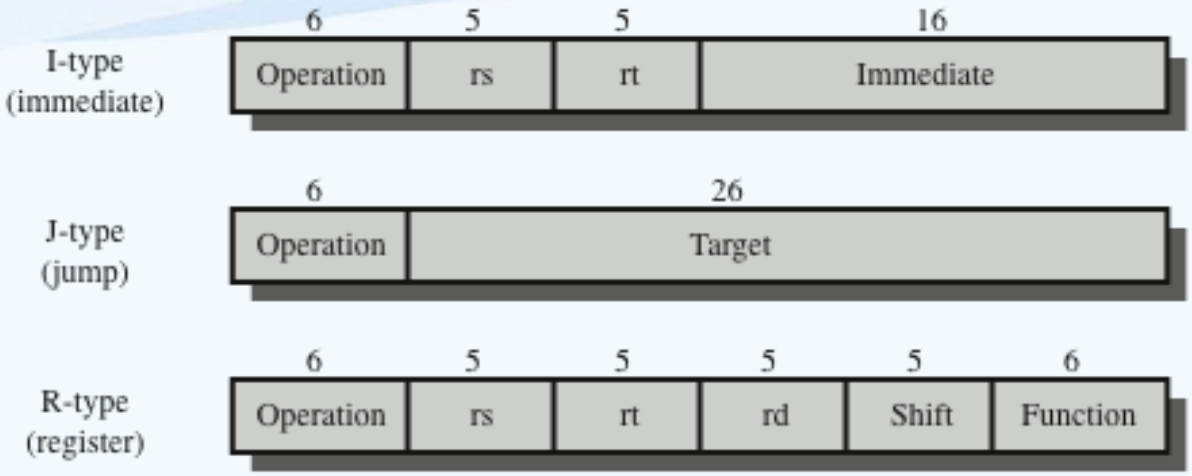
Table 15.9
MIPS R-Series Instruction Set

OP	Description	OP	Description
	Load/Store Instructions		**Multiply/Divide Instructions**
LB	Load Byte	MULT	Multiply
LBU	Load Byte Unsigned	MULTU	Multiply Unsigned
LH	Load Halfword	DIV	Divide
LHU	Load Halfword Unsigned	DIVU	Divide Unsigned
LW	Load Word	MFHI	Move From HI
LWL	Load Word Left	MTHI	Move To HI
LWR	Load Word Right	MFLO	Move From LO
SB	Store Byte	MTLO	Move To LO
SH	Store Halfword		**Jump and Branch Instructions**
SW	Store Word	J	Jump
SWL	Store Word Left	JAL	Jump and Link
SWR	Store Word Right	JR	Jump to Register
	Arithmetic Instructions (ALU Immediate)	JALR	Jump and Link Register
ADDI	Add Immediate	BEQ	Branch on Equal
ADDIU	Add Immediate Unsigned	BNE	Branch on Not Equal
SLTI	Set on Less Than Immediate	BLEZ	Branch on Less Than or Equal to Zero
SLTIU	Set on Less Than Immediate Unsigned	BGTZ	Branch on Greater Than Zero
ANDI	AND Immediate	BLTZ	Branch on Less Than Zero
ORI	OR Immediate	BGEZ	Branch on Greater Than or Equal to Zero
XORI	Exclusive-OR Immediate	BLTZAL	Branch on Less Than Zero And Link
LUI	Load Upper Immediate	BGEZAL	Branch on Greater Than or Equal to Zero And Link
	Arithmetic Instructions (3-operand, R-type)		**Coprocessor Instructions**
ADD	Add	LWCz	Load Word to Coprocessor
ADDU	Add Unsigned	SWCz	Store Word to Coprocessor
SUB	Subtract	MTCz	Move To Coprocessor
SUBU	Subtract Unsigned	MFCz	Move From Coprocessor
SLT	Set on Less Than	CTCz	Move Control To Coprocessor
SLTU	Set on Less Than Unsigned	CFCz	Move Control From Coprocessor
AND	AND	COPz	Coprocessor Operation
OR	OR	BCzT	Branch on Coprocessor z True
XOR	Exclusive-OR	BCzF	Branch on Coprocessor z False
NOR	NOR		**Special Instructions**
	Shift Instructions	SYSCALL	System Call
SLL	Shift Left Logical	BREAK	Break
SRL	Shift Right Logical		
SRA	Shift Right Arithmetic		
SLLV	Shift Left Logical Variable		
SRLV	Shift Right Logical Variable		
SRAV	Shift Right Arithmetic Variable		

Table 15.9
MIPS R-Series Instruction Set

- Table 15.9 lists the basic instruction set for all MIPS R series processors. All processor 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.

MIPS Instruction Formats



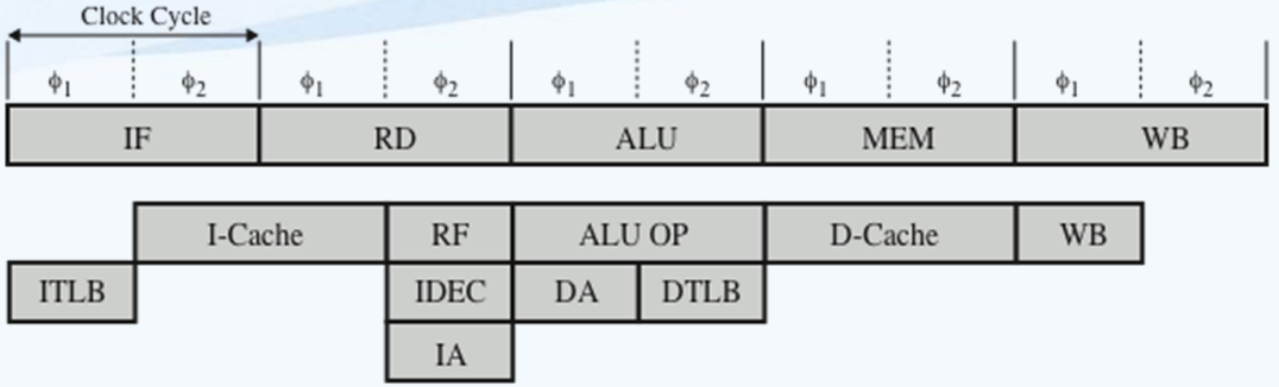
Operation	Operation code
rs	Source register specifier
rt	Source/destination register specifier
Immediate	Immediate, branch, or address displacement
Target	Jump target address
rd	Destination register specifier
Shift	Shift amount
Function	ALU/shift function specifier

Figure 15.9 MIPS Instruction Formats

MIPS Instruction Formats

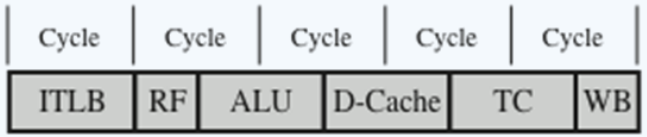
- As with most RISC-based machines, the MIPS uses a single 32-bit instruction length. This single instruction length simplifies instruction fetch and decode, and it also simplifies the interaction of instruction fetch with the virtual memory management unit (i.e., instructions do not cross word or page boundaries). The three instruction formats (Figure 15.9) share common formatting of opcodes and register references, simplifying instruction decode. The effect of more complex instructions can be synthesized at compile time.

Enhancing the R3000 Pipeline



(a) Detailed R3000 pipeline

(b) Modified R3000 pipeline with reduced latencies



(c) Optimized R3000 pipeline with parallel TLB and cache accesses

IF	=	Instruction fetch
RD	=	Read
MEM	=	Memory access
WB	=	Write back to register file
I-Cache	=	Instruction cache access
RF	=	Fetch operand from register
D-Cache	=	Data cache access
ITLB	=	Instruction address translation
IDEC	=	Instruction decode
IA	=	Compute instruction address
DA	=	Calculate data virtual address
DTLB	=	Data address translation
TC	=	Data cache tag check

Figure 15.10 Enhancing the R3000 Pipeline

Enhancing the R3000 Pipeline

- Figure 15.10a 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

Enhancing the R3000 Pipeline

- As illustrated in Figure 15.10a, 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, over- lapped 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.

Enhancing the R3000 Pipeline

- 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 15.10b shows a first step. Remember that the cycles in this figure are half as long as those in Figure 15.10a. 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.

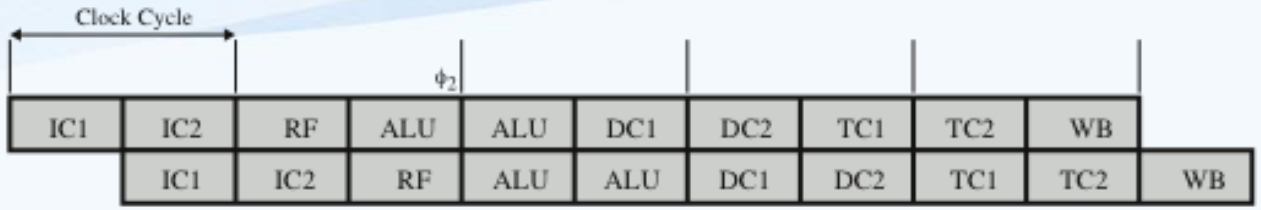
Table 15.10 R3000 Pipeline Stages

The functions performed in each stage are summarized in Table 15.10.

Pipeline Stage	Phase	Function
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.

Table 15.10 R3000 Pipeline Stages

Theoretical R3000 and Actual R4000 Superpipelines



(a) Superpipelined implmentation 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 15.11 Theoretical R3000 and Actual R4000 Superpipelines

Theoretical R3000 and Actual R4000 Superpipelines

- 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 15.11a shows the optimized R3000 pipeline using this superpipelining. Note that this is essentially the same dynamic structure as Figure 15.10c.

- 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 15.11b.

R4000 Pipeline Stages

- 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
  - One of three activities can occur:
    - Instruction is decoded and check made for interlock conditions
    - Instruction cache tag check is made
    - Operands are fetched from the register file
- Tag check
  - Cache tag checks are performed for loads and stores
- Instruction execute
  - One of three activities can occur:
    - If register-to-register operation the ALU performs the operation
    - If a load or store the data virtual address is calculated
    - If branch the branch target virtual address is calculated and branch operations 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
- Write back
  - Instruction result is written back to register file

R4000 Pipeline Stages

- 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.

R4000 Pipeline Stages

- **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.

R4000 Pipeline Stages

- **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.

SPARC

Scalable Processor Architecture

- Architecture defined by Sun Microsystems
- Sun licenses the architecture to other vendors to produce SPARC-compatible machines
- Inspired by the Berkeley RISC 1 machine, and its instruction set and register organization is based closely on the Berkeley RISC model

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 Window Layout With Three Procedures

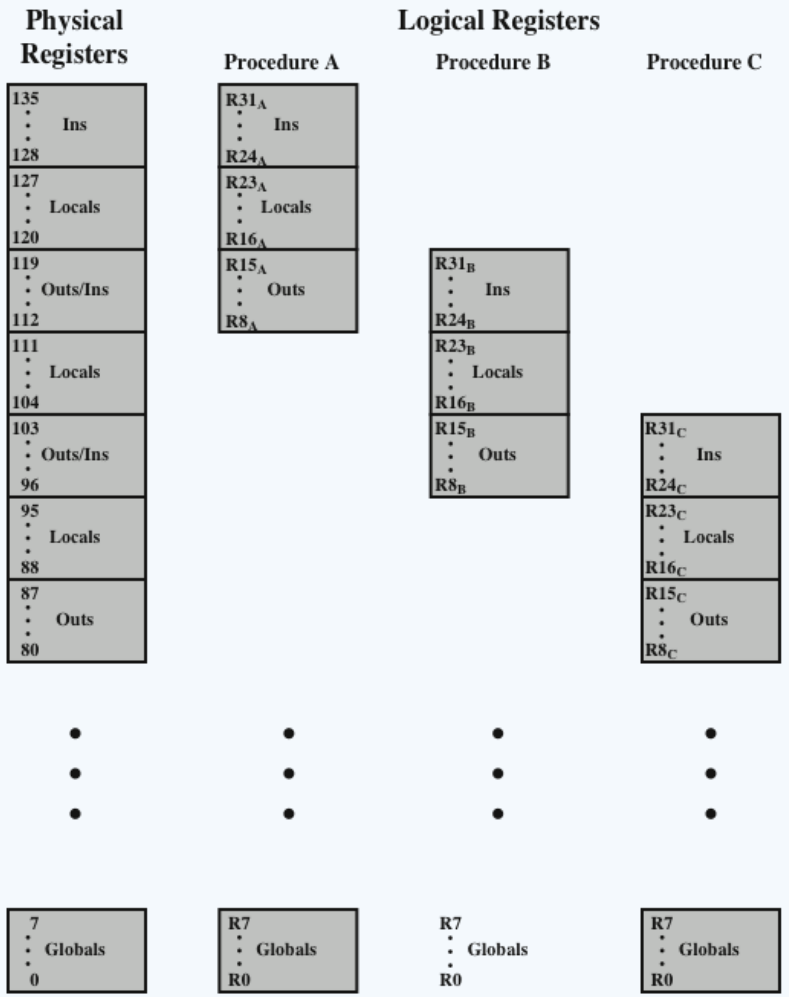


Figure 15.12 SPARC Register Window Layout with Three Procedures

SPARC Register Window Layout With Three Procedures

- 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 15.12 illustrates an implementation that supports 8 windows, using a total of 136 physical registers; as the discussion in Section 15.2 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 12.1 indicates, the availability of 8 registers for parameter passing should be adequate in most cases (e.g., see Table 15.4).

Eight Register Windows Forming a Circular Stack in SPARC

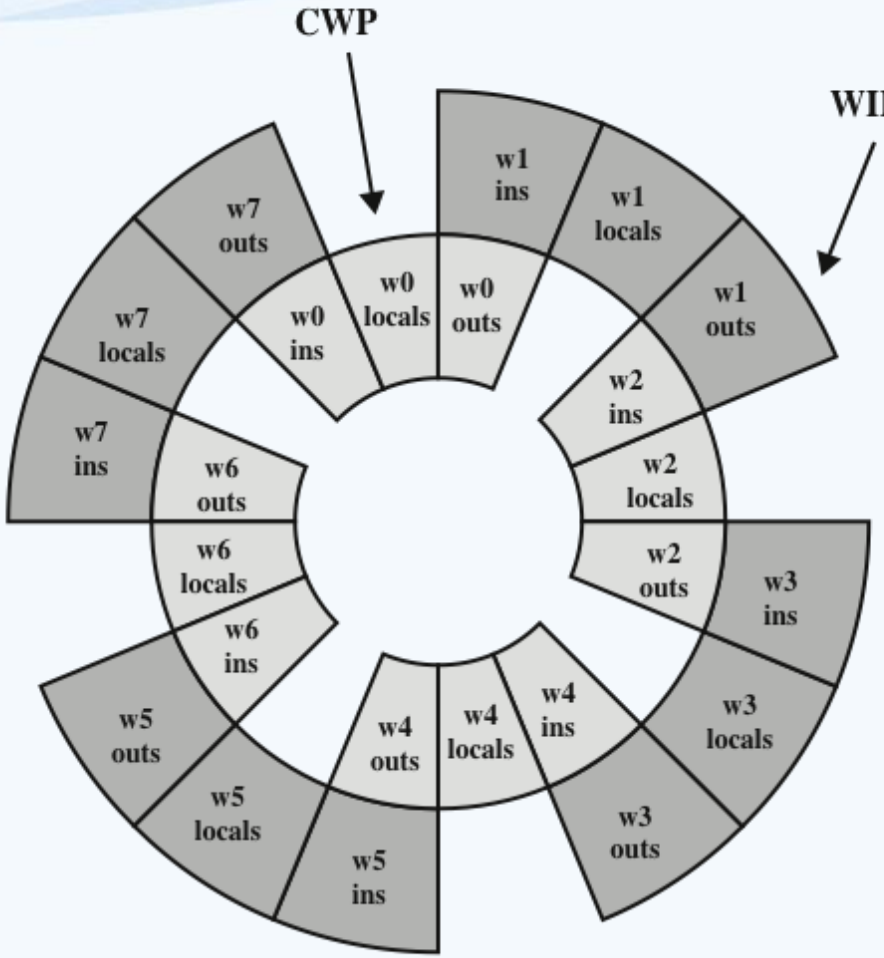


Figure 15.13 Eight Register Windows Forming a Circular Stack in SPARC

Eight Register Windows Forming a Circular Stack in SPARC

- Figure 15.13 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 it *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.

- 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.

Table 15.11 SPARC Instruction Set

Table 15.11 lists the instructions for the SPARC architecture. Most of the instructions reference only register operands.

OP	Description	OP	Description
	Load/Store Instructions		**Arithmetic Instructions**
LDSB	Load signed byte	ADD	Add
LDSH	Load signed halfword	ADDCC	Add, set icc
LDUB	Load unsigned byte	ADDX	Add with carry
LDUH	Load unsigned halfword	ADDXCC	Add with carry, set icc
LD	Load word	SUB	Subtract
LDD	Load doubleword	SUBCC	Subtract, set icc
STB	Store byte	SUBX	Subtract with carry
STH	Store halfword	SUBXCC	Subtract with carry, set icc
STD	Store word	MULSCC	Multiply step, set icc
STDD	Store doubleword		**Jump/Branch Instructions**
	Shift Instructions	BCC	Branch on condition
SLL	Shift left logical	FBCC	Branch on floating-point condition
SRL	Shift right logical	CBCC	Branch on coprocessor condition
SRA	Shift right arithmetic	CALL	Call procedure
	Boolean Instructions	JMPL	Jump and link
AND	AND	TCC	Trap on condition
ANDCC	AND, set icc	SAVE	Advance register window
ANDN	NAND	RESTORE	Move windows backward
ANDNCC	NAND, set icc	RETT	Return from trap
OR	OR		**Miscellaneous Instructions**
ORCC	OR, set icc	SETHI	Set high 22 bits
ORN	NOR	UNIMP	Unimplemented instruction (trap)
ORNCC	NOR, set icc	RD	Read a special register
XOR	XOR	WR	Write a special register
XORCC	XOR, set icc	IFLUSH	Instruction cache flush
XNOR	Exclusive NOR		
XNORCC	Exclusive NOR, set icc		

Table 15.11 SPARC Instruction Set

Table 15.12
Synthesizing Other Addressing Modes with SPARC Addressing Modes

Instruction Type	Addressing Mode	Algorithm	SPARC Equivalent
Register-to-register	Immediate	operand = A	S2
Load, store	Direct	EA = A	$R_0 + S2$
Register-to-register	Register	EA = R	R_{S1}, R_{S2}
Load, store	Register Indirect	EA = (R)	$R_{S1} + 0$
Load, store	Displacement	EA = (R) + A	$R_{S1} + S2$

S2 = either a register operand or a 13-bit immediate operand

SPARC Instruction Formats

- 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 15.12.

SPARC Instruction Formats

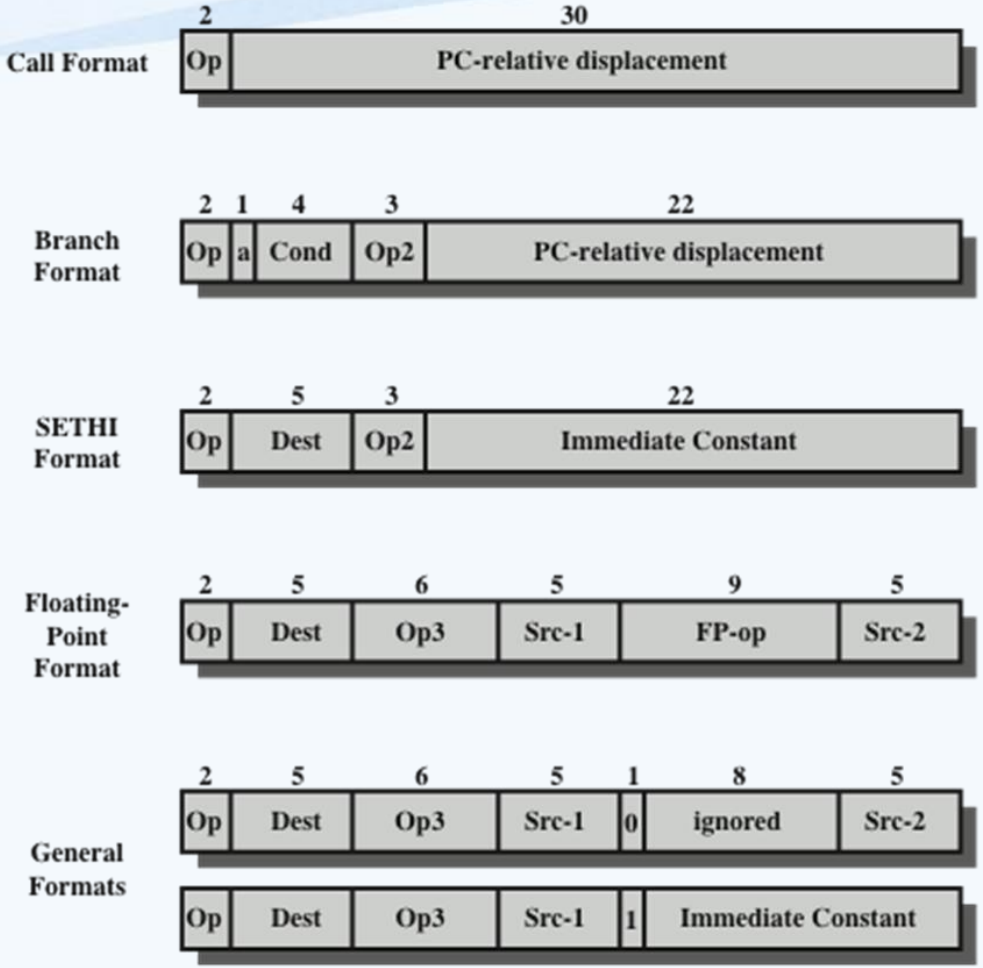


Figure 15.14 SPARC Instruction Formats

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 15.5 (see Figure 15.7). 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.

SPARC Instruction Formats

- 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 r8ld [%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.

SPARC Instruction Formats

- 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 15.14. 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.

RISC versus CISC Controversy

- Quantitative
  - Compare program sizes and execution speeds of programs on RISC and CISC machines that use comparable technology
- Qualitative
  - Examine issues of high level language support and use of VLSI real estate
- Problems with comparisons:
  - No pair of RISC and CISC machines that are comparable in life-cycle cost, level of technology, gate complexity, sophistication of compiler, operating system support, etc.
  - No definitive set of test programs exists
  - Difficult to separate hardware effects from complier effects
  - Most comparisons done on "toy" rather than commercial products
  - Most commercial devices advertised as RISC possess a mixture of RISC and CISC characteristics

RISC versus CISC Controversy

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

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

  - **Quantitative:** Attempts to compare program size and execution speed of programs on RISC and CISC machines that use comparable technology
  - **Qualitative:** Examines issues such as high-level language support and optimum use of VLSI real estate

RISC versus CISC Controversy

- Most of the work on quantitative assessment has been done by those working on RISC systems [PATT82b, HEAT84, PATT84], and it has been, by and large, favorable to the RISC approach. Others have examined the issue and come away unconvinced [COLW85a, FLYN87, DAVI87]. There are several problems with attempting such comparisons [SERL86]:
  - There is no pair of RISC and CISC machines that are comparable in life-cycle cost, level of technology, gate complexity, sophistication of compiler, operating system support, and so on.
  - No definitive test set of programs exists. Performance varies with the program.
  - It is difficult to sort out hardware effects from effects due to skill in compiler writing.

RISC versus CISC Controversy

- Most of the comparative analysis on RISC has been done on "toy" machines rather than commercial products. Furthermore, most commercially available machines advertised as RISC possess a mixture of RISC and CISC characteristics. Thus, a fair comparison with a commercial, "pure-play" CISC machine (e.g., VAX, Pentium) is difficult.
- The qualitative assessment is, almost by definition, subjective. Several researchers have turned their attention to such an assessment [COLW85a, WALL85], but the results are, at best, ambiguous, and certainly subject to rebuttal [PATT85b] and, of course, counterrebuttal [COLW85b].

Summary of Chapter 15 Reduced Instruction Set Computers (RISC)

- Instruction execution characteristics
  - Operations
  - Operands
  - Procedure calls
  - Implications
- The use of a large register file
  - Register windows
  - Global variables
  - Large register file versus cache
- Reduced instruction set architecture
  - Characteristics of RISC
  - CISC versus RISC characteristics
- RISC pipelining
  - Pipelining with regular instructions
  - Optimization of pipelining
- MIPS R4000
  - Instruction set
  - Instruction pipeline
- SPARC
  - SPARC register set
  - Instruction set
  - Instruction format
- Compiler-based register optimization
- RISC versus CISC controversy

Problems Chapter 15 Assignment # 11 Question # 1 to 5 (Due next lecture)

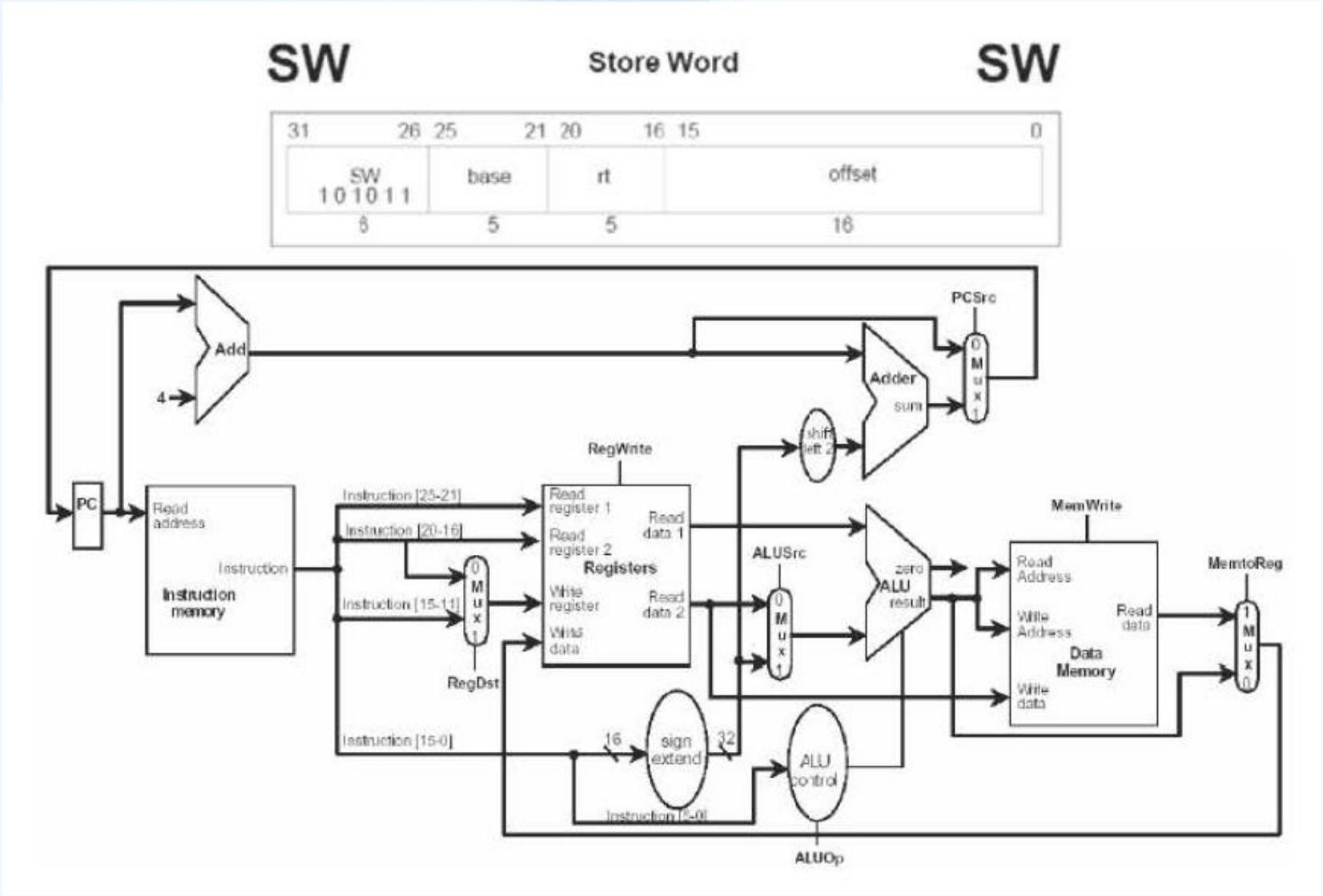
1. What are some typical distinguishing characteristics of RISC organization?
2. Briefly explain the two basic approaches used to minimize register-memory operations on RISC machines.
3. If a circular register buffer is used to handle local variables for nested procedures, describe two approaches for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

Sample Problems Chapter 15 Assignment # 11 Question # 1 to 7 (Due next lecture)

13.1 A processor has a non-linear pipeline with 4 stages A, B, C and D. Each instruction goes through different stages in the following order A B C B A D C. Find the bounds on the maximum instruction throughput in a static hazard free schedule.

13.2 The instruction, sw $18, -12($29), which stores a word (4 bytes) from register 18 into the memory location with an address equal to the contents of register 29 minus the value 12, is executed on the MIPS single cycle datapath. Assume the registers are all pre-loaded with the register number prior to execution and each data memory byte is loaded with the least significant byte of its own address. For example, 0x00000001 contains 0x01, 0x00000002 contains 0x02, 0x0000003F contains 0x0F, etc. Thus, if you lw (load 4 bytes) from 0x00000000, you will get 0x03020100; lw from 0x00000008 will get 0x0B0A0908. Note that little endian is used. Fill in the values of the following buses and control signals in Hex. (You must place an "X" in the blank if the signal is a "don't care".)

Sample Problems Chapter 15 Assignment # 11 Question # 1 to 7 (Due next lecture)



Sample Problems Chapter 15 Assignment # 11 Question # 1 to 7 (Due next lecture)

Instruction[15-0] after sign extension = ?
Read data 1 = ?
Read data 2 = ?
Write Address in Data Memory = ?

Control signals

ALUSrc = ?	MemtoReg = ?
PCSrc = ?	MemWrite = ?
RegDst = ?	RegWrite = ?

13.3 a. Identify all data dependencies in the following code, assuming that we are using the 5-stage MIPS pipelined datapath. Which dependencies can be resolved via forwarding?

```
add $2,$5,$4
add $4,$2,$5
sw $5,100($2)
add $3,$2,$4
```

b. Consider executing the following code on the 5-stage pipelined datapath: Which registers are read during the fifth clock cycle, and which registers are written at the end of the fifth clock cycle? Consider only the registers in the register file (i.e., $1, $2, $3, etc.)

```
add $1,$2,$3
add $4,$5,$6
add $7,$8,$9
add $10,$11,$12
add $13,$14,$15
```

Sample Problems Chapter 15 Assignment # 11 Question # 1 to 7 (Due next lecture)

13.4 You are designing the new Illin 511 processor, a 2-wide in-order pipeline that will run at 2GHz. The pipeline front end (address generation, fetch, decode, and in-order issue) is 14 cycles deep. Branch instructions then take 6 cycles to execute in the back end, load and floating-point instructions take 8 cycles, and integer ALU operations take 4 cycles to execute.

a. Your lab partner has written some excellent assembly code that would be able to achieve a sustained throughput on the Illin 511 of 4 billion instructions/second, as long as no branches mispredict. Assume that an average of 1 out of 10 instructions is a branch, and that branches are correctly predicted at a rate of p. Give an expression for the average sustained throughput in terms of p.

b. Unfortunately, it turns out that there was a bug in the original code. The bug fix made the code somewhat larger, and now you are observing instruction cache misses in the 2Kbyte L1 instruction cache. Each L1 cache miss causes a 10 cycle bubble to be inserted in the instruction stream (while we fetch the cache line from the L2). If you could only double the size of the L1 cache you could completely eliminate the L1 cache misses. Unfortunately building a 4Kbyte fully pipelined2GHz L1 cache would add an extra 5 cycles to the front end. Under what circumstances (e.g. cache miss rate and branch prediction rate) would it be a good or bad idea to change the pipeline to include the larger cache?

Sample Problems Chapter 15 Assignment # 11 Question # 1 to 7 (Due next lecture)

13.5 Briefly give two ways in which loop unrolling can increase performance and one in which it can decrease performance.

13.6 Some RISC architectures require the compiler (or assembly programmer) to guarantee that a register not be accessed for a given number of cycles after it is loaded from memory. Give an advantage and a disadvantage of this design choice.

13.7 Consider the following code:

```
Loop:lw $1, 0($2)
addi $1, $1, 1
sw $1, 0($2)
addi $2, $2, 4
sub $4, $3, $2
bne $4, $0, Loop
```

Assume that the initial value of R3 is R2 + 396

This code snippet will be executed on a MIPS pipelined processor with a 5-stage pipeline. Branches are resolved in the decode stage and do not have delay slots. All memory accesses take 1 clock cycle. In the following three parts, you will be filling out pipeline diagrams for the above code sequence. Please use acronyms F, D, X, M and W for the 5 pipeline stages. Simulate at most 7 instructions, making one pass through the loop and performing the first instruction a second time.

a. Fill in a pipeline diagram in the style of Figure 13.7 for the execution of the above code sequence without any forwarding or bypassing hardware but assuming a register read and a write in the same clock cycle "forwards" through the register file.

b. Fill in a pipeline diagram for the execution of the above code sequence with traditional pipeline forwarding:

Sample Problems Solutions Chapter 15 Assignment # 11 Question # 1 to 7 (Due next lecture)

13.1 First draw the pipeline for this

-	1	2	3	4	5	6	7
A	X				X		
B		X		X			
C			X				X
D						X	

Intervals that cause collisions are
- Row A – 4,
- Row B – 2 ,
- Row C – 4, and
- Row D – none.

Therefore, the initial collision vector is - 001010
No. of 1's in the initial collision vector = 2.
Therefore, minimum average latency ≤ 2+1 = 3
That is, maximum instruction throughput ≥ 1/3 instructions per cycle.
Maximum number of checks in a row of the reservation table = 2
Therefore, minimum average latency ≥ 2
That is, maximum instruction throughput ≤ 1/2 instructions per cycle

Sample Problems Solutions Chapter 15 Assignment # 11 Question # 1 to 7 (Due next lecture)

13.2 Buses

Instruction[15-0] after sign extension = 0x FFFF FFF4 (32-bit)
Read data 1 = 0x 0000 001D (32-bit)
Read data 2 = 0x 0000 0012 (32-bit)
Write Address in Data Memory = 0x 0000 0011 (32-bit)

Control signals

ALUSrc = 1	MemtoReg = X
PCSrc = 0	MemWrite = 1
RegDst = X	RegWrite = 0

13.3 **a.** The second instr. is dependent upon the first (because of $2)
The third instr. is dependent upon the first (because of $2)
The fourth instr. is dependent upon the first (because of $2)
The fourth instr. is dependent upon the second (because of $4)
There, All these dependencies can be solved by forwarding

b. Registers $11 and $12 are read during the fifth clock cycle.
Register $1 is written at the end of fifth clock cycle.

Sample Problems Solutions Chapter 15 Assignment # 11 Question # 1 to 7 (Due next lecture)

13.4 a. Expected instructions to next mispredict is $e = 10/(1 - p)$. Number of instruction slots (two slots/cycle) to execute all those and then get the next good path instruction to the issue unit is $t = e + 2.20 = 10/(1 - p) + 40 = (50 - 40p)/(1 - p)$. So the utilization (rate at which we execute useful instructions) is $e/t = 10/(50 - 40p)$

b. If we increase the cache size the branch mispredict penalty increases from 20 cycles to 25 cycles than, by part (a), utilization would be $10/(60 - 50p)$. If we keep the i-cache small, it will have a non-zero miss-rate, m. We still fetch e instructions between mispredicts, but now of those e , em are 10 cycle (=20 slot) L1 misses. So the number of instruction slots to execute our instructions is

$$t = e(1 + 20m) + 2.20 = [(10)/(1 - p)][1 + 20m] + 40 = (50 + 200m - 40p)/(1 - p)$$

The utilization will be $10/(50 + 200m - 40p)$. The larger cache will provide better utilization when

$(10)/60 - 50p) > (10)/(50 + 200m - 40p)$, or

$50 + 200m - 40p > 60 - 50p$, or

$p > 1 - 20m$

Sample Problems Solutions Chapter 15 Assignment # 11 Question # 1 to 7 (Due next lecture)

13.5 Performance can be increased by:

- Fewer loop conditional evaluations.
- Fewer branches/jumps.
- Opportunities to reorder instructions across iterations.
- Opportunities to merge loads and stores across iterations.

Performance can be decreased by:

- Increased pressure on the I-cache.
- Large branch/jump distances may require slower instructions

13.6 The advantages stem from a simplification of the pipeline logic, which can improve power consumption, area consumption, and maximum speed. The disadvantages are an increase in compiler complexity and a reduced ability to evolve the processor implementation independently of the instruction set (newer generations of chip might have a different pipeline depth and hence a different optimal number of delay cycles).

Sample Problems Solutions Chapter 15 Assignment # 11 Question # 1 to 7 (Due next lecture)

13.7 a.

	Cycle																
Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`lw $1,0($2)`	F	D	X	M	W												
`addi $1,$1,1`		F	D	D	X	M	W										
`sw $1,0($2)`			F	F	D	X	M	W									
`addi $2,$2,4`					F	D	X	M	W								
`sub $4,$3,$2`						F	D	X	M	W							
`bne $4,$0,…`							F	D	D								
`lw $1,0($2)`										F	D	X	M	W			

b.

	Cycle																
Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`lw $1,0($2)`	F	D	X	M	W												
`addi $2,$2,4`		F	D	X	M	W											
`addi $1,$1,1`			F	D	X	M	W										
`sw $1,-4($2)`				F	D	X	M	W									
`sub $4,$3,$2`					F	D	X	M	W								
`bne $4,$0,…`						F	D										
`lw $1,0($2)`								F	D	X	M	W					

We shall InshaAllah continue…

PART FOUR – THE CENTRAL PROCESSING UNIT

Chapter 16 – Instruction-Level Parallelism and Superscalar Processors