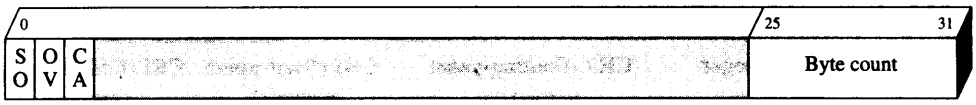


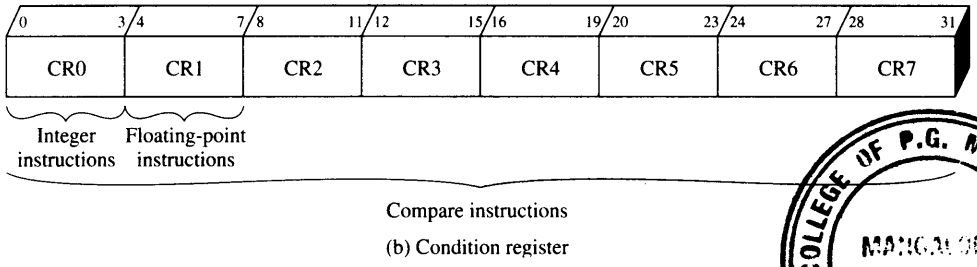
SO = Summary overflow: set to 1 to indicate that an overflow occurred during the exection of an instruction; remains 1 until reset by software

OV = Overflow: set to 1 to indicate that an overflow occurred during the exection of an instruction; reset to 0 by next instruction if there is no overflow

CA = Carry: set to 1 to indicate carry out of bit 0 during the execution of an instruction

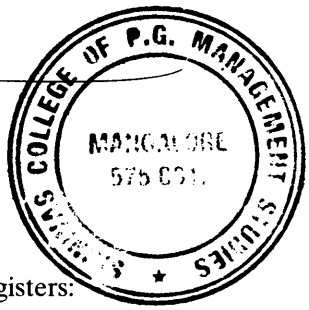
Byte count = Specifies number of bytes to be transferred by Load/Store String indexed instruction

(a) Fixed-point exception register (XER)



(b) Condition register

Figure 12.24 PowerPC Register Formats



The branch processing unit contains these user-visible registers:

- **Condition register:** Consists of eight 4-bit condition code fields (Figure 12.24b).
- **Link register:** The link register can be used in a conditional branch instruction for indirect addressing of the target address. This register is also used for call/return behavior. If the LK bit in a conditional branch instruction is set, then the address following the branch instruction is placed in the link register, and it can be used for a later return.
- **Count:** The count register can be used to control an iteration loop, as explained in Chapter 10; the count register is decremented each time it is tested in a conditional branch instruction. Another use for this register is indirect addressing of the target address in a branch instruction.

The fields of the condition register have a number of uses. The first 4 bits (CR0) are set for all integer arithmetic instructions for which the Rc bit is set. As Table 12.5 shows, the field indicates whether the result of the operation is positive, negative, or zero. The fourth bit is a copy of the summary overflow bit from the XER. The next field (CR1) is set for all floating-point arithmetic instructions for which the Rc bit is set. In this case, the 4 bits are set equal to the first four bits of the FPSCR (Table 12.4). Finally, the eight condition fields (CR0 through CR7) can be used with a compare instruction; in each case, the identity of the field is specified in the instruction itself. For both fixed-point and floating-point compare instructions, the first 3 bits of the designated condition field record whether the first operand is less than, greater than, or equal to the second operand. The fourth

Table 12.5 Interpretation of Bits in Condition Register

Bit position	CR0 (integer instruction with Rc = 1)	CR1 (floating-point instruction with Rc = 1)	CRi (fixed-point compare instruction)	CRi (floating-point compare instruction)
i	result < 0	Exception summary	op1 < op2	op1 < op2
i + 1	result > 0	Enabled exception summary	op1 > op2	op1 > op2
i + 2	result = 0	Invalid operation exception summary	op1 = op2	op1 = op2
i + 3	Summary overflow	Overflow exception	Summary overflow	unordered (one operand is a NaN)

bit is the summary overflow bit for a fixed-point compare, and an unordered indicator for a floating-point compare.

Interrupt Processing

As with any processor, the PowerPC includes a facility that enables the processor to interrupt the currently executing program to deal with an exception condition.

Types of Interrupts Interrupts on a PowerPC are classified as those caused by some system condition or event and those caused by the execution of an instruction. Table 12.6 lists the interrupts recognized by the PowerPC.

Most of the interrupts listed in the table are easily understood. A few warrant further comment. The system reset interrupt happens at power on and when the reset button on the system unit is pressed, and it causes the system to reboot. The machine check interrupt deals with certain anomalies, such as cache parity error and reference to a nonexistent memory location, and may cause the system to enter what is known as a checkstop state; this state suspends processor execution and freezes the contents of registers until a reboot. The floating-point assist enables the processor to invoke software routines to complete operations that cannot be handled directly by the floating-point unit, such as those involving denormalized numbers or unimplemented floating-point opcodes.

Machine State Register Fundamental to the interruption of a program is the ability to recover the state of the processor at the time of the interrupt. This includes not only the contents of the various registers but also various control conditions relating to execution. These conditions are conveniently summarized in the MSR (Table 12.7). Again, several of the bits in this register warrant further comment.

When the privilege mode bit (bit 49) is set, the processor is operating at a user privilege level. Only a subset of the instruction set is available. When the bit is cleared, the processor operates at supervisor privilege level. This enables all of the instructions and provides access to certain system registers (such as the MSR) not accessible from the user privilege level.

Table 12.6 PowerPC Interrupt Table

Entry Point	Interrupt Type	Description
00000h	Reserved	
00100h	System reset	Assertion of the processor's hard or soft reset input signals by external logic
00200h	Machine check	Assertion of TEA# to the processor when it is enabled to recognize machine checks
00300h	Data storage	Examples: data page fault; access rights violation on load/store
00400h	Instruction storage	Code page fault; attempted instruction fetch from I/O segment; access rights violation
00500h	External	Assertion of the processor's external interrupt input signal by external logic when external interrupt recognition is enabled.
00600h	Alignment	Unsuccessful attempt to access memory due to misaligned operand
00700h	Program	Floating-point interrupt; user attempts to execute privileged instruction; trap instruction executed with specified condition met; illegal instruction
00800h	Floating-point unavailable	Attempt to execute floating-point instruction with floating-point unit disabled
00900h	Decrementer	Exhaustion of the decrementer register when external interrupt recognition is enabled
00A00h	Reserved	
00B00h	Reserved	
00C00h	System call	Execution of a system call instruction
00D00h	Trace	Single-step or branch trace interrupt
00E00h	Floating-point assist	Attempt to execute relatively infrequent, complex floating-point operation (e.g., operation on denormalized number)
00E10h through 00FFFh	Reserved	
01000h through 02FFFh	Reserved (implementation specific)	

Unshaded: interrupts caused by instruction execution
Shaded: interrupts not caused by instruction execution

The values of the two floating-point exception bits (bits 52 and 55) define the types of interrupts that the floating-point unit may generate. The interpretation is as follows:

FE0	FE1	Interrupts that will be recognized
0	0	None
0	1	Imprecise nonrecoverable
1	0	Imprecise recoverable
1	1	Precise

Table 12.7 PowerPC Machine State Register

Bit	Definition
0	Processor is in 32-bit/64-bit mode
1:44	Reserved
45	Power management enabled/disabled
46	Implementation dependent
47	Defines whether interrupt handlers run in big-endian or little-endian mode
48	External interrupt enabled/disabled
49	Privileged/nonprivileged state
50	Floating-point unit available/unavailable
51	Machine check interrupts enabled/disabled
52	Floating-point exception mode 0
53	Single-step trace enabled/disabled
54	Branch trace enabled/disabled
55	Floating-point exception mode 1
56	Reserved
57	Most significant part of exception address is 000h/FFFh
58	Instruction address translation on/off
59	Data address translation on/off
60:61	Reserved
62	Interrupt is recoverable/nonrecoverable
63	Processor is in Big-Endian/Little-Endian mode

Unshaded: copied to SRR1
Shaded: not copied to SRR1

When the single-step trace bit (bit 53) is set, the processor branches to the trace interrupt handler after the successful completion of each instruction. When the branch trace bit (bit 54) is set, the processor branches to the branch trace interrupt handler after the successful completion of each branch instruction, whether or not the branch was taken.

The instruction address translation (bit 58) and data address translation (bit 59) determine whether real addressing is used or whether the memory-management unit performs address translation.

Interrupt Handling When an interrupt occurs and is recognized by the processor, the following sequence of events takes place:

1. The processor places the address of the instruction to be executed next in the Save/Restore Register 0 (SRR0). This is the address of the currently executing instruction if the interrupt was caused by a failed attempt to execute that instruction; otherwise, it is the address of the next instruction to be executed after the current instruction.
2. The processor copies machine state information from the MSR to the Save/Restore Register 1 (SRR1). The bits that are depicted as unshaded in

Table 12.7 are copied. The remaining bits of SRR1 are loaded with information specific to the interrupt type.

3. The MSR is set to a hardware-defined value specific to the interrupt type. For all interrupt types, address translation is turned off and external interrupts are disabled.
4. The processor then transfers control to the appropriate interrupt handler. The addresses of the interrupt handlers are stored in the Interrupt Table (Table 12.6). The base address of that table is determined by bit 57 of the MSR.

To return from an interrupt, the interrupt service routine executes an rfi (return from interrupt) instruction. This causes the bit values saved in SRR1 to be restored to the MSR. Execution resumes at the location stored in SRR0.

12.7 RECOMMENDED READING

[PATT01] and [MOSH01] provide excellent coverage of the pipelining issues discussed in this chapter. [HENN91] contains a detailed discussions of pipelining. [SOHI90] provides an excellent, detailed discussion of the hardware design issues involved in an instruction pipeline.

[EVER01] examines the evolution of branch prediction strategies. [CRAG92] is a detailed study of branch prediction in instruction pipelines. [DUBE91] and [LILJ88] examine various branch prediction strategies that can be used to enhance the performance of instruction pipelining. [KAEL91] examines the difficulty introduced into branch prediction by instructions whose target address is variable.

[BREY03] provides good coverage of interrupt processing on the Pentium, as does [SHAN95] for the PowerPC.

BREY03 Brey, B. *The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Processor, Pentium II, Pentium III, Pentium 4.* Upper Saddle River, NJ: Prentice Hall, 2003.

CRAG92 Cragon, H. *Branch Strategy Taxonomy and Performance Models.* Los Alamitos, CA: IEEE Computer Society Press, 1992.

DUBE91 Dubey, P., and Flynn, M. "Branch Strategies: Modeling and Optimization." *IEEE Transactions on Computers,* October 1991.

EVER01 Evers, M., and Yeh, T. "Understanding Branches and Designing Branch Predictors for High-Performance Microprocessors." *Proceedings of the IEEE,* November 2001.

HENN91 Hennessy, J., and Jouppi, N. "Computer Technology and Architecture: An Evolving Interaction." *Computer,* September 1991.

KAEL91 Kaeli, D., and Emma, P. "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns." *Proceedings, 18th Annual International Symposium on Computer Architecture,* May 1991.

LILJ88 Lilja, D. "Reducing the Branch Penalty in Pipelined Processors." *Computer,* July 1988.

MOSH01 Moshovos, A., and Sohi, G. "Microarchitectural Innovations: Boosting Microprocessor Performance Beyond Semiconductor Technology Scaling." *Proceedings of the IEEE,* November 2001.

PATT01 Patt, Y. "Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution." *Proceedings of the IEEE*, November 2001.

SHAN95 Shanley, T. *PowerPC System Architecture.* Reading, MA: Addison-Wesley, 1995.

SOHI90 Sohi, G. "Instruction Issue Logic for High-Performance Interruptable, Multiple Functional Unit, Pipelined Computers." *IEEE Transactions on Computers*, March 1990.

12.8 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

branch prediction	**flag**	**instruction prefetch**
condition code	instruction cycle	program status word (PSW)
delayed branch	**instruction pipeline**	

Review Questions

12.1 What general roles are performed by processor registers?

12.2 What categories of data are commonly supported by user-visible registers?

12.3 What is the function of condition codes?

12.4 What is a program status word?

12.5 Why is a two-stage instruction pipeline unlikely to cut the instruction cycle time in half, compared with the use of no pipeline?

12.6 List and briefly explain various ways in which an instruction pipeline can deal with conditional branch instructions.

12.7 How are history bits used for branch prediction?

Problems

12.1 **a.** If the last operation performed on a computer with an 8-bit word was an addition in which the two operands were 00000010 and 00000011, what would be the value of the following flags?
- Carry
- Zero
- Overflow
- Sign
- Even parity
- Half-carry

b. Repeat for the addition of −1 (twos complement) and +1?

12.2 Repeat Problem 12.1 for the operation A − B, where A contains 11110000 and B contains 0010100.

12.3 A microprocessor is clocked at a rate of 5 GHz.
a. How long is a clock cycle?
b. What is the duration of a particular type of machine instruction consisting of three clock cycles?

12.4 A microprocessor provides an instruction capable of moving a string of bytes from one area of memory to another. The fetching and initial decoding of the instruction takes 10 clock cycles. Thereafter, it takes 15 clock cycles to transfer each byte. The microprocessor is clocked at a rate of 5 GHz.
   a. Determine the length of the instruction cycle for the case of a string of 64 bytes.
   b. What is the worst-case delay for acknowledging an interrupt if the instruction is noninterruptible?
   c. Repeat part (b) assuming the instruction can be interrupted at the beginning of each byte transfer.

12.5 The Intel 8088 consists of a bus interface unit (BIU) and an execution unit (EU), which form a 2-stage pipeline. The BIU fetches instructions into a 4-byte instruction queue. The BIU also participates in address calculations, fetches operands, and writes results in memory as requested by the EU. If no such requests are outstanding and the bus is free, the BIU fills any vacancies in the instruction queue. When the EU completes execution of an instruction, it passes any results to the BIU (destined for memory or I/O) and proceeds to the next instruction.
   a. Suppose the tasks performed by the BIU and EU take about equal time. By what factor does pipelining improve the performance of the 8088? Ignore the effect of branch instructions.
   b. Repeat the calculation assuming that the EU takes twice as long as the BIU.

12.6 Assume an 8088 is executing a program in which the probability of a program jump is 0.1. For simplicity, assume that all instructions are 2 bytes long.
   a. What fraction of instruction fetch bus cycles is wasted?
   b. Repeat if the instruction queue is 8 bytes long.

12.7 Consider the timing diagram of Figure 12.10. Assume that there is only a two-stage pipeline (fetch, execute). Redraw the diagram to show how many time units are now needed for four instructions.

12.8 Assume a pipeline with 4 stages: fetch instruction (FI), decode instruction and calculate addresses (DA), fetch operand (FO), and execute (EX). Draw a diagram similar to Figure 12.10 for a sequence of 7 instructions, in which the third instruction is a branch that is taken and in which there are no data dependencies.

12.9 A pipelined processor has a clock rate of 2.5 GHz and executes a program with 1.5 million instructions. The pipeline has five stages, and instructions are issued at a rate of one per clock cycle. Ignore penalties due to branch instructions and out-of-sequence executions.
   a. What is the speedup of this processor for this program compared to a nonpipelined processor, making the same assumptions used in Section 12.4?
   b. What is throughput (in MIPS) of the pipelined processor?

12.10 A nonpipelined processor has a clock rate of 2.5 GHz and an average CPI (cycles per instruction) of 4. An upgrade to the processor introduces a five-stage pipeline. However, due to internal pipeline delays, such as latch delay, the clock rate of the new processor has to be reduced to 2 GHz.
   a. What is the speedup achieved for a typical program?
   b. What is the MIPS rate for each processor?

12.11 Consider an instruction sequence of length n that is streaming through the instruction pipeline. Let p be the probability of encountering a conditional or unconditional branch instruction, and let q be the probability that execution of a branch instruction I causes a jump to a nonconsecutive address. Assume that each such jump requires the pipeline to be cleared, destroying all ongoing instruction processing, when I emerges from the last stage. Revise Equations (12.1) and (12.2) to take these probabilities into account.

12.12 One limitation of the multiple-stream approach to dealing with branches in a pipeline is that additional branches will be encountered before the first branch is resolved. Suggest two additional limitations or drawbacks.

12.13 Consider the state diagrams of Figure 12.25.
   a. Describe the behavior of each.

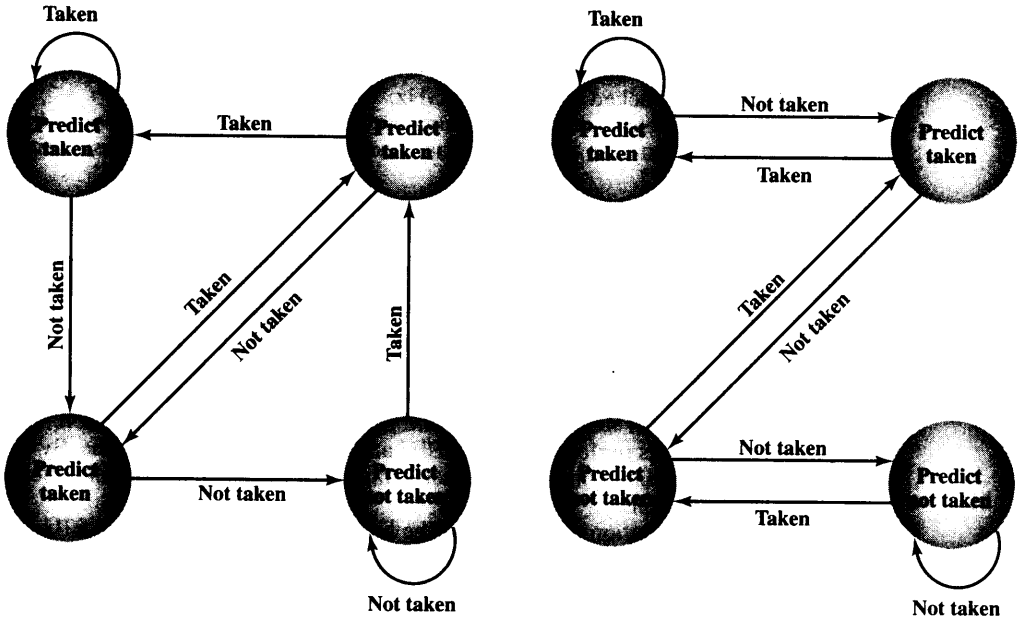


Figure 12.25 State Diagram for Problem 12.5

b. Compare these with the branch prediction state diagram in Section 12.4. Discuss the relative merits of each of the three approaches to branch prediction.

12.14 The Motorola 680x0 machines include the instruction Decrement and Branch According to Condition, which has the following form:

DBcc Dn, displacement

where cc is one of the testable conditions, Dn is a general-purpose register, and displacement specifies the target address relative to the current address. The instruction can be defined as follows:

```
if (cc = False)
then begin
    Dn := (Dn) − 1;
    if Dn ≠ −1 then PC := (PC) + displacement end
else PC := (PC) + 2;
```

When the instruction is executed, the condition is first tested to determine whether the termination condition for the loop is satisfied. If so, no operation is performed and execution continues at the next instruction in sequence. If the condition is false, the specified data register is decremented and checked to see if it is less than zero. If it is less than zero, the loop is terminated and execution continues at the next instruction in sequence. Otherwise, the program branches to the specified location. Now consider the following assembly-language program fragment:

```
AGAIN   CMPM.L    (A0)+,(A1)+
        DBNE      D1,AGAIN
        NOP
```

Two strings addressed by A0 and A1 are compared for equality; the string pointers are incremented with each reference. D1 initially contains the number of longwords (4 bytes) to be compared.

Table 12.8 Branch Behavior in Sample Applications

Occurrence of branch classes:			
Type 1: Branch	72.5%		
Type 2: Loop control	9.8%		
Type 3: Procedure call, return	17.7%		
Type 1 branch: where it goes	**Scientific**	**Commercial**	**Systems**
Unconditional—100% go to target	20%	40%	35%
Conditional—went to target	43.2%	24.3%	32.5%
Conditional—did not go to target (inline)	36.8%	35.7%	32.5%
Type 2 branch (all environments)			
That go to target	91%		
That go inline	9%		
Type 3 branch			
100% go to target			

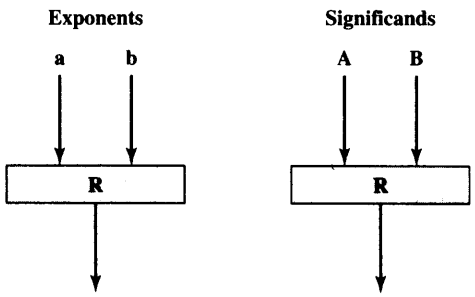
a. The initial contents of the registers are A 0 = $00004000, A 1 = $00005000, and D 1 = $000000FF (the $ indicates hexadecimal notation). Memory between $4000 and $6000 is loaded with words $AAAA. If the foregoing program is run, specify the number of times the DBNE loop is executed and the contents of the three registers when the NOP instruction is reached.

b. Repeat (a), but now assume that memory between $4000 and $4FEE is loaded with $0000 and between $5000 and $6000 is loaded with $AAA.

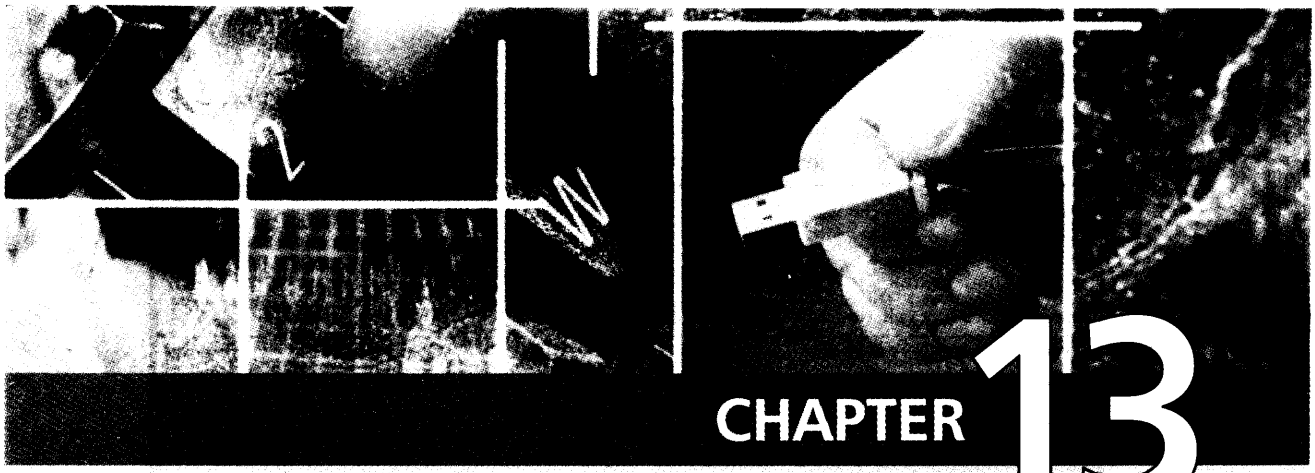
12.15 Redraw Figure 12.19c, assuming that the conditional branch is not taken.

12.16 Table 12.8 summarizes statistics from [MACD84] concerning branch behavior for various classes of applications. With the exception of type 1 branch behavior, there is no noticeable difference among the application classes. Determine the fraction of all branches that go to the branch target address.

12.17 Pipelining can be applied within the ALU to speed up floating-point operations. Consider the case of floating-point addition and subtraction. In simplified terms, the pipeline could have 4 stages: 1. Compare the exponents; 2. Choose the exponent and align the significands; 3. Add or subtract significands; 4. Normalize the results. The pipeline can be considered to have two parallel threads, one handling exponents and one handling significands, and could start out like this:



In this figure, the boxes labeled R refer to a set of registers used to hold temporary results. Complete the block diagram that shows at a top level the structure of the pipeline.



CHAPTER 13

REDUCED INSTRUCTION SET COMPUTERS

KEY POINTS

- Studies of the execution behavior of high-level language programs have provided guidance in designing a new type of processor architecture: the reduced instruction set computer (RISC). Assignment statements predominate, suggesting that the simple movement of data should be optimized. There are also many IF and LOOP instructions, which suggest that the underlying sequence control mechanism needs to be optimized to permit efficient pipelining. Studies of operand reference patterns suggest that it should be possible to enhance performance by keeping a moderate number of operands in registers.
- These studies have motivated the key characteristics of RISC machines: (1) a limited instruction set with a fixed format, (2) a large number of registers or the use of a compiler that optimizes register usage, and (3) an emphasis on optimizing the instruction pipeline.
- The simple instruction set of a RISC lends itself to efficient pipelining because there are fewer and more predictable operations performed per instruction. A RISC instruction set architecture also lends itself to the delayed branch technique, in which branch instructions are rearranged with other instructions to improve pipeline efficiency.

Since the development of the stored-program computer around 1950, there have been remarkably few true innovations in the areas of computer organization and architecture. The following are some of the major advances since the birth of the computer:

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

- **Reduced instruction set computer (RISC) architecture:** This is the focus of this chapter.

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

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

Table 13.1 compares several RISC and non-RISC systems.

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.

13.1 INSTRUCTION EXECUTION CHARACTERISTICS

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

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

Alas, this solution gave rise to another problem, known as the *semantic gap*, the difference between the operations provided in HLLs and those provided in computer architecture. Symptoms of this gap are alleged to include execution inefficiency, excessive 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.

Table 13.1 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

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

To understand the line of reasoning of the RISC advocates, we begin with a brief review of instruction execution characteristics. The aspects of computation of interest are as follows:

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

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

Operations

A variety of studies have been made to analyze the behavior of HLL programs. Table 4.8, discussed in Chapter 4, includes key results from a number of studies. There is quite good agreement in the results of this mixture of languages and applications. Assignment statements predominate, suggesting that the simple movement of data is of high importance. There is also a preponderance of conditional statements (IF, LOOP). These statements are implemented in machine language with some sort of compare and branch instruction. This suggests that the sequence control mechanism of the instruction set is important.

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

To get at this underlying phenomenon, the Patterson programs [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 13.2 show the relative frequency of occurrence of various HLL instructions 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 are dynamic

Table 13.2 Weighted Relative Dynamic Frequency of HLL Operations [PATT82a]

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

frequency statistics. To obtain the data in columns four and five (machine-instruction weighted), each value in the second and third columns is multiplied by the number of machine instructions produced by the compiler. These results are then normalized so that columns four and five show the relative frequency of occurrence, weighted by the number of machine instructions per HLL statement. Similarly, the sixth and seventh columns are obtained by multiplying the frequency of occurrence of each statement type by the relative number of memory references caused by each statement. The data in columns four through seven provide surrogate measures of the actual time spent executing the various statement types. The results suggest that the procedure call/return is the most time-consuming operation in typical HLL programs.

The reader should be clear on the significance of Table 13.2. This table indicates the relative significance of various statement types in an HLL, when that HLL is compiled for a typical contemporary instruction set architecture. Some other architecture could conceivably produce different results. However, this study produces results that are representative for contemporary complex instruction set computer (CISC) architectures. Thus, they can provide guidance to those looking for more efficient ways to support HLLs.

Operands

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

The Patterson study already referenced [PATT82a] also looked at the dynamic frequency of occurrence of classes of variables (Table 13.3). The results, consistent between Pascal and C programs, show that the majority of references are to simple

Table 13.3 Dynamic Percentage of Operands

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

scalar variables. Further, more than 80% of the scalars were local (to the procedure) variables. In addition, references to arrays/structures require a previous reference to their index or pointer, which again is usually a local scalar. Thus, there is a preponderance of references to scalars, and these are highly localized.

The Patterson study examined the dynamic behavior of HLL programs, independent of the underlying architecture. As discussed before, it is necessary to deal with actual architectures to examine program behavior more deeply. One study, [LUND77], examined DEC-10 instructions dynamically and found that each instruction on the average references 0.5 operand in memory and 1.4 registers. Similar results are reported in [HUCK83] for C, Pascal, and FORTRAN programs on S/370, PDP-11, and VAX. Of course, these figures depend highly on both the architecture and the compiler, but they do illustrate the frequency of operand accessing.

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

Procedure Calls

We have seen that procedure calls and returns are an important aspect of HLL programs. The evidence (Table 13.2) 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 13.4. 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.

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.

Table 13.4 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%

Rather, they found that a program remains confined to a rather narrow window of procedure-invocation depth. This is illustrated in Figure 4.16, which was discussed in Chapter 4. These results reinforce the conclusion that operand references are highly localized.

Implications

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

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

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

Finally, a simplified (reduced) instruction set is indicated. This point is not as obvious as the others, but should become clearer in the ensuing discussion.

13.2 THE USE OF A LARGE REGISTER FILE

The results summarized in Section 13.1 point out the desirability of quick access to operands. We have seen that there is a large proportion of assignment statements in HLL programs, and many of these are of the simple form A ← B. Also, there is a significant number of operand accesses per HLL statement. If we couple these results with the fact that most accesses are to local scalars, heavy reliance on register storage is suggested.

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

Two basic approaches are possible, one based on software and the other on hardware. The software approach is to rely on the compiler to maximize register usage. The compiler will attempt to allocate registers to those variables that will

be used the most in a given time period. This approach requires the use of sophisticated program-analysis algorithms. The hardware approach is simply to use more registers so that more variables can be held in registers for longer periods of time.

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

Register Windows

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

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

The solution is based on two other results reported in Section 13.1. First, a typical procedure employs only a few passed parameters and local variables (Table 13.4). Second, the depth of procedure activation fluctuates within a relatively narrow range (Figure 4.16). 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 13.1. 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

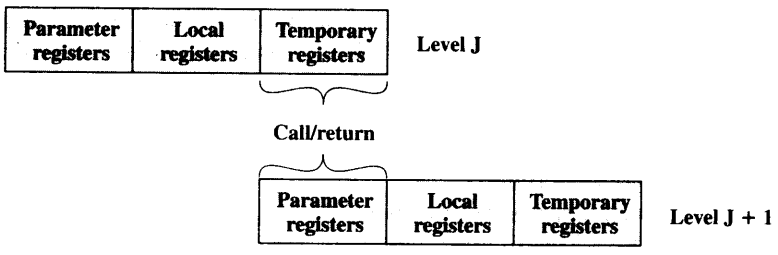


Figure 13.1 Overlapping Register Windows

procedure). The temporary registers at one level are physically the same as the parameter registers at the next lower level. This overlap permits parameters to be passed without the actual movement of data.

To handle any possible pattern of calls and returns, the number of register windows would have to be unbounded. Instead, the register windows can be used to hold the few most recent procedure activations. Older activations must be saved in memory and later restored when the nesting depth decreases. Thus, the actual organization of the register file is as a circular buffer of overlapping windows. Two notable examples of this approach are Sun's SPARC architecture, described in Section 13.7, and the IA-64 architecture used in Intel's Itanium processor, described in Chapter 15.

The circular organization is shown in Figure 13.2, 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 identifies the window most recently saved in memory. If procedure D now calls procedure E, arguments for E are placed in D's temporary registers (the overlap between w3 and w4) and the CWP is advanced by one window.

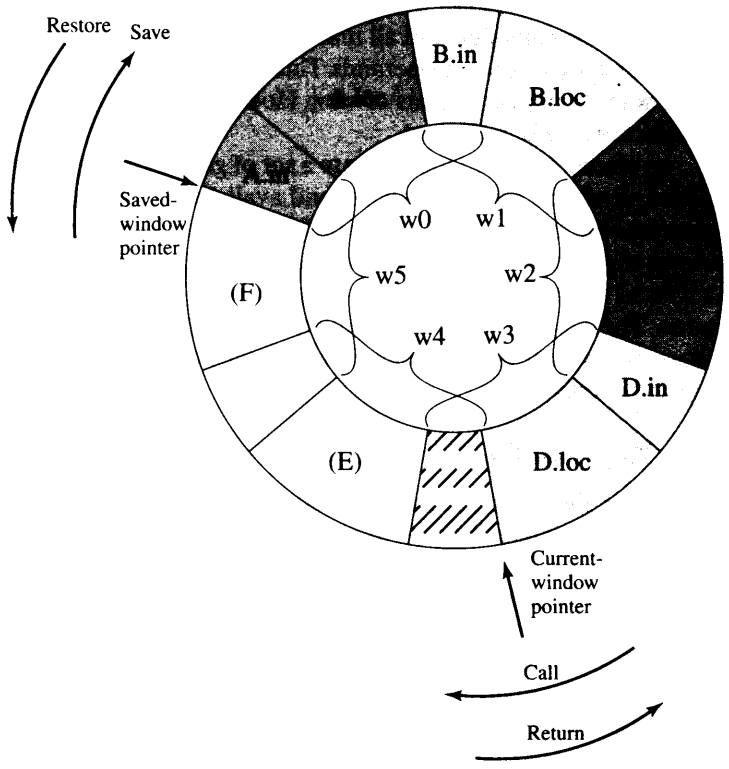


Figure 13.2 Circular-Buffer Organization of Overlapped Windows

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

From the preceding, it can be seen that an N-window register file can hold only $N - 1$ procedure activations. The value of N need not be large. 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

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 compiler must decide which global variables should be assigned to registers.

Large Register File versus Cache

The register file, organized into windows, acts as a small, fast buffer for holding a subset of all variables that are likely to be used 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 13.5 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

Table 13.5 Characteristics of Large-Register-File and Cache Organizations

Large Register File	Cache
All local scalars	Recently-used local scalars
Individual variables	Blocks of memory
Compiler-assigned global variables	Recently-used global variables
Save/Restore based on procedure nesting depth	Save/Restore based on cache replacement algorithm
Register addressing	Memory addressing

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.

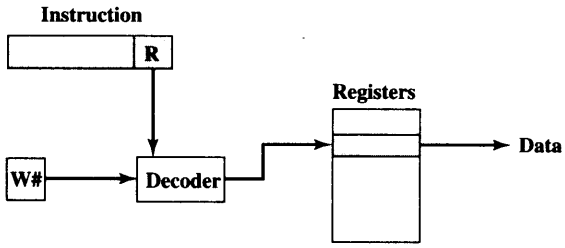
A register file may make inefficient use of space, because not all procedures will need the full window space allotted to them. On the other hand, the cache suffers from another sort of inefficiency: Data are read into the cache in blocks. Whereas the register file contains only those variables in use, 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, it is difficult for a compiler to determine which globals will be heavily used.

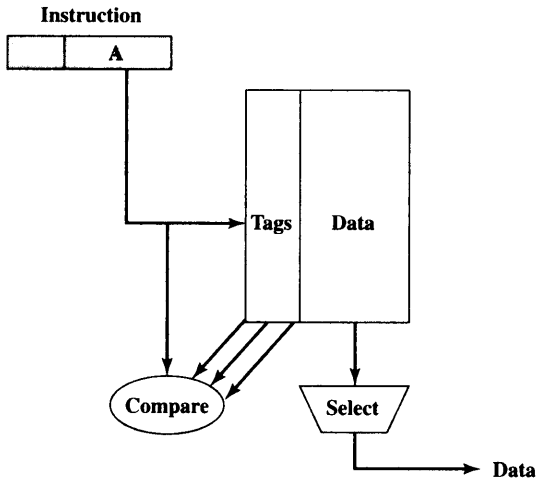
With the register file, the movement of data between registers and memory is determined by the procedure nesting depth. Because this depth usually fluctuates within a narrow range, the use of memory is relatively infrequent. Most cache memories are set associative with a small set size. Thus, there is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

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

Figure 13.3 illustrates the difference. To reference a local scalar in a window-based register file, a "virtual" register number and a window number are used. These can pass through a relatively simple decoder to select one of the physical registers. To reference a memory location in cache, a full-width memory address must be generated. The complexity of this operation depends on the addressing mode. In a set associative cache, a portion of the address is used to read a number of words and tags equal to the set size. Another portion of the address is compared with the tags, and one of the words that were read is selected. It should be clear that even if the cache is as fast as the register file, the access time will be considerably longer. Thus, from the point of view of performance, the window-based register file is superior for local scalars. Further performance improvement could be achieved by the addition of a cache for instructions only.



(a) Windows-based register file



(b) Cache

Figure 13.3 Referencing a Scalar

13.3 COMPILER-BASED REGISTER OPTIMIZATION

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

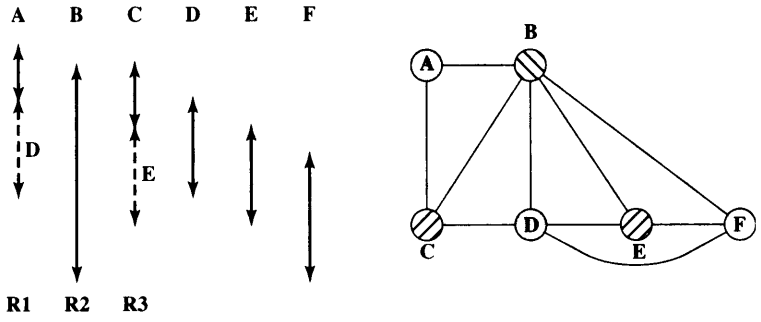
In general, the approach taken is as follows. Each program quantity that is a candidate for residing in a register is assigned to a symbolic or virtual register. The compiler then maps the unlimited number of symbolic registers into a fixed number of real registers. Symbolic registers whose usage does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

The essence of the optimization task is to decide which quantities are to be assigned to registers at any given point in the program. The technique most commonly used in RISC compilers is known as graph coloring, which is a technique borrowed from the discipline of topology [CHAI82, CHOW86, COUT86, CHOW90].

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

Figure 13.4 is a simple example of the process. Assume a program with six symbolic registers to be compiled into three actual registers. Figure 13.4a shows the time sequence of active use of each symbolic register, and part b shows the register interference graph (shading and cross-hatching are used instead of colors). A possible coloring with three colors is indicated. One symbolic register, F, is left uncolored and must be dealt with using loads and stores.

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



(a) Time sequence of active use of registers (b) Register interference graph

Figure 13.4 Graph Coloring Approach

concerned with optimizing the use of a small number of registers rather than comparing the use of large register sets with optimization efforts.

13.4 REDUCED INSTRUCTION SET ARCHITECTURE

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

Why CISC

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

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

The first of the reasons cited, compiler simplification, seems obvious. The task of the compiler writer is to generate a sequence of machine instructions for each HLL statement. If there are machine instructions that resemble HLL statements, this task is simplified. This reasoning has been disputed by the RISC researchers ([HENN82], [RADI83], [PATT82b]). They have found that complex machine instructions are often hard to exploit because the compiler must find those cases that exactly fit the construct. The task of optimizing the generated code to minimize code size, reduce instruction execution count, and enhance pipelining is much more difficult with a complex instruction set. As evidence of this, studies cited earlier in this chapter indicate that most of the instructions in a compiled program are the relatively simple ones.

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

There are two advantages to smaller programs. 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 two ways. First, fewer instructions means fewer instruction bytes to be fetched. Second, in a paging environment, smaller programs occupy fewer pages, reducing page faults.

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

Table 13.6 Code Size Relative to RISC I

	[PATT82a] 11 C Programs	[KATE83] 12 C Programs	[HEAT84] 5 C Programs
RISC I	1.0	1.0	1.0
VAX-11/780	0.8	0.67	
M68000	0.9		0.9
Z8002	1.2		1.12
PDP-11/70	0.9	0.71	

Table 13.6 shows results from three studies that compared the size of compiled C programs on a variety of machines, including RISC I, which has a reduced instruction set architecture. Note that there is little or no savings using a CISC over a RISC. It is also interesting to note that the VAX, which has a much more complex instruction set than the PDP-11, achieves very little savings over the latter. These results were confirmed by IBM researchers [RADI83], who found that the IBM 801 (a RISC) produced code that was 0.9 times the size of code on an IBM S/370. The study used a set of PL/I programs.

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

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

In fact, some researchers have found that the speedup in the execution of complex functions is due not so much to the power of the complex machine instructions as to their residence in high-speed control store [RADI83]. In effect, the control store acts as an instruction cache. Thus, the hardware architect is in the position of trying to determine which subroutines or functions will be used most frequently and assigning those to the control store by implementing them in microcode. The results have been less than encouraging. On S/390 systems, instructions such as Translate and Extended-Precision-Floating-Point-Divide reside in high-speed storage, while the sequence involved in setting up procedure calls or initiating an interrupt handler are in slower main memory.

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

Characteristics of Reduced Instruction Set Architectures

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

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

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

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

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

This emphasis on register-to-register operations is notable for RISC designs. Contemporary CISC machines provide such instructions but also include memory-to-memory and mixed register/memory operations. Attempts to compare these approaches were made in the 1970s, before the appearance of RISCs. Figure 13.5a 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 13.5b may be a fairer comparison.

A third characteristic is the use of **simple addressing modes**. Almost all RISC instructions use simple register addressing. Several additional modes, such as displacement and PC-relative, may be included. Other, more complex modes can be synthesized in software from the simple ones. Again, this design feature simplifies the instruction set and the control unit.

A final common characteristic is the use of **simple instruction formats**. Generally, only one or a few formats are used. Instruction length is fixed and aligned on word

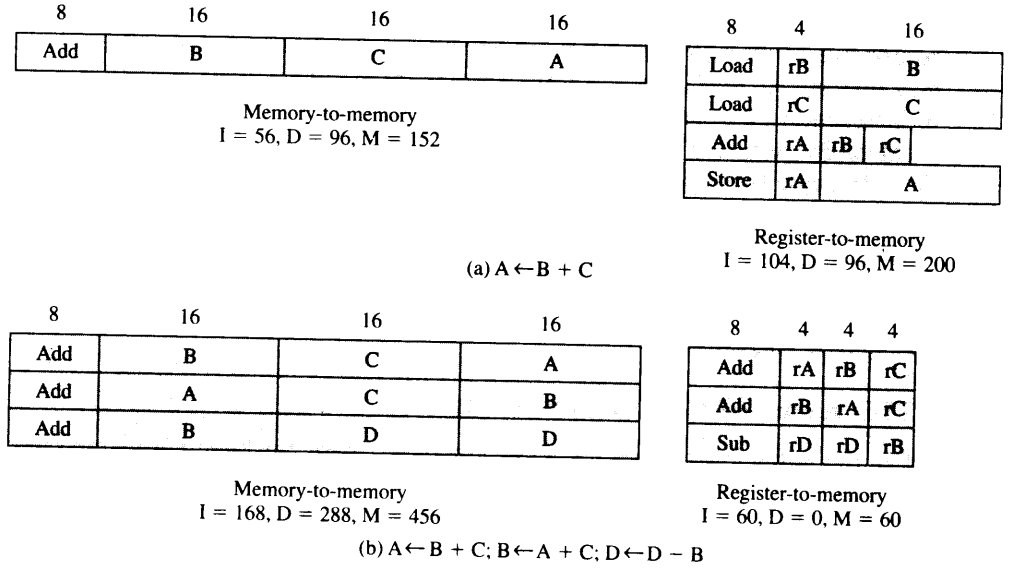


Figure 13.5 Two Comparisons of Register-to-Register and Memory-to-Memory Approaches

boundaries. Field locations, especially the opcode, are fixed. This design feature has a number of benefits. With fixed fields, opcode decoding and register operand accessing can occur simultaneously. Simplified formats simplify the control unit. Instruction fetching is optimized because word-length units are fetched. Alignment on a word boundary also means that a single instruction does not cross page boundaries.

Taken together, these characteristics can be assessed to determine the potential performance benefits of the RISC approach. A certain amount of "circumstantial evidence" can be presented. First, more effective optimizing compilers can be developed. With more-primitive instructions, there are more opportunities for moving functions out of loops, reorganizing code for efficiency, maximizing register utilization, and so forth. It is even possible to compute parts of complex instructions at compile time. For example, the S/390 Move Characters (MVC) instruction moves a string of characters from one location to another. Each time it is executed, the move will depend on the length of the string, whether and in which direction the locations overlap, and what the alignment characteristics are. In most cases, these will all be known at compile time. Thus, the compiler could produce an optimized sequence of primitive instructions for this function.

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

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

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

The case for improved performance for a reduced instruction set architecture is strong, but one could perhaps still make an argument for CISC. A number of studies have been done but not on machines of comparable technology and power. Further, most studies have not attempted to separate the effects of a reduced instruction set and the effects of a large register file. The "circumstantial evidence," however, is suggestive.

CISC versus RISC Characteristics

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

An interesting comparison in [MASH95] provides some insight into this issue. Table 13.7 lists a number of processors and compares them across a number of characteristics. For purposes of this comparison, the following are considered typical of a classic RISC:

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

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

Table 13.7 Characteristics of Some Processors

Processor	Number of instruction sizes	Max instruction 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.

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.

13.5 RISC PIPELINING

Pipelining with Regular Instructions

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.

Figure 13.6a 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 13.6b shows a two-stage pipelining scheme, in which the I and E stages of two different instructions are performed simultaneously. 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

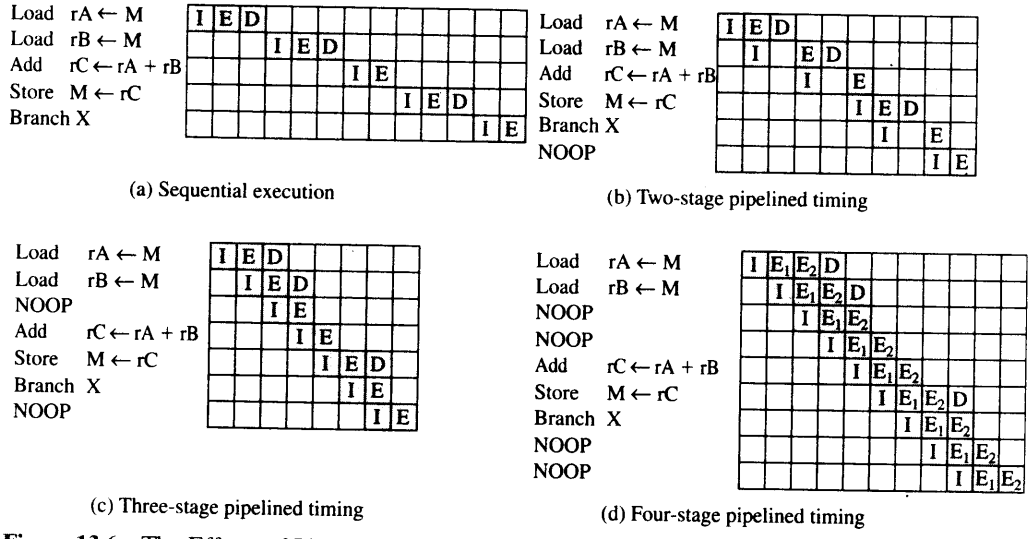


Figure 13.6 The Effects of Pipelining

of a wait state in some instructions. Second, a branch instruction interrupts the sequential flow of execution. To accommodate this with minimum circuitry, a NOOP instruction can be inserted into the instruction stream by the compiler or assembler.

Pipelining can be improved further by permitting two memory accesses per stage. This yields the sequence shown in Figure 13.6c. 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 13.6d shows the result with a four-stage pipeline. Up to four instructions at a time can be under way, and the maximum potential speedup is a factor of 4. Note again the use of NOOPs to account for data and branch delays.

Optimization of Pipelining

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

To compensate for these dependencies, code reorganization techniques have been developed. 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 13.8. 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

Table 13.8 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		

pipeline, a NOOP is inserted after this branch. However, increased performance is achieved if the instructions at 101 and 102 are interchanged.

Figure 13.7 shows the result. Figure 13.7a shows the traditional approach to pipelining, of the type discussed in Chapter 12 (e.g., see Figures 12.11 and 12.12). 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 13.7b 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 13.7c shows the use of the delayed branch. The JUMP instruction is fetched at time 2, before the ADD instruction, which is fetched at time 3. Note, however, that the ADD instruction is fetched before the execution of the JUMP instruction has a chance to alter the program counter. Therefore, during time 4, the ADD instruction is executed at the same time that instruction 105 is fetched. Thus, the original semantics of the program are retained but one less clock cycle is required for execution.

Time →

	1	2	3	4	5	6	7	8
100 LOAD X, rA	I	E	D					
101 ADD 1, rA		I		E				
102 JUMP 105				I	E			
103 ADD rA, rB					I			
105 STORE rA, Z						I	E	D

(a) Traditional pipeline

100 LOAD X, rA	I	E	D				
101 ADD 1, rA		I	E				
102 JUMP 106			I	E			
103 NOOP				I	E		
106 STORE rA, Z					I	E	D

(b) RISC Pipeline with inserted NOOP

100 LOAD X, Ar	I	E	D			
101 JUMP 105		I	E			
102 ADD 1, rA			I	E		
105 STORE rA, Z				I	E	D

(c) Reversed instructions

Figure 13.7 Use of the Delayed Branch