Table 13.11 R3000 Pipeline Stages

Pipeline Stage	Phase	Function
IF	$\phi1$	Using the TLB, translate an instruction virtual address to a physical address (after a branching decision).
IF	$\phi2$	Send the physical address to the instruction address.
RD	$\phi1$	Return instruction from instruction cache.
		Compare tags and validity of fetched instruction.
RD	$\phi2$	Decode instruction.
		Read register file.
		If branch, calculate branch target address.
ALU	$\phi1 + \phi2$	If register-to-register operation, the arithmetic or logical operation is performed.
ALU	$\phi1$	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	$\phi2$	If a memory reference, translate data virtual address to physical using TLB.
MEM	$\phi1$	If a memory reference, send physical address to data cache.
MEM	$\phi2$	If a memory reference, return data from data cache, and check tags.
WB	$\phi1$	Write to register file.

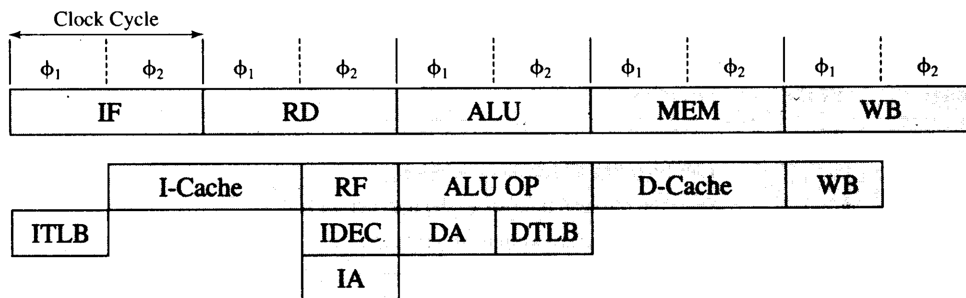
incorporated on the chip. Before looking at the final R4000 pipeline, let us consider how the R3000 pipeline can be modified to improve performance using R4000 technology.

Figure 13.9b shows a first step. Remember that the cycles in this figure are half as long as those in Figure 13.9a. Because they are on the same chip, the instruction and data cache stages take only half as long; so they still occupy only one clock cycle. Again, because of the speedup of the register file access, register read and write still occupy only half of a clock cycle.

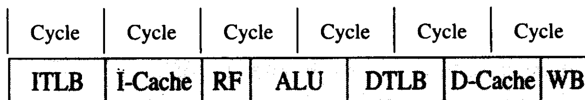
Because the R4000 caches are on-chip, the virtual-to-physical address translation can delay the cache access. This delay is reduced by implementing virtually indexed caches and going to a parallel cache access and address translation. Figure 13.9c shows the optimized R3000 pipeline with this improvement. Because of the compression of events, the data cache tag check is performed separately on the next cycle after cache access.

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 13.10a shows the optimized R3000 pipeline using this superpipelining. Note that this is essentially the same dynamic structure as Figure 13.9c.

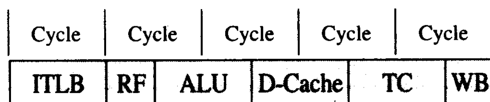
Further improvements can be made. For the R4000, a much larger and specialized adder was designed. This makes it possible to execute ALU operations at twice the rate. Other improvements allow the execution of loads and stores at twice the rate. The resulting pipeline is shown in Figure 13.10b.



(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
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

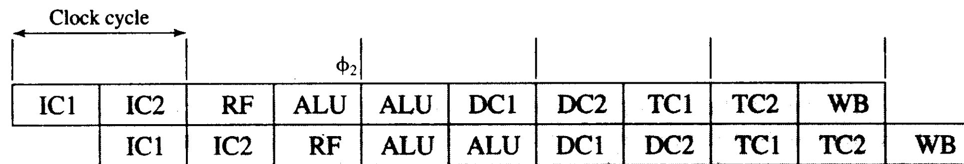
Figure 13.9 Enhancing the R3000 Pipeline

- Data memory reference
- Write back into register file

As illustrated in Figure 13.9a, there is not only parallelism due to pipelining but also parallelism within the execution of a single instruction. The 60-ns clock cycle is divided into two 30-ns stages. The external instruction and data access operations to the cache each require 60 ns, as do the major internal operations (OP, DA, IA). Instruction decode is a simpler operation, requiring only a single 30-ns stage, overlapped with register fetch in the same instruction. Calculation of an address for a branch instruction also overlaps instruction decode and register fetch, so that a branch at instruction i can address the ICACHE access of instruction $i + 2$. Similarly, a load at instruction i fetches data that are immediately used by the OP of instruction $i + 1$, while an ALU/shift result gets passed directly into instruction $i + 1$ with no delay. This tight coupling between instructions makes for a highly efficient pipeline.

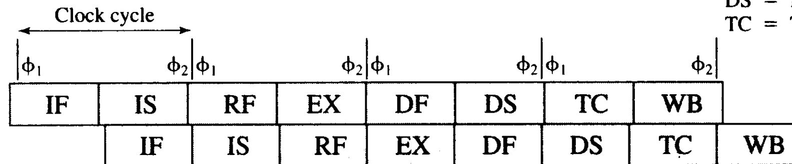
In detail, then, each clock cycle is divided into separate stages, denoted as $\phi 1$ and $\phi 2$. The functions performed in each stage are summarized in Table 13.11.

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



(a) Superpipelined implementation of the optimized R3000 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



(b) R4000 pipeline

Figure 13.10 Theoretical R3000 and Actual R4000 Superpipelines

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

- **Instruction fetch first half:** Virtual address is presented to the instruction cache and the translation lookaside buffer.
- **Instruction fetch second half:** Instruction cache outputs the instruction and the TLB generates the physical address.
- **Register file:** Three activities occur in parallel:
  - Instruction is decoded and check made for interlock conditions (i.e., this instruction depends on the result of a preceding instruction).
  - Instruction cache tag check is made.
  - Operands are fetched from the register file.
- **Instruction execute:** One of three activities can occur:
  - If the instruction is a register-to-register operation, the ALU performs the arithmetic or logical operation.
  - If the instruction is a load or store, the data virtual address is calculated.
  - If the instruction is a branch, the branch target virtual address is calculated and branch conditions are checked.
- **Data cache first:** Virtual address is presented to the data cache and TLB.
- **Data cache second:** Data cache outputs the instruction, and the TLB generates the physical address.
- **Tag check:** Cache tag checks are performed for loads and stores.
- **Write back:** Instruction result is written back to register file.

13.7 SPARC

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

SPARC Register Set

As with the Berkeley RISC, the SPARC makes use of register windows. Each window consists of 24 registers, and the total number of windows is implementation dependent and ranges from 2 to 32 windows. Figure 13.11 illustrates an implementation that

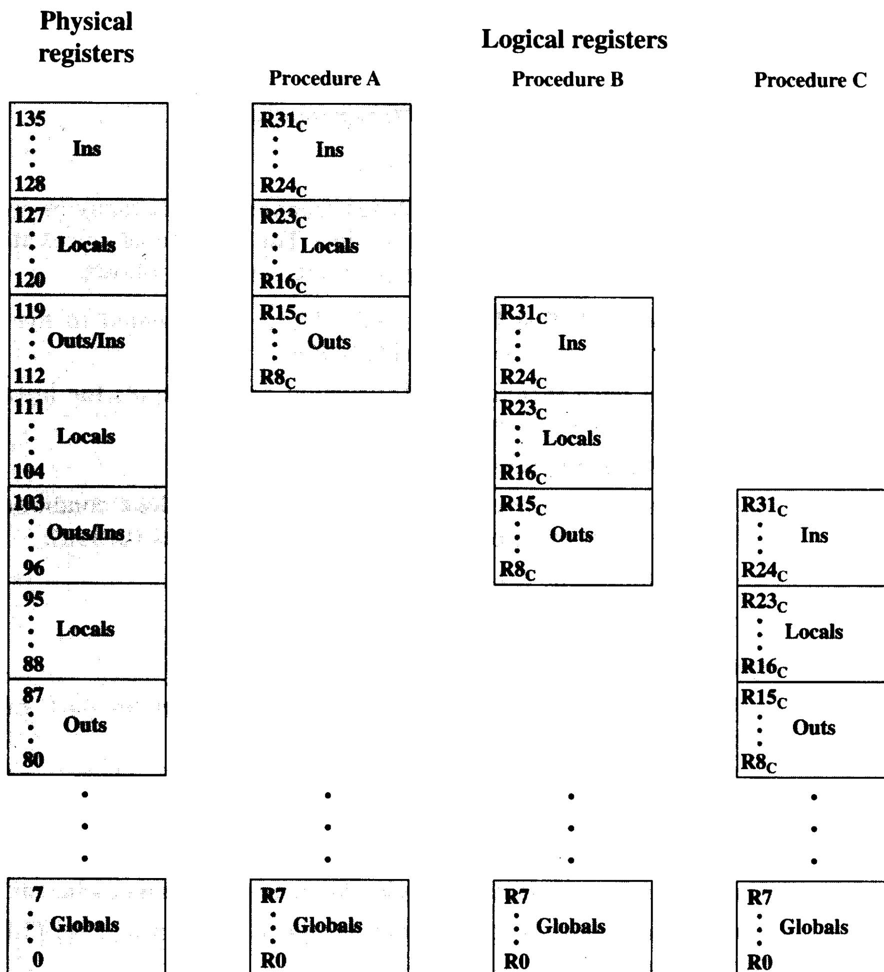


Figure 13.11 SPARC Register Window Layout with Three Procedures

supports 8 windows, using a total of 136 physical registers; as the discussion in Section 13.2 indicates, this seems a reasonable number of windows. Physical registers 0 through 7 are global registers shared by all procedures. Each process 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 13.4).

Figure 13.12 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.

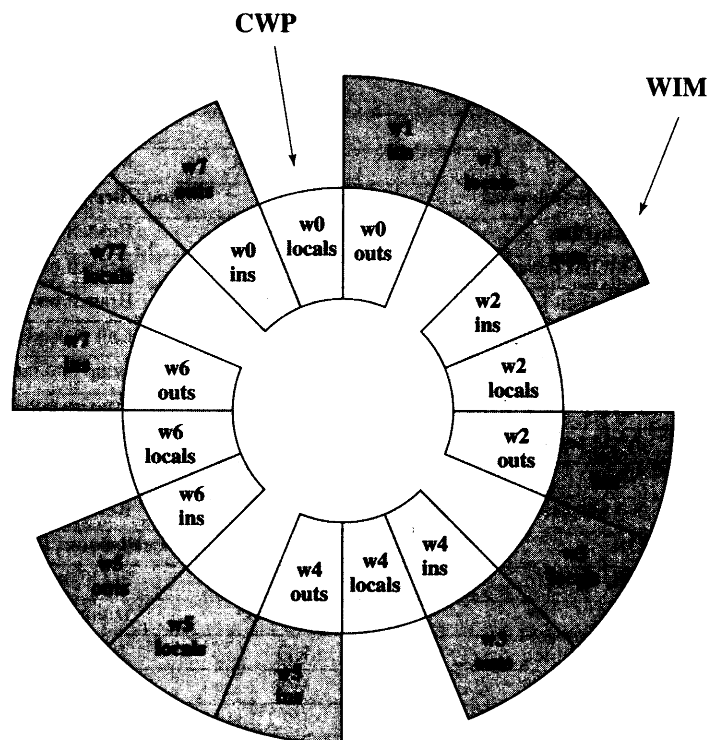


Figure 13.12 Eight Register Windows Forming a Circular Stack in SPARC

Instruction Set

Table 13.12 lists the instructions for the SPARC architecture. Most of the instructions reference only register operands. Register-to-register instructions have three operands and can be expressed in the form

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

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

Table 13.12 SPARC Instruction Set

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 13.13 Synthesizing Other Addressing Modes with SPARC Addressing Modes

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

The available ALU operations can be grouped as follows:

- Integer addition (with or without carry)
- Integer subtraction (with or without carry)
- Bitwise Boolean AND, OR, XOR and their negations
- Shift left logical, right logical, or right arithmetic

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

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

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

$$EA = (R_{S1}) + S2$$
$$\text{or } EA = (R_{S1}) + (R_{S2})$$

depending on whether the second operand is immediate or a register reference. To perform a load or store, an extra stage is added to the instruction cycle. During the second stage, the memory address is calculated using the ALU; the load or store occurs in a third stage. This single addressing mode is quite versatile and can be used to synthesize other addressing modes, as indicated in Table 13.13.

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

Instruction Format

As with the MIPS R4000, SPARC uses a simple set of 32-bit instruction formats (Figure 13.13). All instructions begin with a 2-bit opcode. For most instructions, this is extended with additional opcode bits elsewhere in the format. For the Call instruction,

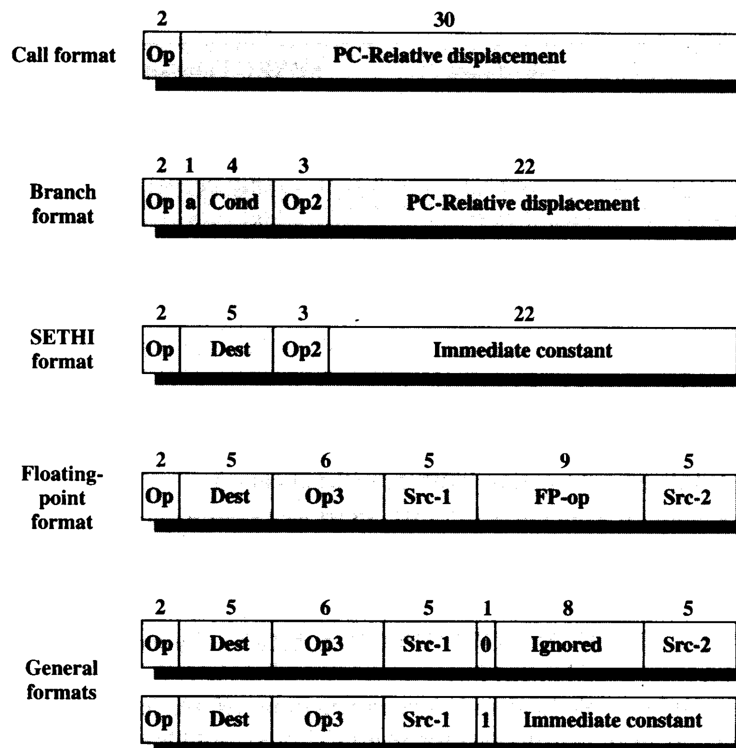


Figure 13.13 SPARC Instruction Formats

a 30-bit immediate operand is extended with two zero bits to the right to form a 32-bit PC-relative address in twos complement form. Instructions are aligned on a 32-bit boundary so that this form of addressing suffices.

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 13.5 (see Figure 13.7). However, when the annul bit is set, the instruction following the branch is executed only if the branch is taken. The processor suppresses the effect of that instruction even though it is already in the pipeline. This annul bit is useful because it makes it easier for the compiler to fill the delay slot following a conditional branch. The instruction that is the target of the branch can always be put in the delay slot, because if the branch is not taken, the instruction can be annulled. The reason this technique is desirable is that conditional branches are generally taken more than half the time.

The SETHI instruction is a special instruction used to load or store a 32-bit value. This feature is needed to load and store addresses and large constants. The SETHI instruction sets the 22 high-order bits of a register with its 22-bit immediate

operand, and zeros out the low-order 10 bits. An immediate constant of up to 13 bits can be specified in one of the general formats, and such an instruction could be used to fill in the remaining 10 bits of the register. A load or store instruction can also be used to achieve a direct addressing mode. To load a value from location K in memory, we could use the following SPARC instructions:

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

The macros %hi and %lo are used to define immediate operands consisting of the appropriate address bits of a location. This use of SETHI is similar to the use of the LUI instruction on the MIPS (Table 13.10).

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 13.13. 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.

13.8 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:** Examination of issues such as high-level language support and optimum use of VLSI real estate

Most of the work on quantitative assessment has been done by those working on RISC systems [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.
- 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].

In more recent years, the RISC versus CISC controversy has died down to a great extent. This is because there has been a gradual convergence of the technologies. As chip densities and raw hardware speeds increase, RISC systems have become more complex. At the same time, in an effort to squeeze out maximum performance, CISC designs have focused on issues traditionally associated with RISC, such as an increased number of general-purpose registers and increased emphasis on instruction pipeline design.

13.9 RECOMMENDED READING

Two classic overview papers on RISC are [PATT85a] and [HENN84]. Another survey article is [STAL88]. Accounts of two pioneering RISC efforts are provided by [RADI83] and [PATT82a].

[KANE92] covers the commercial MIPS machine in detail. [MIRA92] provides a good overview of the MIPS R4000. [BASH91] discusses the evolution from the R3000 pipeline to the R4000 superpipeline. The SPARC is covered in some detail in [DEWA90].

BASH91 Bashteen, A.; Lui, I.; and Mullan, J. "A Superpipeline Approach to the MIPS Architecture." *Proceedings, COMPCON Spring '91,* February 1991.

DEWA90 Dewar, R., and Smosna, M. *Microprocessors: A Programmer's View.* New York: McGraw-Hill, 1990.

HENN84 Hennessy, J. "VLSI Processor Architecture." *IEEE Transactions on Computers,* December 1984.

KANE92 Kane, G., and Heinrich, J. *MIPS RISC Architecture.* Englewood Cliffs, NJ: Prentice Hall, 1992.

MIRA92 Mirapuri, S.; Woodacre, M.; and Vasseghi, N. "The MIPS R4000 Processor." *IEEE Micro,* April 1992.

PATT82a Patterson, D., and Sequin, C. "A VLSI RISC." *Computer,* September 1982.

PATT85a Patterson, D. "Reduced Instruction Set Computers." *Communications of the ACM,* January 1985.

RADI83 Radin, G. "The 801 Minicomputer." *IBM Journal of Research and Development,* May 1983.

STAL88 Stallings, W. "Reduced Instruction Set Computer Architecture." *Proceedings of the IEEE,* January 1988.

13.10 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

complex instruction set computer (CISC)	delayed load	register file
delayed branch	high-level language (HLL)	register window
	reduced instruction set computer (RISC)	SPARC

Review Questions

13.1 What are some typical distinguishing characteristics of RISC organization?

13.2 Briefly explain the two basic approaches used to minimize register-memory operations on RISC machines.

13.3 If a circular register buffer is used to handle local variables for nested procedures, describe two approaches for handling global variables.

13.4 What are some typical characteristics of a RISC instruction set architecture?

13.5 What is a delayed branch?

Problems

13.1 Considering the call–return pattern in Figure 4.16, how many overflows and underflows (each of which causes a register save/restore) will occur with a window size of

a. 5?
b. 8?
c. 16?

13.2 In the discussion of Figure 13.2, it was stated that only the first two portions of a window are saved or restored. Why is it not necessary to save the temporary registers?

13.3 We wish to determine the execution time for a given program using the various pipelining schemes discussed in Section 13.5. Let

N = number of executed instructions
D = number of memory accesses
J = number of jump instructions

For the simple sequential scheme (Figure 13.6a), the execution time is $2N + D$ stages. Derive formulas for two-stage, three-stage, and four-stage pipelining.

13.4 Reorganize the code sequence in Figure 13.6d to reduce the number of NOOPs.

13.5 Consider the following code fragment in a high-level language:

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

Assume that Q is an array of 32-byte records and the VAL field is in the first 4 bytes of each record. Using 80 × 86 code, we can compile this program fragment as follows:

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

This program makes use of the IMUL instruction, which multiplies the second operand by the immediate value in the third operand and places the result in the first operand (see Problem 10.13). A RISC advocate would like to demonstrate that a clever compiler can eliminate unnecessarily complex instructions such as IMUL. Provide the demonstration by rewriting the above 80 × 86 program without using the IMUL instruction.

13.6 Consider the following loop:

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

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

```
        LD      R1, 0            ;keep value of S in R1
        LD      R2, 1            ;keep value of K in R2
LP      SUB     R1, R1, R2       ;S := S − K
        BEQ     R2, 100, EXIT    ;done if K = 100
        ADD     R2, R2, 1        ;else increment K
        JMP     LP               ;back to start of loop
```

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

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

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

a. First do the register mapping and then any possible instruction reordering. How many machine registers are used? Has there been any pipeline improvement?

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

13.8 Add entries for the following processors to Table 13.7:

a. Pentium II

b. PowerPC

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

a. Register-to-register move

b. Increment, decrement

c. Complement

d. Negate

e. Clear

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

13.11 SPARC is lacking a number of instructions commonly found on CISC machines. Some of these are easily simulated using either register R0, which is always set to 0, or a constant operand. These simulated instructions are called pseudoinstructions and are recognized by the SPARC compiler. Show how to simulate the following pseudoinstructions, each with a single SPARC instruction. In all of these, src and dst refer to registers. *Hint:* A store to R0 has no effect.

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

13.12 Consider the following code fragment:

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

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

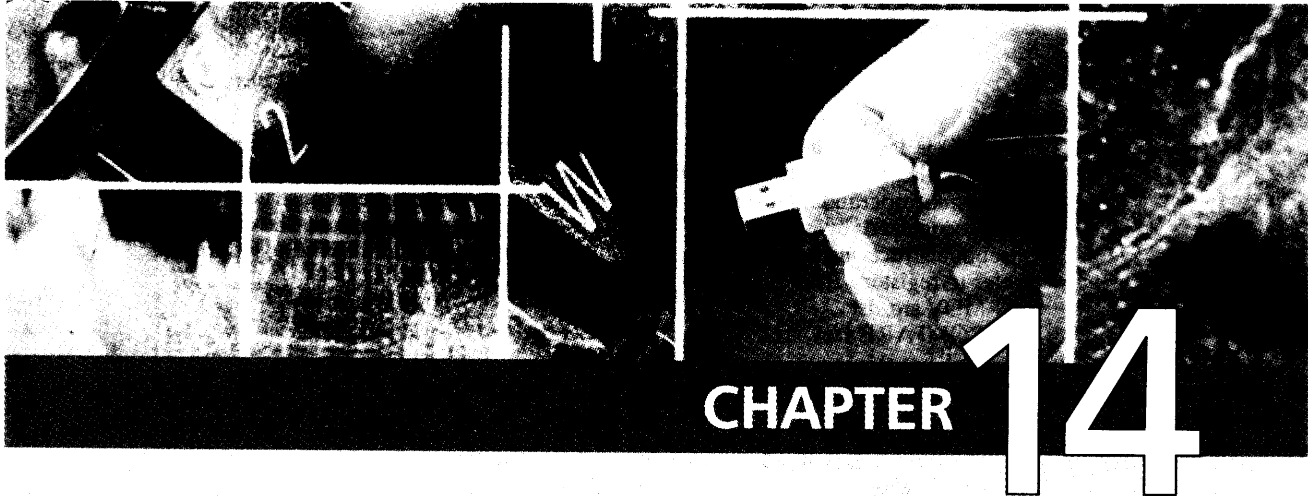
```
      sethi  %hi(K), %r8              ;load high-order 22 bits of address of location
                                      ;K into register r8
      ld     [%r8 + %lo(K)], %r8      ;load contents of location K into r8
      cmp    %r8, 10                  ;compare contents of r8 with 10
      ble    L1                       ;branch if (r8) = 10
      nop
      sethi  %hi(K), %r9
      ld     [%r9 + %lo(K)], %r9      ;load contents of location K into r9
      inc    %r9                      ;add 1 to (r9)
      sethi  %hi(L), %r10
      st     %r9, [%r10 + %lo(L)]     ;store (r9) into location L
      b      L2
      nop
L1:   sethi  %hi(K), %r11
      ld     [%r11 + %lo(K)], %r12    ;load contents of location K into r12
      dec    %r12                     ;subtract 1 from (r12)
      sethi  %hi(L), %r13
      st     %r12, [%r13 + %lo(L)]    ;store (r12) into location L
L2:
```

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

a. Standard compiler optimizations that have nothing to do with RISC machines are generally effective in being able to perform two transformations on the foregoing code. Notice that two of the loads are unnecessary and that the two stores can be merged if the store is moved to a different place in the code. Show the program after making these two changes.

b. It is now possible to perform some optimizations peculiar to SPARC. The nop after the ble can be replaced by moving another instruction into that delay slot and setting the annul bit on the ble instruction (expressed as ble,a L1). Show the program after this change.

c. There are now two unnecessary instructions. Remove these and show the resulting program.



Instruction-Level Parallelism and Superscalar Processors

14.1 Overview

14.2 Design Issues

14.3 Pentium 4

14.4 PowerPC

14.5 Recommended Reading

14.6 Key Terms, Review Questions, and Problems