2 Instruction Set Principles and Examples

A n *Add the number in storage location* n *into the accumulator.*

E n *If the number in the accumulator is greater than or equal to zero execute next the order which stands in storage location* n; *otherwise proceed serially.*

Z *Stop the machine and ring the warning bell.*

Wilkes and Renwick
Selection from the List of 18 Machine Instructions for the EDSAC (1949)

2.1 Introduction

In this chapter we concentrate on instruction set architecture—the portion of the machine visible to the programmer or compiler writer. This chapter introduces the wide variety of design alternatives available to the instruction set architect. In particular, this chapter focuses on four topics. First, we present a taxonomy of instruction set alternatives and give some qualitative assessment of the advantages and disadvantages of various approaches. Second, we present and analyze some instruction set measurements that are largely independent of a specific instruction set. Third, we address the issue of languages and compilers and their bearing on instruction set architecture. Finally, the *Putting It All Together* section shows how these ideas are reflected in the DLX instruction set, which is typical of recent instruction set architectures. The appendices add four examples of these recent architectures—MIPS, Power PC, Precision Architecture, SPARC—and one older architecture, the 80x86. Before we discuss how to classify architectures, we need to say something about instruction set measurement.

Throughout this chapter, we examine a wide variety of architectural measurements. These measurements depend on the programs measured and on the

compilers used in making the measurements. The results should not be interpreted as absolute, and you might see different data if you did the measurement with a different compiler or a different set of programs. The authors believe that the measurements shown in these chapters are reasonably indicative of a class of typical applications. Many of the measurements are presented using a small set of benchmarks, so that the data can be reasonably displayed and the differences among programs can be seen. An architect for a new machine would want to analyze a much larger collection of programs to make his architectural decisions. All the measurements shown are *dynamic*—that is, the frequency of a measured event is weighed by the number of times that event occurs during execution of the measured program.

We begin by exploring how instruction set architectures can be classified and analyzed.

2.2 Classifying Instruction Set Architectures

The type of internal storage in the CPU is the most basic differentiation, so in this section we will focus on the alternatives for this portion of the architecture. The major choices are a stack, an accumulator, or a set of registers. Operands may be named explicitly or implicitly: The operands in a *stack architecture* are implicitly on the top of the stack, in an *accumulator architecture* one operand is implicitly the accumulator, and *general-purpose register architectures* have only explicit operands—either registers or memory locations. The explicit operands may be accessed directly from memory or may need to be first loaded into temporary storage, depending on the class of instruction and choice of specific instruction. Figure 2.1 shows how the code sequence `C = A + B` would typically appear on these three classes of instruction sets. As Figure 2.1 shows, there are really two classes of register machines. One can access memory as part of any instruction, called *register-memory* architecture, and one can access memory only with load and store instructions, called *load-store* or *register-register* architecture. A third class, not found in machines shipping today, keeps all operands in memory and is called a *memory-memory* architecture.

Stack	Accumulator	Register (register-memory)	Register (load-store)
`Push A`	`Load A`	`Load R1,A`	`Load R1,A`
`Push B`	`Add B`	`Add R1,B`	`Load R2,B`
`Add`	`Store C`	`Store C,R1`	`Add R3,R1,R2`
`Pop C`			`Store C,R3`

FIGURE 2.1 The code sequence for C = A + B for four instruction sets. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed.

Although most early machines used stack or accumulator-style architectures, virtually every machine designed after 1980 uses a load-store register architecture. The major reasons for the emergence of general-purpose register (GPR) machines are twofold. First, registers—like other forms of storage internal to the CPU—are faster than memory. Second, registers are easier for a compiler to use and can be used more effectively than other forms of internal storage. For example, on a register machine the expression `(A*B) – (C*D) – (E*F)` may be evaluated by doing the multiplications in any order, which may be more efficient because of the location of the operands or because of pipelining concerns (see Chapter 3). But on a stack machine the expression must be evaluated left to right, unless special operations or swaps of stack positions are done.

More importantly, registers can be used to hold variables. When variables are allocated to registers, the memory traffic reduces, the program speeds up (since registers are faster than memory), and the code density improves (since a register can be named with fewer bits than can a memory location). Compiler writers would prefer that all registers be equivalent and unreserved. Older machines compromise this desire by dedicating registers to special uses, effectively decreasing the number of general-purpose registers. If the number of truly general-purpose registers is too small, trying to allocate variables to registers will not be profitable. Instead, the compiler will reserve all the uncommitted registers for use in expression evaluation.

How many registers are sufficient? The answer of course depends on how they are used by the compiler. Most compilers reserve some registers for expression evaluation, use some for parameter passing, and allow the remainder to be allocated to hold variables.

Two major instruction set characteristics divide GPR architectures. Both characteristics concern the nature of operands for a typical arithmetic or logical instruction (ALU instruction). The first concerns whether an ALU instruction has two or three operands. In the three-operand format, the instruction contains a result and two source operands. In the two-operand format, one of the operands is both a source and a result for the operation. The second distinction among GPR architectures concerns how many of the operands may be memory addresses in ALU instructions. The number of memory operands supported by a typical ALU instruction may vary from none to three. Combinations of these two attributes are shown in Figure 2.2, with examples of machines. Although there are seven possible combinations, three serve to classify nearly all existing machines. As we mentioned earlier, these three are register-register (also called load-store), register-memory, and memory-memory.

Number of memory addresses	Maximum number of operands allowed	Examples
0	3	SPARC, MIPS, Precision Architecture, PowerPC, ALPHA
1	2	Intel 80x86, Motorola 68000
2	2	VAX (also has three-operand formats)
3	3	VAX (also has two-operand formats)

FIGURE 2.2 Possible combinations of memory operands and total operands per typical ALU instruction with examples of machines. Machines with no memory reference per ALU instruction are called load-store or register-register machines. Instructions with multiple memory operands per typical ALU instruction are called register-memory or memory-memory, according to whether they have one or more than one memory operand.

The advantages and disadvantages of each of these alternatives are shown in Figure 2.3. Of course, these advantages and disadvantages are not absolutes: They are qualitative and their actual impact depends on the compiler and implementation strategy. A GPR machine with memory-memory operations can easily be subsetted by the compiler and used as a register-register machine. One of the most pervasive architectural impacts is on instruction encoding and the number of instructions needed to perform a task.We will see the impact of these architectural alternatives on implementation approaches in Chapters 3 and 4.

Type	Advantages	Disadvantages
Register-register (0,3)	Simple, fixed-length instruction encoding. Simple code-generation model. Instructions take similar numbers of clocks to execute (see Ch 3).	Higher instruction count than architectures with memory references in instructions. Some instructions are short and bit encoding may be wasteful.
Register-memory (1,2)	Data can be accessed without loading first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction varies by operand location.
Memory-memory (3,3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. Also, large variation in work per instruction. Memory accesses create memory bottleneck.

FIGURE 2.3 Advantages and disadvantages of the three most common types of general-purpose register machines. The notation (*m, n*) means *m* memory operands and *n* total operands. In general, machines with fewer alternatives make the compiler's task simpler since there are fewer decisions for the compiler to make. Machines with a wide variety of flexible instruction formats reduce the number of bits required to encode the program. A machine that uses a small number of bits to encode the program is said to have good *instruction density*—a smaller number of bits do as much work as a larger number on a different architecture. The number of registers also affects the instruction size.

Summary: Classifying Instruction Set Architectures

Here and in subsections at the end of sections 2.3 to 2.7 we summarize those characteristics we would expect to find in a new instruction set architecture, building the foundation for the DLX architecture introduced in section 2.8. From this section we should clearly expect the use of general-purpose registers. Figure 2.3, combined with the following chapter on pipelining, lead to the expectation of a register-register (also called load-store) architecture.

With the class of architecture covered, the next topic is addressing operands.

2.3 Memory Addressing

Independent of whether the architecture is register-register or allows any operand to be a memory reference, it must define how memory addresses are interpreted and how they are specified. We deal with these two topics in this section. The measurements presented here are largely, but not completely, machine independent. In some cases the measurements are significantly affected by the compiler technology. These measurements have been made using an optimizing compiler, since compiler technology is playing an increasing role.

Interpreting Memory Addresses

How is a memory address interpreted? That is, what object is accessed as a function of the address and the length? All the instruction sets discussed in this book are byte addressed and provide access for bytes (8 bits), half words (16 bits), and words (32 bits). Most of the machines also provide access for double words (64 bits).

There are two different conventions for ordering the bytes within a word. *Little Endian* byte order puts the byte whose address is "x...x00" at the least-significant position in the word (the little end). *Big Endian* byte order puts the byte whose address is "x...x00" at the most-significant position in the word (the big end). In Big Endian addressing, the address of a datum is the address of the most-significant byte; while in Little Endian, the address of a datum is the address of the least-significant byte. When operating within one machine, the byte order is often unnoticeable—only programs that access the same locations as both words and bytes can notice the difference. Byte order is a problem when exchanging data among machines with different orderings, however. Little Endian ordering also fails to match normal ordering of words when strings are compared. Strings appear "SDRAWKCAB" in the registers.

In many machines, accesses to objects larger than a byte must be *aligned*. An access to an object of size s bytes at byte address A is aligned if $A \bmod s = 0$. Figure 2.4 shows the addresses at which an access is aligned or misaligned.

Object addressed	Aligned at byte offsets	Misaligned at byte offsets
Byte	0,1,2,3,4,5,6,7	Never
Half word	0,2,4,6	1,3,5,7
Word	0,4	1,2,3,5,6,7
Double word	0	1,2,3,4,5,6,7

FIGURE 2.4 Aligned and misaligned accesses of objects. The byte offsets are specified for the low-order three bits of the address.

Why would someone design a machine with alignment restrictions? Misalignment causes hardware complications, since the memory is typically aligned on a word or double-word boundary. A misaligned memory access will, therefore, take multiple aligned memory references.Thus, even in machines that allow misaligned access, programs with aligned accesses run faster.

Even if data are aligned, supporting byte and half-word accesses requires an alignment network to align bytes and half words in registers. Depending on the instruction, the machine may also need to sign-extend the quantity. On some machines a byte or half word does not affect the upper portion of a register. For stores only the affected bytes in memory may be altered. (Although all the machines discussed in this book permit byte and half-word accesses to memory, only the Intel 80x86 supports ALU operations on register operands with a size shorter than a word.)

Addressing Modes

We now know what bytes to access in memory given an address. In this subsection we will look at addressing modes—how architectures specify the address of an object they will access. In GPR machines, an addressing mode can specify a constant, a register, or a location in memory. When a memory location is used, the actual memory address specified by the addressing mode is called the *effective address*.

Figure 2.5 shows all the data-addressing modes that have been used in recent machines. Immediates or literals are usually considered memory-addressing modes (even though the value they access is in the instruction stream), although registers are often separated. We have kept addressing modes that depend on the program counter, called *PC-relative addressing*, separate. PC-relative addressing is used primarily for specifying code addresses in control transfer instructions. The use of PC-relative addressing in control instructions is discussed in section 2.4.

Figure 2.5 shows the most common names for the addressing modes, though the names differ among architectures. In this figure and throughout the book, we will use an extension of the C programming language as a hardware description notation. In this figure, only one non-C feature is used: The left arrow (←) is used

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4]←Regs[R4]+ Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4]←Regs[R4]+3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4]←Regs[R4]+ Mem[100+Regs[R1]]`	Accessing local variables.
Register deferred or indirect	`Add R4,(R1)`	`Regs[R4]←Regs[R4]+ Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1 + R2)`	`Regs[R3]←Regs[R3]+ Mem[Regs[R1]+Regs[R2]]`	Sometimes useful in array addressing: `R1` = base of array; `R2` = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1]←Regs[R1]+ Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect or memory deferred	`Add R1,@(R3)`	`Regs[R1]←Regs[R1]+ Mem[Mem[Regs[R3]]]`	If `R3` is the address of a pointer *p*, then mode yields **p*.
Autoincrement	`Add R1,(R2)+`	`Regs[R1]←Regs[R1]+ Mem[Regs[R2]]` `Regs[R2]←Regs[R2]+`*d*	Useful for stepping through arrays within a loop. `R2` points to start of array; each reference increments `R2` by size of an element, *d*.
Autodecrement	`Add R1,–(R2)`	`Regs[R2]←Regs[R2]–`*d* `Regs[R1]←Regs[R1]+ Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1]← Regs[R1]+ Mem[100+Regs[R2]+Regs [R3]*`*d*`]`	Used to index arrays. May be applied to any indexed addressing mode in some machines.

FIGURE 2.5 Selection of addressing modes with examples, meaning, and usage. The extensions to C used in the hardware descriptions are defined above. In autoincrement/decrement and scaled addressing modes, the variable *d* designates the size of the data item being accessed (i.e., whether the instruction is accessing 1, 2, 4, or 8 bytes); this means that these addressing modes are only useful when the elements being accessed are adjacent in memory. In our measurements, we use the first name shown for each mode.

for assignment. We also use the array `Mem` as the name for main memory and the array `Regs` for registers. Thus, `Mem[Regs[R1]]` refers to the contents of the memory location whose address is given by the contents of register 1 (`R1`). Later, we will introduce extensions for accessing and transferring data smaller than a word.

Addressing modes have the ability to significantly reduce instruction counts; they also add to the complexity of building a machine and may increase the average CPI (clock cycles per instruction) of machines that implement those modes.

Thus, the usage of various addressing modes is quite important in helping the architect choose what to include.

Figure 2.6 shows the results of measuring addressing mode usage patterns in three programs on the VAX architecture. We use the VAX architecture for a few measurements in this chapter because it has the fewest restrictions on memory addressing. For example, it supports all the modes shown in Figure 2.5. Most measurements in this chapter, however, will use the more recent load-store architectures to show how programs use instruction sets of current machines.

As Figure 2.6 shows, immediate and displacement addressing dominate addressing mode usage. Let's look at some properties of these two heavily used modes.

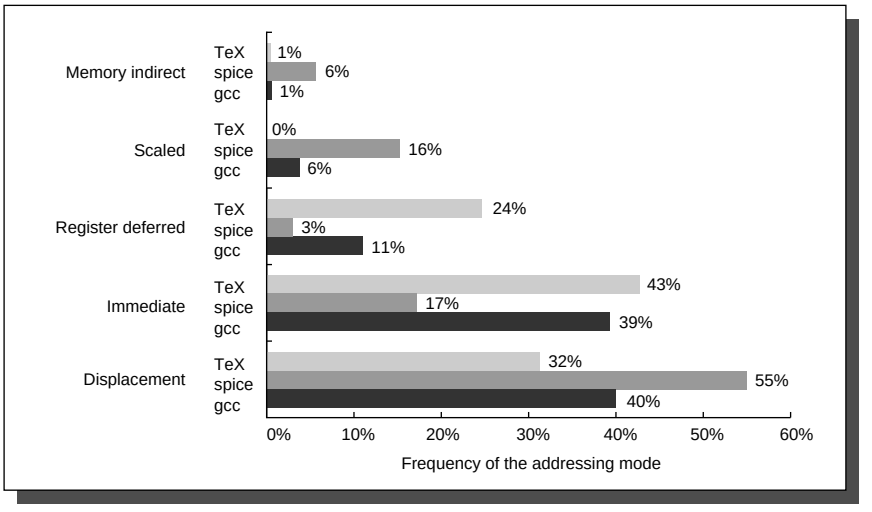

FIGURE 2.6 Summary of use of memory addressing modes (including immediates). The data were taken on a VAX using three programs from SPEC89. Only the addressing modes with an average frequency of over 1% are shown. The PC-relative addressing modes, which are used almost exclusively for branches, are not included. Displacement mode includes all displacement lengths (8, 16, and 32 bit). Register modes, which are not counted, account for one-half of the operand references, while memory addressing modes (including immediate) account for the other half. The memory indirect mode on the VAX can use displacement, autoincrement, or autodecrement to form the initial memory address; in these programs, almost all the memory indirect references use displacement mode as the base. Of course, the compiler affects what addressing modes are used; we discuss this further in section 2.7. These major addressing modes account for all but a few percent (0% to 3%) of the memory accesses.

Displacement Addressing Mode

The major question that arises for a displacement-style addressing mode is that of the range of displacements used. Based on the use of various displacement sizes, a decision of what sizes to support can be made. Choosing the displacement field

sizes is important because they directly affect the instruction length. Measurements taken on the data access on a load-store architecture using our benchmark programs are shown in Figure 2.7. We will look at branch offsets in the next section—data accessing patterns and branches are so different, little is gained by combining them.

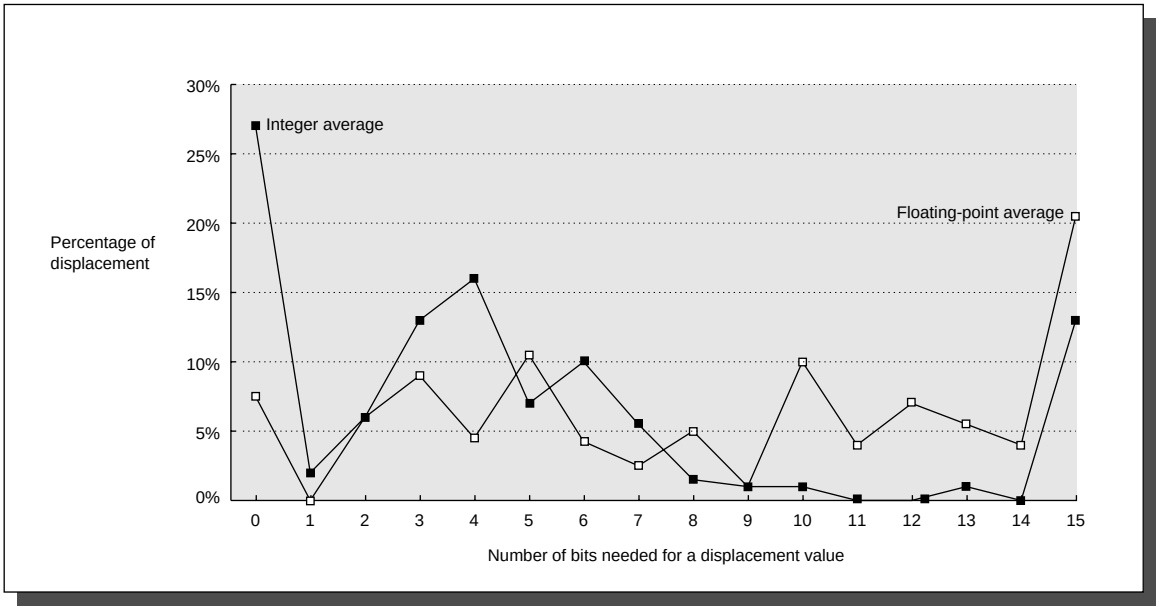

FIGURE 2.7 Displacement values are widely distributed. The x axis is $\log_2$ of the displacement; that is, the size of a field needed to represent the magnitude of the displacement. These data were taken on the MIPS architecture, showing the average of five programs from SPECint92 (compress, espresso, eqntott, gcc, li) and the average of five programs from SPECfp92 (dudoc, ear, hydro2d, mdljdp2, su2cor). Although there are a large number of small values in this data, there are also a fair number of large values. The wide distribution of displacement values is due to multiple storage areas for variables and different displacements used to access them. The different storage areas and their access patterns are discussed further in section 2.7. The graph shows only the magnitude of the displacement and not the sign, which is heavily affected by the storage layout. The entry corresponding to 0 on the x axis shows the percentage of displacements of value 0. The vast majority of the displacements are positive, but a majority of the largest displacements (14+ bits) are negative. Again, this is due to the overall addressing scheme used by the compiler and might change with a different compilation scheme. Since this data was collected on a machine with 16-bit displacements, it cannot tell us anything about accesses that might want to use a longer displacement. Such accesses are broken into two separate instructions—the first of which loads the upper 16 bits of a base register. By counting the frequency of these "load high immediate" instructions, which have limited use for other purposes, we can bound the number of accesses with displacements potentially larger than 16 bits. Such an analysis indicates that we may actually require a displacement longer than 16 bits for about 1% of immediates on SPECint92 and 1% of those for SPECfp92. Relating this data to the graph above, if it were widened to 32 bits we would see 1% of immediates collectively between sizes 16 and 31 for both SPECint92 and SPECfp92. And if the displacement is larger than 15 bits, it is likely to be quite a bit larger since such constants are large, as shown in Figure 2.9 on page 79.To evaluate the choice of displacement length, we might also want to examine a cumulative distribution, as shown in Exercise 2.1 (see Figure 2.32 on page 119). In summary, 12 bits of displacement would capture about 75% of the full 32-bit displacements and 16 bits should capture about 99%.

Immediate or Literal Addressing Mode

Immediates can be used in arithmetic operations, in comparisons (primarily for branches), and in moves where a constant is wanted in a register. The last case occurs for constants written in the code, which tend to be small, and for address constants, which can be large. For the use of immediates it is important to know whether they need to be supported for all operations or for only a subset. The chart in Figure 2.8 shows the frequency of immediates for the general classes of integer operations in an instruction set.

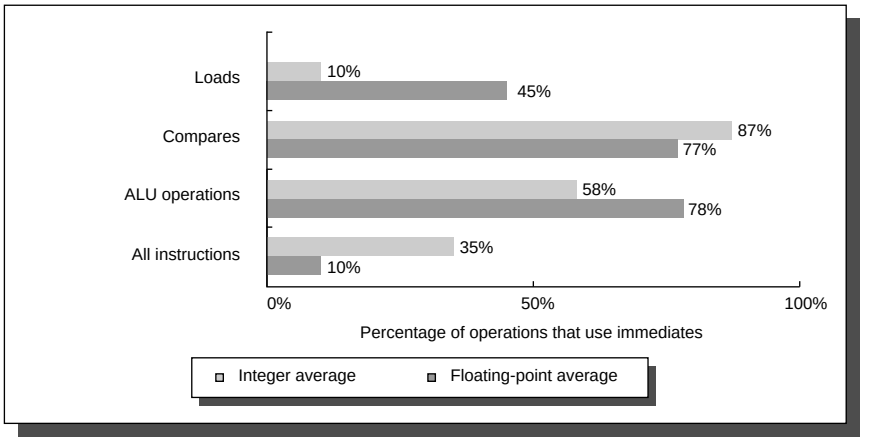

FIGURE 2.8 We see that for ALU operations about one-half to three-quarters of the operations have an immediate operand, while 75% to 85% of compare operations use an immediate operation. (For ALU operations, shifts by a constant amount are included as operations with immediate operands.) For loads, the load immediate instructions load 16 bits into either half of a 32-bit register. These load immediates are not loads in a strict sense because they do not reference memory. In some cases, a pair of load immediates may be used to load a 32-bit constant, but this is rare. The compares include comparisons against zero that are done in conditional branches based on this comparison. These measurements were taken on the DLX architecture with full compiler optimization (see section 2.7). The compiler attempts to use simple compares against zero for branches whenever possible, because these branches are efficiently supported in the architecture. Note that the bottom bars show that integer programs use immediates in about one-third of the instructions, while floating-point programs use immediates in about one-tenth of the instructions. Floating-point programs have many data transfers and operations on floating-point data that do not have immediate forms in the DLX instruction set. (These percentages are the averages of the same 10 programs as in Figure 2.7 on page 77.)

Another important instruction set measurement is the range of values for immediates. Like displacement values, the sizes of immediate values affect instruction lengths. As Figure 2.9 shows, immediate values that are small are most heavily used. Large immediates are sometimes used, however, most likely in addressing calculations. The data in Figure 2.9 were taken on a VAX because, un-

like recent load-store architectures, it supports 32-bit long immediates. For these measurements the VAX has the drawback that many of its instructions have zero as an implicit operand. These include instructions to compare against zero and to store zero into a word. Because of the use of these instructions, the measurements show less frequent use of zero than on architectures without such instructions.

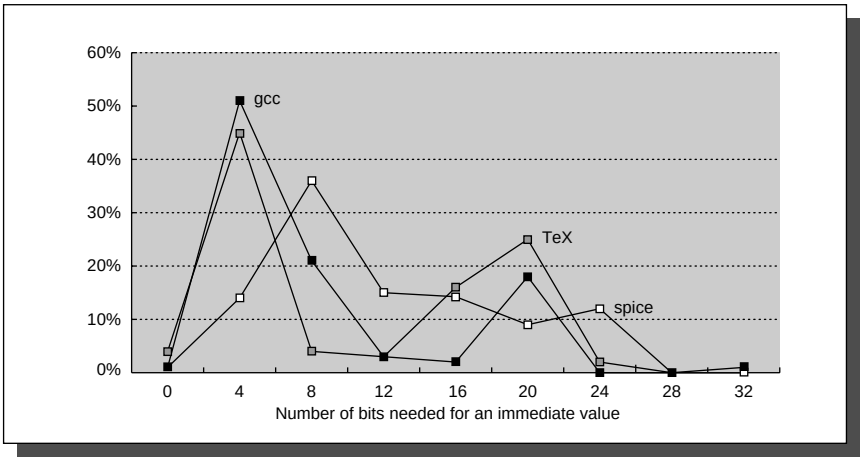

FIGURE 2.9 The distribution of immediate values is shown. The *x* axis shows the number of bits needed to represent the magnitude of an immediate value—0 means the immediate field value was 0. The vast majority of the immediate values are positive: Overall, less than 6% of the immediates are negative.These measurements were taken on a VAX, which supports a full range of immediates and sizes as operands to any instruction. The measured programs are gcc, spice, and TeX. Note that 50% to 70% of the immediates fit within 8 bits and 75% to 80% fit within 16 bits.

Summary: Memory Addressing

First, because of their popularity, we would expect a new architecture to support at least the following addressing modes: displacement, immediate, and register deferred. Figure 2.6 on page 76 shows they represent 75% to 99% of the addressing modes used in our measurements. Second, we would expect the size of the address for displacement mode to be at least 12 to 16 bits, since the caption in Figure 2.7 on page 77 suggests these sizes would capture 75% to 99% of the displacements. Third, we would expect the size of the immediate field to be at least 8 to 16 bits. As the caption in Figure 2.9 suggests, these sizes would capture 50% to 80% of the immediates.

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, and, subtract, or
Data transfer	Loads-stores (move instructions on machines with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel operations, compression/decompression operations

FIGURE 2.10 Categories of instruction operators and examples of each. All machines generally provide a full set of operations for the first three categories. The support for system functions in the instruction set varies widely among architectures, but all machines must have some instruction support for basic system functions. The amount of support in the instruction set for the last four categories may vary from none to an extensive set of special instructions. Floating-point instructions will be provided in any machine that is intended for use in an application that makes much use of floating point. These instructions are sometimes part of an optional instruction set. Decimal and string instructions are sometimes primitives, as in the VAX or the IBM 360, or may be synthesized by the compiler from simpler instructions. Graphics instructions typically operate on many smaller data items in parallel; for example, performing eight 8-bit additions on two 64-bit operands.

2.4 Operations in the Instruction Set

The operators supported by most instruction set architectures can be categorized, as in Figure 2.10. One rule of thumb across all architectures is that the most widely executed instructions are the simple operations of an instruction set. For example, Figure 2.11 shows 10 simple instructions that account for 96% of instructions executed for a collection of integer programs running on the popular Intel 80x86. Hence the implementor of these instructions should be sure to make these fast, as they are the common case.

Because the measurements of branch and jump behavior are fairly independent of other measurements, we examine the use of control-flow instructions next.

Instructions for Control Flow

There is no consistent terminology for instructions that change the flow of control. In the 1950s they were typically called *transfers*. Beginning in 1960 the name *branch* began to be used. Later, machines introduced additional names. Throughout this book we will use *jump* when the change in control is unconditional and *branch* when the change is conditional.

Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		96%

FIGURE 2.11 The top 10 instructions for the 80x86. These percentages are the average of the same five SPECint92 programs as in Figure 2.7 on page 77.

We can distinguish four different types of control-flow change:

1. Conditional branches
2. Jumps
3. Procedure calls
4. Procedure returns

We want to know the relative frequency of these events, as each event is different, may use different instructions, and may have different behavior. The frequencies of these control-flow instructions for a load-store machine running our benchmarks are shown in Figure 2.12.

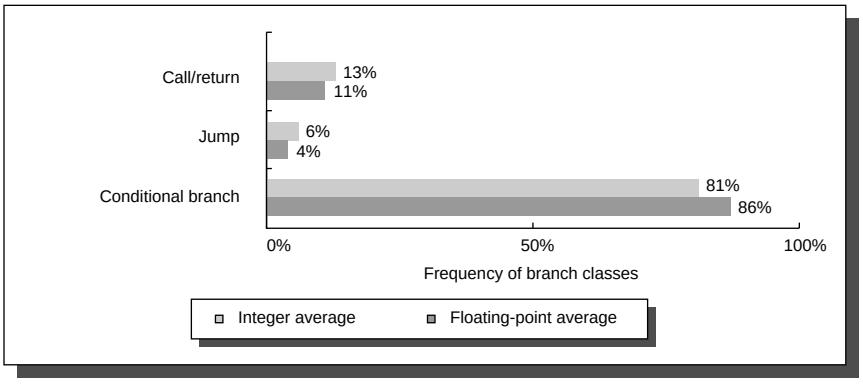

FIGURE 2.12 Breakdown of control flow instructions into three classes: calls or returns, jumps, and conditional branches. Each type is counted in one of three bars. Conditional branches clearly dominate. The programs and machine used to collect these statistics are the same as those in Figure 2.7.

The destination address of a control flow instruction must always be specified. This destination is specified explicitly in the instruction in the vast majority of cases—procedure return being the major exception—since for return the target is not known at compile time. The most common way to specify the destination is to supply a displacement that is added to the *program counter*, or PC. Control flow instructions of this sort are called *PC-relative*. PC-relative branches or jumps are advantageous because the target is often near the current instruction, and specifying the position relative to the current PC requires fewer bits. Using PC-relative addressing also permits the code to run independently of where it is loaded. This property, called *position independence*, can eliminate some work when the program is linked and is also useful in programs linked during execution.

To implement returns and indirect jumps in which the target is not known at compile time, a method other than PC-relative addressing is required. Here, there must be a way to specify the target dynamically, so that it can change at runtime. This dynamic address may be as simple as naming a register that contains the target address; alternatively, the jump may permit any addressing mode to be used to supply the target address.These register indirect jumps are also useful for three other important features: *case* or *switch* statements found in many programming languages (which select among one of several alternatives), *dynamically shared libraries* (which allow a library to be loaded only when it is actually invoked by the program), and *virtual functions* in object-oriented languages like C++ (which allow different routines to be called depending on the type of the data). In all three cases the target address is not known at compile time, and hence is usually loaded from memory into a register before the register indirect jump.

As branches generally use PC-relative addressing to specify their targets, a key question concerns how far branch targets are from branches. Knowing the distribution of these displacements will help in choosing what branch offsets to support and thus will affect the instruction length and encoding. Figure 2.13 shows the distribution of displacements for PC-relative branches in instructions. About 75% of the branches are in the forward direction.

Since most changes in control flow are branches, deciding how to specify the branch condition is important. The three primary techniques in use and their advantages and disadvantages are shown in Figure 2.14.

One of the most noticeable properties of branches is that a large number of the comparisons are simple equality or inequality tests, and a large number are comparisons with zero. Thus, some architectures choose to treat these comparisons as special cases, especially if a *compare and branch* instruction is being used. Figure 2.15 shows the frequency of different comparisons used for conditional branching. The data in Figure 2.8 said that a large percentage of the comparisons had an immediate operand, and while not shown, 0 was the most heavily used immediate. When we combine this with the data in Figure 2.15, we can see that a significant percentage (over 50%) of the integer compares in branches are simple tests for equality with 0.

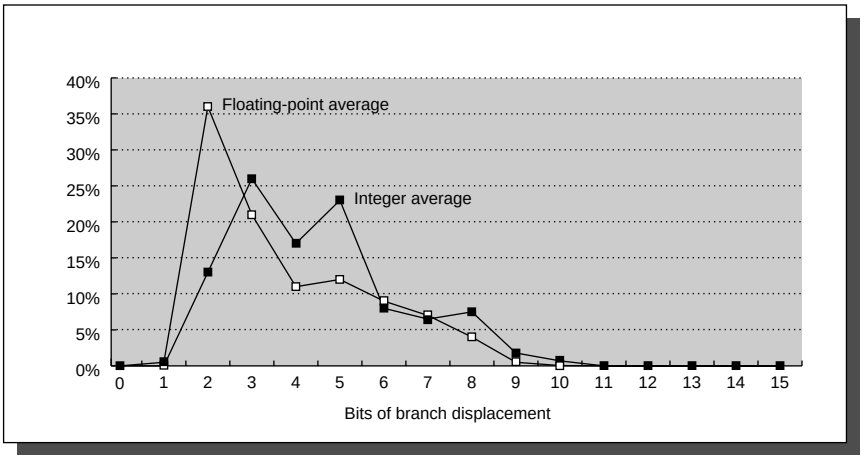

FIGURE 2.13 Branch distances in terms of number of instructions between the target and the branch instruction. The most frequent branches in the integer programs are to targets that are four to seven instructions away. This tells us that short displacement fields often suffice for branches and that the designer can gain some encoding density by having a shorter instruction with a smaller branch displacement. These measurements were taken on a load-store machine (DLX architecture). An architecture that requires fewer instructions for the same program, such as a VAX, would have shorter branch distances. Similarly, the number of bits needed for the displacement may change if the machine allows instructions to be arbitrarily aligned. A cumulative distribution of this branch displacement data is shown in Exercise 2.1 (see Figure 2.32 on page 119). The programs and machine used to collect these statistics are the same as those in Figure 2.7.

Name	How condition is tested	Advantages	Disadvantages
Condition code (CC)	Special bits are set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.
Condition register	Test arbitrary register with the result of a comparison.	Simple.	Uses up a register.
Compare and branch	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch.	May be too much work per instruction.

FIGURE 2.14 The major methods for evaluating branch conditions, their advantages, and their disadvantages. Although condition codes can be set by ALU operations that are needed for other purposes, measurements on programs show that this rarely happens. The major implementation problems with condition codes arise when the condition code is set by a large or haphazardly chosen subset of the instructions, rather than being controlled by a bit in the instruction. Machines with compare and branch often limit the set of compares and use a condition register for more complex compares. Often, different techniques are used for branches based on floating-point comparison versus those based on integer comparison. This is reasonable since the number of branches that depend on floating-point comparisons is much smaller than the number depending on integer comparisons.

Procedure calls and returns include control transfer and possibly some state saving; at a minimum the return address must be saved somewhere. Some archi-

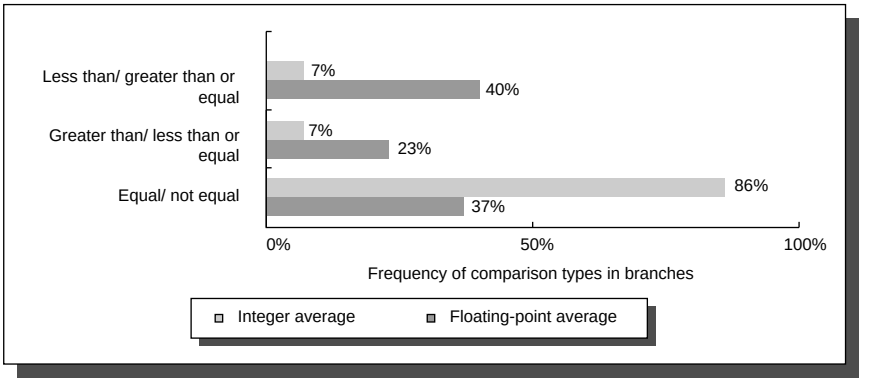

FIGURE 2.15 Frequency of different types of compares in conditional branches. This includes both the integer and floating-point compares in branches. Remember that earlier data in Figure 2.8 indicate that most integer comparisons are against an immediate operand. The programs and machine used to collect these statistics are the same as those in Figure 2.7.

tectures provide a mechanism to save the registers, while others require the compiler to generate instructions. There are two basic conventions in use to save registers. *Caller saving* means that the calling procedure must save the registers that it wants preserved for access after the call. *Callee saving* means that the called procedure must save the registers it wants to use. There are times when caller save must be used because of access patterns to globally visible variables in two different procedures. For example, suppose we have a procedure P1 that calls procedure P2, and both procedures manipulate the global variable *x*. If P1 had allocated *x* to a register it must be sure to save *x* to a location known by P2 before the call to P2. A compiler's ability to discover when a called procedure may access register-allocated quantities is complicated by the possibility of separate compilation and situations where P2 may not touch *x* but can call another procedure, P3, that may access *x*. Because of these complications, most compilers will conservatively caller save *any* variable that may be accessed during a call.

In the cases where either convention could be used, some programs will be more optimal with callee save and some will be more optimal with caller save. As a result, the most sophisticated compilers use a combination of the two mechanisms, and the register allocator may choose which register to use for a variable based on the convention. Later in this chapter we will examine the mismatch between sophisticated instructions for automatically saving registers and the needs of the compiler.

Summary: Operations in the Instruction Set

From this section we see the importance and popularity of simple instructions: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch, jump, call, and return. Although there are many options for conditional branches, we would expect branch addressing in a new architecture to be able to jump to about 100 instructions either above or below the branch, implying a PC-relative branch displacement of at least 8 bits. We would also expect to see register-indirect and PC-relative addressing for jump instructions to support returns as well as many other features of current systems.

2.5 Type and Size of Operands

How is the type of an operand designated? There are two primary alternatives: First, the type of an operand may be designated by encoding it in the opcode—this is the method used most often. Alternatively, the data can be annotated with tags that are interpreted by the hardware. These tags specify the type of the operand, and the operation is chosen accordingly. Machines with tagged data, however, can only be found in computer museums.

Usually the type of an operand—for example, integer, single-precision floating point, character—effectively gives its size. Common operand types include character (1 byte), half word (16 bits), word (32 bits), single-precision floating point (also 1 word), and double-precision floating point (2 words). Characters are almost always in ASCII and integers are almost universally represented as two's complement binary numbers. Until the early 1980s, most computer manufacturers chose their own floating-point representation. Almost all machines since that time follow the same standard for floating point, the IEEE standard 754. The IEEE floating-point standard is discussed in detail in Appendix A.

Some architectures provide operations on character strings, although such operations are usually quite limited and treat each byte in the string as a single character. Typical operations supported on character strings are comparisons and moves.

For business applications, some architectures support a decimal format, usually called *packed decimal* or *binary-coded decimal*—4 bits are used to encode the values 0–9, and 2 decimal digits are packed into each byte. Numeric character strings are sometimes called *unpacked decimal*, and operations—called *packing* and *unpacking*—are usually provided for converting back and forth between them.

Our benchmarks use byte or character, half word (short integer), word (integer), and floating-point data types. Figure 2.16 shows the dynamic distribution of the sizes of objects referenced from memory for these programs. The frequency of access to different data types helps in deciding what types are most important to support efficiently. Should the machine have a 64-bit access path, or would

taking two cycles to access a double word be satisfactory? How important is it to support byte accesses as primitives, which, as we saw earlier, require an alignment network? In Figure 2.16, memory references are used to examine the types of data being accessed. In some architectures, objects in registers may be accessed as bytes or half words. However, such access is very infrequent—on the VAX, it accounts for no more than 12% of register references, or roughly 6% of all operand accesses in these programs. The successor to the VAX not only removed operations on data smaller than 32 bits, it also removed data transfers on these smaller sizes: The first implementations of the Alpha required multiple instructions to read or write bytes or half words.

Note that Figure 2.16 was measured on a machine with 32-bit addresses: On a 64-bit address machine the 32-bit addresses would be replaced by 64-bit addresses. Hence as 64-bit address architectures become more popular, we would expect that double-word accesses will be popular for integer programs as well as floating-point programs.

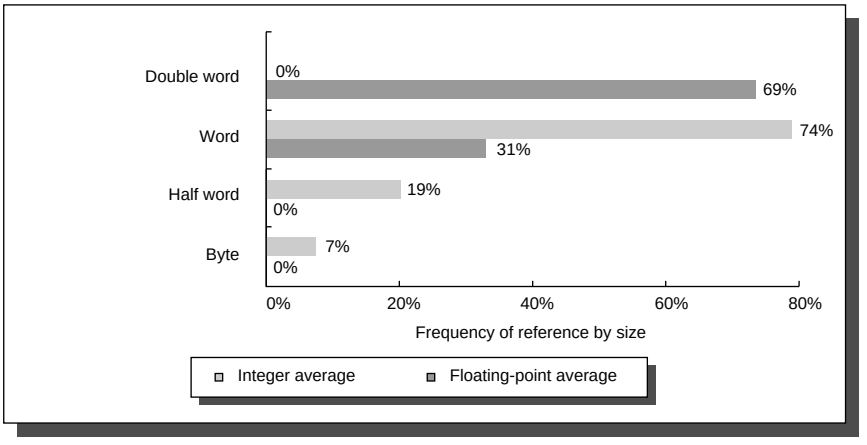

FIGURE 2.16 Distribution of data accesses by size for the benchmark programs. Access to the major data type (word or double word) clearly dominates each type of program. Half words are more popular than bytes because one of the five SPECint92 programs (eqntott) uses half words as the primary data type, and hence they are responsible for 87% of the data accesses (see Figure 2.31 on page 110). The double-word data type is used solely for double-precision floating-point in floating-point programs. These measurements were taken on the memory traffic generated on a 32-bit load-store architecture.

Summary: Type and Size of Operands

From this section we would expect a new 32-bit architecture to support 8-, 16-, and 32-bit integers and 64-bit IEEE 754 floating-point data; a new 64-bit address architecture would need to support 64-bit integers as well. The level of support for decimal data is less clear, and it is a function of the intended use of the machine as well as the effectiveness of the decimal support.

2.6 Encoding an Instruction Set

Clearly the choices mentioned above will affect how the instructions are encoded into a binary representation for execution by the CPU. This representation affects not only the size of the compiled program, it affects the implementation of the CPU, which must decode this representation to quickly find the operation and its operands. The operation is typically specified in one field, called the *opcode*. As we shall see, the important decision is how to encode the addressing modes with the operations.

This decision depends on the range of addressing modes and the degree of independence between opcodes and modes. Some machines have one to five operands with 10 addressing modes for each operand (see Figure 2.5 on page 75). For such a large number of combinations, typically a separate *address specifier* is needed for each operand: the address specifier tells what addressing mode is used to access the operand. At the other extreme is a load-store machine with only one memory operand and only one or two addressing modes; obviously, in this case, the addressing mode can be encoded as part of the opcode.

When encoding the instructions, the number of registers and the number of addressing modes both have a significant impact on the size of instructions, since the addressing mode field and the register field may appear many times in a single instruction. In fact, for most instructions many more bits are consumed in encoding addressing modes and register fields than in specifying the opcode. The architect must balance several competing forces when encoding the instruction set:

1. The desire to have as many registers and addressing modes as possible.
2. The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.
3. A desire to have instructions encode into lengths that will be easy to handle in the implementation. As a minimum, the architect wants instructions to be in multiples of bytes, rather than an arbitrary length. Many architects have chosen to use a fixed-length instruction to gain implementation benefits while sacrificing average code size.

Since the addressing modes and register fields make up such a large percentage of the instruction bits, their encoding will significantly affect how easy it is for an implementation to decode the instructions. The importance of having easily decoded instructions is discussed in Chapter 3.

Figure 2.17 shows three popular choices for encoding the instruction set. The first we call *variable*, since it allows virtually all addressing modes to be with all operations. This style is best when there are many addressing modes and operations. The second choice we call *fixed*, since it combines the operation and the

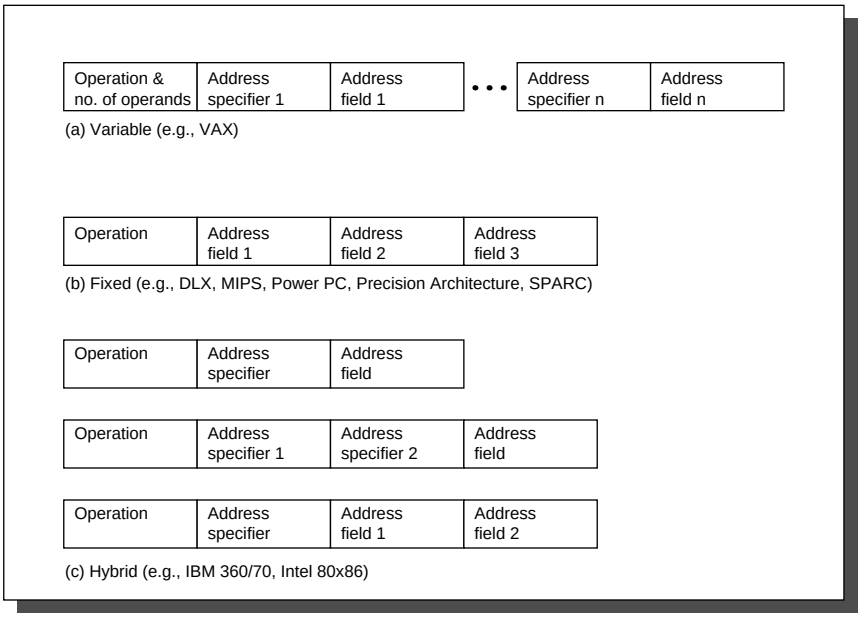

FIGURE 2.17 Three basic variations in instruction encoding. The variable format can support any number of operands, with each address specifier determining the addressing mode for that operand. The fixed format always has the same number of operands, with the addressing modes (if options exist) specified as part of the opcode (see also Figure C.3 on page C-4). Although the fields tend not to vary in their location, they will be used for different purposes by different instructions. The hybrid approach will have multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address (see also Figure D.7 on page D-12).

addressing mode into the opcode. Often fixed encoding will have only a single size for all instructions; it works best when there are few addressing modes and operations. The trade-off between variable encoding and fixed encoding is size of programs versus ease of decoding in the CPU. Variable tries to use as few bits as possible to represent the program, but individual instructions can vary widely in both size and the amount of work to be performed. For example, the VAX integer add can vary in size between 3 and 19 bytes and vary between 0 and 6 in data memory references. Given these two poles of instruction set design, the third alternative immediately springs to mind: Reduce the variability in size and work of the variable architecture but provide multiple instruction lengths so as to reduce code size. This *hybrid* approach is the third encoding alternative.

To make these general classes more specific, this book contains several examples. Fixed formats of five machines can be seen in Figure C.3 on page C-4 and the hybrid formats of the Intel 80x86 can be seen in Figure D.8 on page D-13.

Let's look at a VAX instruction to see an example of the variable encoding:

```
addl3 r1,737(r2),(r3)
```

The name `addl3` means a 32-bit integer add instruction with three operands, and this opcode takes 1 byte. A VAX address specifier is 1 byte, generally with the first 4 bits specifying the addressing mode and the second 4 bits specifying the register used in that addressing mode. The first operand specifier—`r1`—indicates register addressing using register 1, and this specifier is 1 byte long. The second operand specifier—`737(r2)`—indicates displacement addressing. It has two parts: The first part is a byte that specifies the 16-bit indexed addressing mode and base register (`r2`); the second part is the 2-byte-long displacement (`737`). The third operand specifier—(`r3`)—specifies register indirect addressing mode using register 3. Thus, this instruction has two data memory accesses, and the total length of the instruction is

$$1 + (1) + (1+2) + (1) = 6 \text{ bytes}$$

The length of VAX instructions varies between 1 and 53 bytes.

Summary: Encoding the Instruction Set

Decisions made in the components of instruction set design discussed in prior sections determine whether or not the architect has the choice between variable and fixed instruction encodings. Given the choice, the architect more interested in code size than performance will pick variable encoding, and the one more interested in performance than code size will pick fixed encoding. In Chapters 3 and 4, the impact of variability on performance of the CPU will be discussed further.

We have almost finished laying the groundwork for the DLX instruction set architecture that will be introduced in section 2.8. But before we do that, it will be helpful to take a brief look at recent compiler technology and its effect on program properties.

2.7 Crosscutting Issues: The Role of Compilers

Today almost all programming is done in high-level languages. This development means that since most instructions executed are the output of a compiler, an instruction set architecture is essentially a compiler target. In earlier times, architectural decisions were often made to ease assembly language programming. Because performance of a computer will be significantly affected by the compiler, understanding compiler technology today is critical to designing and efficiently implementing an instruction set. In earlier days it was popular to try to isolate the

compiler technology and its effect on hardware performance from the architecture and its performance, just as it was popular to try to separate an architecture from its implementation. This separation is essentially impossible with today's compilers and machines. Architectural choices affect the quality of the code that can be generated for a machine and the complexity of building a good compiler for it. Isolating the compiler from the hardware is likely to be misleading. In this section we will discuss the critical goals in the instruction set primarily from the compiler viewpoint. What features will lead to high-quality code? What makes it easy to write efficient compilers for an architecture?

The Structure of Recent Compilers

To begin, let's look at what optimizing compilers are like today. The structure of recent compilers is shown in Figure 2.18.

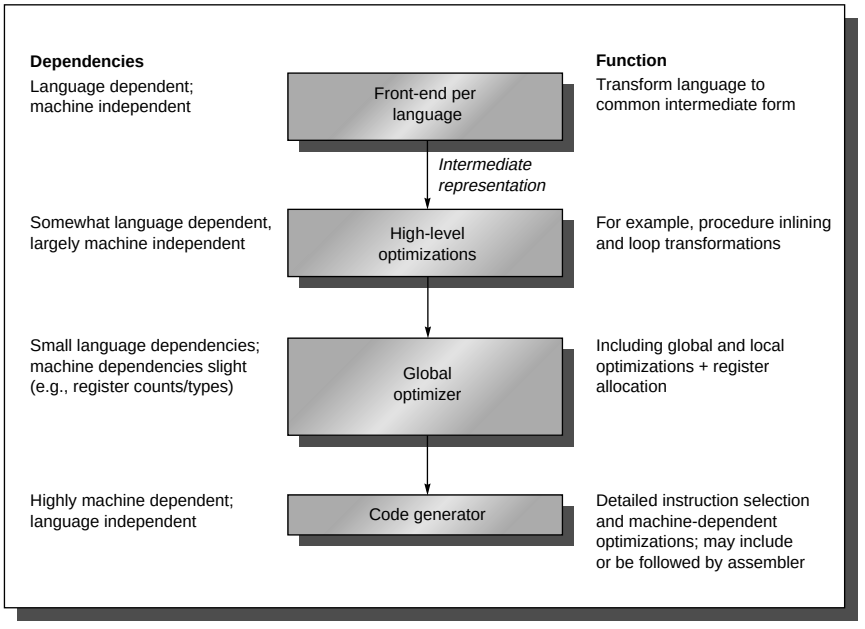

FIGURE 2.18 Current compilers typically consist of two to four passes, with more highly optimizing compilers having more passes. A *pass* is simply one phase in which the compiler reads and transforms the entire program. (The term *phase* is often used interchangeably with *pass.*) The optimizing passes are designed to be optional and may be skipped when faster compilation is the goal and lower quality code is acceptable. This structure maximizes the probability that a program compiled at various levels of optimization will produce the same output when given the same input. Because the optimizing passes are also separated, multiple languages can use the same optimizing and code-generation passes. Only a new front end is required for a new language. The high-level optimization mentioned here, *procedure inlining,* is also called *procedure integration.*

A compiler writer's first goal is correctness—all valid programs must be compiled correctly. The second goal is usually speed of the compiled code. Typically, a whole set of other goals follows these two, including fast compilation, debugging support, and interoperability among languages. Normally, the passes in the compiler transform higher-level, more abstract representations into progressively lower-level representations, eventually reaching the instruction set. This structure helps manage the complexity of the transformations and makes writing a bug-free compiler easier.

The complexity of writing a correct compiler is a major limitation on the amount of optimization that can be done. Although the multiple-pass structure helps reduce compiler complexity, it also means that the compiler must order and perform some transformations before others. In the diagram of the optimizing compiler in Figure 2.18, we can see that certain high-level optimizations are performed long before it is known what the resulting code will look like in detail. Once such a transformation is made, the compiler can't afford to go back and revisit all steps, possibly undoing transformations. This would be prohibitive, both in compilation time and in complexity. Thus, compilers make assumptions about the ability of later steps to deal with certain problems. For example, compilers usually have to choose which procedure calls to expand inline before they know the exact size of the procedure being called. Compiler writers call this problem the *phase-ordering problem*.

How does this ordering of transformations interact with the instruction set architecture? A good example occurs with the optimization called *global common subexpression elimination*. This optimization finds two instances of an expression that compute the same value and saves the value of the first computation in a temporary. It then uses the temporary value, eliminating the second computation of the expression. For this optimization to be significant, the temporary must be allocated to a register. Otherwise, the cost of storing the temporary in memory and later reloading it may negate the savings gained by not recomputing the expression. There are, in fact, cases where this optimization actually slows down code when the temporary is not register allocated. Phase ordering complicates this problem, because register allocation is typically done near the end of the global optimization pass, just before code generation. Thus, an optimizer that performs this optimization *must* assume that the register allocator will allocate the temporary to a register.

Optimizations performed by modern compilers can be classified by the style of the transformation, as follows:

1. *High-level optimizations* are often done on the source with output fed to later optimization passes.
2. *Local optimizations* optimize code only within a straight-line code fragment (called a *basic block* by compiler people).

3. *Global optimizations* extend the local optimizations across branches and introduce a set of transformations aimed at optimizing loops.
4. *Register allocation.*
5. *Machine-dependent optimizations* attempt to take advantage of specific architectural knowledge.

Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important—if not the most important—optimizations. Recent register allocation algorithms are based on a technique called *graph coloring*. The basic idea behind graph coloring is to construct a graph representing the possible candidates for allocation to a register and then to use the graph to allocate registers. Although the problem of coloring a graph is NP-complete, there are heuristic algorithms that work well in practice.

Graph coloring works best when there are at least 16 (and preferably more) general-purpose registers available for global allocation for integer variables and additional registers for floating point. Unfortunately, graph coloring does not work very well when the number of registers is small because the heuristic algorithms for coloring the graph are likely to fail. The emphasis in the approach is to achieve 100% allocation of active variables.

It is sometimes difficult to separate some of the simpler optimizations—local and machine-dependent optimizations—from transformations done in the code generator. Examples of typical optimizations are given in Figure 2.19. The last column of Figure 2.19 indicates the frequency with which the listed optimizing transforms were applied to the source program. The effect of various optimizations on instructions executed for two programs is shown in Figure 2.20.

The Impact of Compiler Technology on the Architect's Decisions

The interaction of compilers and high-level languages significantly affects how programs use an instruction set architecture. There are two important questions: How are variables allocated and addressed? How many registers are needed to allocate variables appropriately? To address these questions, we must look at the three separate areas in which current high-level languages allocate their data:

- The *stack* is used to allocate local variables. The stack is grown and shrunk on procedure call or return, respectively. Objects on the stack are addressed relative to the stack pointer and are primarily scalars (single variables) rather than arrays. The stack is used for activation records, *not* as a stack for evaluating expressions. Hence values are almost never pushed or popped on the stack.

Optimization name	Explanation	Percentage of the total number of optimizing transforms
High-level	**At or near the source level; machine-independent**	
Procedure integration	Replace procedure call by procedure body	N.M.
Local	**Within straight-line code**	
Common subexpression elimination	Replace two instances of the same computation by single copy	18%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M.
Global	**Across a branch**	
Global common subexpression elimination	Same as local, but this version crosses branches	13%
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., *A* = *X*) with *X*	11%
Code motion	Remove code from a loop that computes same value each iteration of the loop	16%
Induction variable elimination	Simplify/eliminate array-addressing calculations within loops	2%
Machine-dependent	**Depends on machine knowledge**	
Strength reduction	Many examples, such as replace multiply by a constant with adds and shifts	N.M.
Pipeline scheduling	Reorder instructions to improve pipeline performance	N.M.
Branch offset optimization	Choose the shortest branch displacement that reaches target	N.M.

FIGURE 2.19 Major types of optimizations and examples in each class. The third column lists the static frequency with which some of the common optimizations are applied in a set of 12 small FORTRAN and Pascal programs. The percentage is the portion of the static optimizations that are of the specified type. These data tell us about the relative frequency of occurrence of various optimizations. There are nine local and global optimizations done by the compiler included in the measurement. Six of these optimizations are covered in the figure, and the remaining three account for 18% of the total static occurrences. The abbreviation *N.M.* means that the number of occurrences of that optimization was not measured. Machine-dependent optimizations are usually done in a code generator, and none of those was measured in this experiment. Data from Chow [1983] (collected using the Stanford UCODE compiler).

- The *global data area* is used to allocate statically declared objects, such as global variables and constants. A large percentage of these objects are arrays or other aggregate data structures.
- The *heap* is used to allocate dynamic objects that do not adhere to a stack discipline. Objects in the heap are accessed with pointers and are typically not scalars.

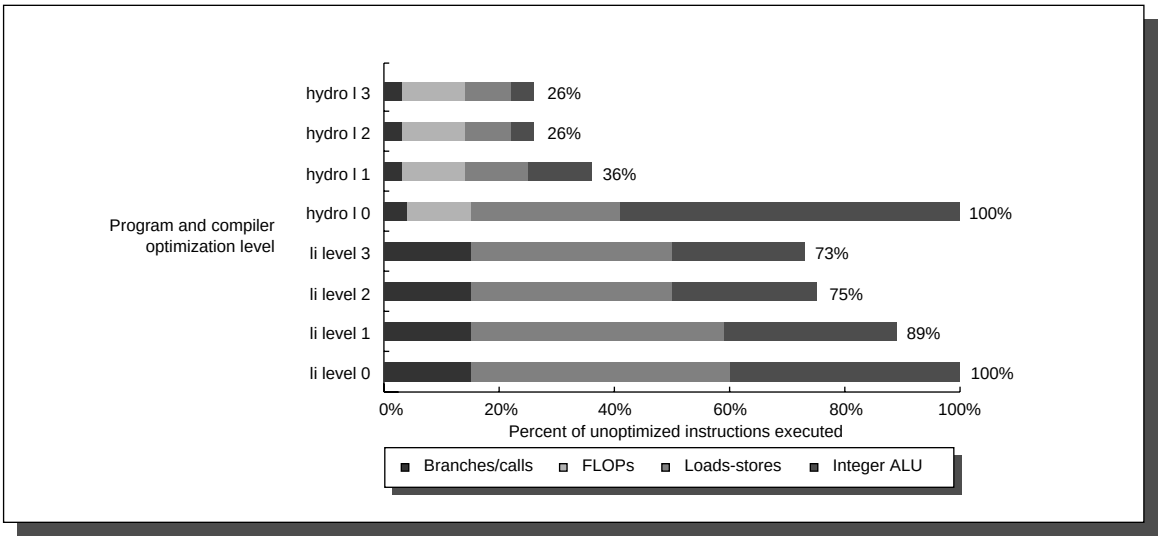

FIGURE 2.20 Change in instruction count for the programs hydro2d and li from the SPEC92 as compiler optimization levels vary. Level 0 is the same as unoptimized code. These experiments were perfomed on the MIPS compilers. Level 1 includes local optimizations, code scheduling, and local register allocation. Level 2 includes global optimizations, loop transformations (*software pipelining*), and global register allocation. Level 3 adds procedure integration.

Register allocation is much more effective for stack-allocated objects than for global variables, and register allocation is essentially impossible for heap-allocated objects because they are accessed with pointers. Global variables and some stack variables are impossible to allocate because they are *aliased*, which means that there are multiple ways to refer to the address of a variable, making it illegal to put it into a register. (Most heap variables are effectively aliased for today's compiler technology.) For example, consider the following code sequence, where & returns the address of a variable and * dereferences a pointer:

```
p = &a        -- gets address of a in p
a = ...       -- assigns to a directly
*p = ...      -- uses p to assign to a
...a...       -- accesses a
```

The variable a could not be register allocated across the assignment to *p without generating incorrect code. Aliasing causes a substantial problem because it is often difficult or impossible to decide what objects a pointer may refer to. A compiler must be conservative; many compilers will not allocate *any* local variables of a procedure in a register when there is a pointer that may refer to *one* of the local variables.

How the Architect Can Help the Compiler Writer

Today, the complexity of a compiler does not come from translating simple statements like A = B + C. Most programs are *locally simple*, and simple translations work fine. Rather, complexity arises because programs are large and globally complex in their interactions, and because the structure of compilers means that decisions must be made about what code sequence is best one step at a time.

Compiler writers often are working under their own corollary of a basic principle in architecture: *Make the frequent cases fast and the rare case correct.* That is, if we know which cases are frequent and which are rare, and if generating code for both is straightforward, then the quality of the code for the rare case may not be very important—but it must be correct!

Some instruction set properties help the compiler writer. These properties should not be thought of as hard and fast rules, but rather as guidelines that will make it easier to write a compiler that will generate efficient and correct code.

1. *Regularity*—Whenever it makes sense, the three primary components of an instruction set—the operations, the data types, and the addressing modes—should be *orthogonal*. Two aspects of an architecture are said to be orthogonal if they are independent. For example, the operations and addressing modes are orthogonal if for every operation to which a certain addressing mode can be applied, all addressing modes are applicable. This helps simplify code generation and is particularly important when the decision about what code to generate is split into two passes in the compiler. A good counterexample of this property is restricting what registers can be used for a certain class of instructions. This can result in the compiler finding itself with lots of available registers, but none of the right kind!

2. *Provide primitives, not solutions*—Special features that "match" a language construct are often unusable. Attempts to support high-level languages may work only with one language, or do more or less than is required for a correct and efficient implementation of the language. Some examples of how these attempts have failed are given in section 2.9.

3. *Simplify trade-offs among alternatives*—One of the toughest jobs a compiler writer has is figuring out what instruction sequence will be best for every segment of code that arises. In earlier days, instruction counts or total code size might have been good metrics, but—as we saw in the last chapter—this is no longer true. With caches and pipelining, the trade-offs have become very complex. Anything the designer can do to help the compiler writer understand the costs of alternative code sequences would help improve the code. One of the most difficult instances of complex trade-offs occurs in a register-memory architecture in deciding how many times a variable should be referenced before it is cheaper to load it into a register. This threshold is hard to compute and, in fact, may vary among models of the same architecture.

4. *Provide instructions that bind the quantities known at compile time as constants*—A compiler writer hates the thought of the machine interpreting at runtime a value that was known at compile time. Good counterexamples of this principle include instructions that interpret values that were fixed at compile time. For instance, the VAX procedure call instruction (`calls`) dynamically interprets a mask saying what registers to save on a call, but the mask is fixed at compile time. However, in some cases it may not be known by the caller whether separate compilation was used.

Summary: The Role of Compilers

This section leads to several recommendations. First, we expect a new instruction set architecture to have at least 16 general-purpose registers—not counting separate registers for floating-point numbers—to simplify allocation of registers using graph coloring. The advice on orthogonality suggests that all supported addressing modes apply to all instructions that transfer data. Finally, the last three pieces of advice of the last subsection—provide primitives instead of solutions, simplify trade-offs between alternatives, don't bind constants at runtime—all suggest that it is better to err on the side of simplicity. In other words, understand that less is more in the design of an instruction set.

2.8 Putting It All Together: The DLX Architecture

In many places throughout this book we will have occasion to refer to a computer's "machine language." The machine we use is a mythical computer called "MIX." MIX is very much like nearly every computer in existence, except that it is, perhaps, nicer … MIX is the world's first polyunsaturated computer. Like most machines, it has an identifying number—the 1009. This number was found by taking 16 actual computers which are very similar to MIX and on which MIX can be easily simulated, then averaging their number with equal weight:

$$\lfloor (360 + 650 + 709 + 7070 + U3 + SS80 + 1107 + 1604 + G20 + B220 + S2000 + 920 + 601 + H800 + PDP\text{-}4 + II)/16 \rfloor = 1009.$$

The same number may be obtained in a simpler way by taking Roman numerals.

Donald Knuth, *The Art of Computer Programming, Volume I: Fundamental Algorithms*

In this section we will describe a simple load-store architecture called DLX (pronounced "Deluxe"). The authors believe DLX to be the world's second polyunsaturated computer—the average of a number of recent experimental and commercial machines that are very similar in philosophy to DLX. Like Knuth,

we derived the name of our machine from an average expressed in Roman numerals:

(AMD 29K, DECstation 3100, HP 850, IBM 801, Intel i860, MIPS M/120A, MIPS M/1000, Motorola 88K, RISC I, SGI 4D/60, SPARCstation-1, Sun-4/110, Sun-4/260) / 13 = 560 = DLX.

The instruction set architecture of DLX and its ancestors was based on observations similar to those covered in the last sections. (In section 2.11 we discuss how and why these architectures became popular.) Reviewing our expectations from each section:

- *Section 2.2*—Use general-purpose registers with a load-store architecture.
- *Section 2.3*—Support these addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred.
- *Section 2.4*—Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8 bits long), jump, call, and return.
- *Section 2.5*—Support these data sizes and types: 8-, 16-, and 32-bit integers and 64-bit IEEE 754 floating-point numbers.
- *Section 2.6*—Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size.
- *Section 2.7*—Provide at least 16 general-purpose registers plus separate floating-point registers, be sure all addressing modes apply to all data transfer instructions, and aim for a minimalist instruction set.

We introduce DLX by showing how it follows these recommendations. Like most recent machines, DLX emphasizes

- A simple load-store instruction set
- Design for pipelining efficiency, including a fixed instruction set encoding (discussed in Chapter 3)
- Efficiency as a compiler target

DLX provides a good architectural model for study, not only because of the recent popularity of this type of machine, but also because it is an easy architecture to understand. We will use this architecture again in Chapters 3 and 4, and it forms the basis for a number of exercises and programming projects.

Registers for DLX

DLX has 32 32-bit general-purpose registers (GPRs), named R0, R1, …, R31. Additionally, there is a set of floating-point registers (FPRs), which can be used as 32 single-precision (32-bit) registers or as even-odd pairs holding double-precision values. Thus, the 64-bit floating-point registers are named F0, F2, ..., F28, F30. Both single- and double-precision floating-point operations (32-bit and 64-bit) are provided.

The value of R0 is always 0. We shall see later how we can use this register to synthesize a variety of useful operations from a simple instruction set.

A few special registers can be transferred to and from the integer registers. An example is the floating-point status register, used to hold information about the results of floating-point operations. There are also instructions for moving between a FPR and a GPR.

Data types for DLX

The data types are 8-bit bytes, 16-bit half words, and 32-bit words for integer data and 32-bit single precision and 64-bit double precision for floating point. Half words were added to the minimal set of recommended data types supported because they are found in languages like C and popular in some programs, such as the operating systems, concerned about size of data structures. They will also become more popular as Unicode becomes more widely used. Single-precision floating-point operands were added for similar reasons. (Remember the early warning that you should measure many more programs before designing an instruction set.)

The DLX operations work on 32-bit integers and 32- or 64-bit floating point. Bytes and half words are loaded into registers with either zeros or the sign bit replicated to fill the 32 bits of the registers. Once loaded, they are operated on with the 32-bit integer operations.

Addressing modes for DLX data transfers

The only data addressing modes are immediate and displacement, both with 16-bit fields. Register deferred is accomplished simply by placing 0 in the 16-bit displacement field, and absolute addressing with a 16-bit field is accomplished by using register 0 as the base register. This gives us four effective modes, although only two are supported in the architecture.

DLX memory is byte addressable in Big Endian mode with a 32-bit address. As it is a load-store architecture, all memory references are through loads or stores between memory and either the GPRs or the FPRs. Supporting the data types mentioned above, memory accesses involving the GPRs can be to a byte, to a half word, or to a word. The FPRs may be loaded and stored with single-precision or double-precision words (using a pair of registers for DP). All memory accesses must be aligned.

DLX Instruction Format

Since DLX has just two addressing modes, these can be encoded into the opcode. Following the advice on making the machine easy to pipeline and decode, all instructions are 32 bits with a 6-bit primary opcode. Figure 2.21 shows the instruction layout. These formats are simple while providing 16-bit fields for displacement addressing, immediate constants, or PC-relative branch addresses.

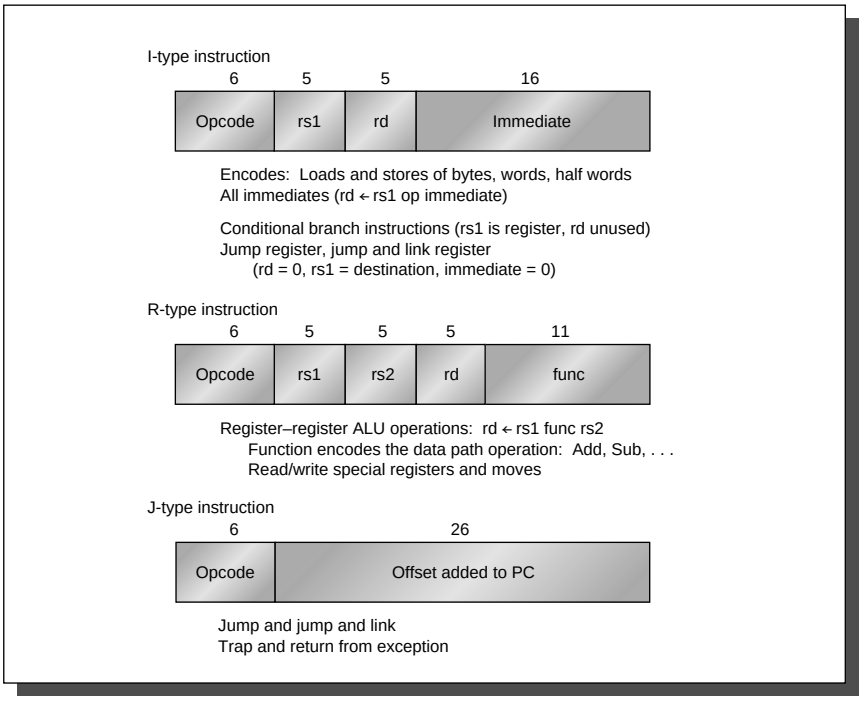

FIGURE 2.21 Instruction layout for DLX. All instructions are encoded in one of three types.

DLX Operations

DLX supports the list of simple operations recommended above plus a few others. There are four broad classes of instructions: loads and stores, ALU operations, branches and jumps, and floating-point operations.

Any of the general-purpose or floating-point registers may be loaded or stored, except that loading R0 has no effect. Single-precision floating-point numbers occupy a single floating-point register, while double-precision values occupy a pair. Conversions between single and double precision must be done explicitly. The floating-point format is IEEE 754 (see Appendix A). Figure 2.22 gives examples

Example instruction	Instruction name	Meaning
LW R1,30(R2)	Load word	Regs[R1]$\leftarrow_{32}$ Mem[30+Regs[R2]]
LW R1,1000(R0)	Load word	Regs[R1]$\leftarrow_{32}$ Mem[1000+0]
LB R1,40(R3)	Load byte	Regs[R1]$\leftarrow_{32}$ (Mem[40+Regs[R3]]$_0$)^{24} ## Mem[40+Regs[R3]]
LBU R1,40(R3)	Load byte unsigned	Regs[R1]$\leftarrow_{32}$ 0^{24} ## Mem[40+Regs[R3]]
LH R1,40(R3)	Load half word	Regs[R1]$\leftarrow_{32}$ (Mem[40+Regs[R3]]$_0$)^{16} ## Mem[40+Regs[R3]]##Mem[41+Regs[R3]]
LF F0,50(R3)	Load float	Regs[F0]$\leftarrow_{32}$ Mem[50+Regs[R3]]
LD F0,50(R2)	Load double	Regs[F0]##Regs[F1]$\leftarrow_{64}$ Mem[50+Regs[R2]]
SW R3,500(R4)	Store word	Mem[500+Regs[R4]]$\leftarrow_{32}$ Regs[R3]
SF F0,40(R3)	Store float	Mem[40+Regs[R3]]$\leftarrow_{32}$ Regs[F0]
SD F0,40(R3)	Store double	Mem[40+Regs[R3]]$\leftarrow_{32}$ Regs[F0]; Mem[44+Regs[R3]]$\leftarrow_{32}$ Regs[F1]
SH R3,502(R2)	Store half	Mem[502+Regs[R2]]$\leftarrow_{16}$ Regs[R3]$_{16..31}$
SB R2,41(R3)	Store byte	Mem[41+Regs[R3]]$\leftarrow_{8}$ Regs[R2]$_{24..31}$

FIGURE 2.22 The load and store instructions in DLX. All use a single addressing mode and require that the memory value be aligned. Of course, both loads and stores are available for all the data types shown.

of the load and store instructions. A complete list of the instructions appears in Figure 2.25 (page 104). To understand these figures we need to introduce a few additional extensions to our C description language:

- A subscript is appended to the symbol $\leftarrow$ whenever the length of the datum being transferred might not be clear. Thus, $\leftarrow_n$ means transfer an n-bit quantity. We use $x, y \leftarrow z$ to indicate that z should be transferred to x and y.
- A subscript is used to indicate selection of a bit from a field. Bits are labeled from the most-significant bit starting at 0. The subscript may be a single digit (e.g., Regs[R4]$_0$ yields the sign bit of R4) or a subrange (e.g., Regs[R3]$_{24..31}$ yields the least-significant byte of R3).
- The variable Mem, used as an array that stands for main memory, is indexed by a byte address and may transfer any number of bytes.
- A superscript is used to replicate a field (e.g., 0^{24} yields a field of zeros of length 24 bits).
- The symbol ## is used to concatenate two fields and may appear on either side of a data transfer.

A summary of the entire description language appears on the back inside cover. As an example, assuming that R8 and R10 are 32-bit registers:

$$\texttt{Regs[R10]}_{16..31} \leftarrow_{16} (\texttt{Mem[Regs[R8]]}_0)^8 \ \#\#\ \texttt{Mem[Regs[R8]]}$$

means that the byte at the memory location addressed by the contents of R8 is sign-extended to form a 16-bit quantity that is stored into the lower half of R10. (The upper half of R10 is unchanged.)

All ALU instructions are register-register instructions. The operations include simple arithmetic and logical operations: add, subtract, AND, OR, XOR, and shifts. Immediate forms of all these instructions, with a 16-bit sign-extended immediate, are provided. The operation `LHI` (load high immediate) loads the top half of a register, while setting the lower half to 0. This allows a full 32-bit constant to be built in two instructions, or a data transfer using any constant 32-bit address in one extra instruction.

As mentioned above, R0 is used to synthesize popular operations. Loading a constant is simply an add immediate where one of the source operands is R0, and a register-register move is simply an add where one of the sources is R0. (We sometimes use the mnemonic `LI`, standing for load immediate, to represent the former and the mnemonic `MOV` for the latter.)

There are also compare instructions, which compare two registers (=, ≠, <, >, ≤, ≥). If the condition is true, these instructions place a 1 in the destination register (to represent true); otherwise they place the value 0. Because these operations "set" a register, they are called set-equal, set-not-equal, set-less-than, and so on. There are also immediate forms of these compares. Figure 2.23 gives some examples of the arithmetic/logical instructions.

Example instruction	Instruction name	Meaning
`ADD R1,R2,R3`	Add	`Regs[R1]`←`Regs[R2]+Regs[R3]`
`ADDI R1,R2,#3`	Add immediate	`Regs[R1]`←`Regs[R2]+3`
`LHI R1,#42`	Load high immediate	`Regs[R1]`←$42\#\#0^{16}$
`SLLI R1,R2,#5`	Shift left logical immediate	`Regs[R1]`←`Regs[R2]<<5`
`SLT R1,R2,R3`	Set less than	`if (Regs[R2]<Regs[R3]) Regs[R1]`←`1 else Regs[R1]`←`0`

FIGURE 2.23 Examples of arithmetic/logical instructions on DLX, both with and without immediates.

Control is handled through a set of jumps and a set of branches. The four jump instructions are differentiated by the two ways to specify the destination address and by whether or not a link is made. Two jumps use a 26-bit signed offset added

to the program counter (of the instruction sequentially following the jump) to determine the destination address; the other two jump instructions specify a register that contains the destination address. There are two flavors of jumps: plain jump, and jump and link (used for procedure calls). The latter places the return address—the address of the next sequential instruction—in R31.

All branches are conditional. The branch condition is specified by the instruction, which may test the register source for zero or nonzero; the register may contain a data value or the result of a compare. The branch target address is specified with a 16-bit signed offset that is added to the program counter, which is pointing to the next sequential instruction. Figure 2.24 gives some typical branch and jump instructions. There is also a branch to test the floating-point status register for floating-point conditional branches, described below.

Example instruction	Instruction name	Meaning
J name	Jump	PC←name; ((PC+4)–2^{25}) ≤ name < ((PC+4)+2^{25})
JAL name	Jump and link	Regs[R31]←PC+4; PC←name; ((PC+4)–2^{25}) ≤ name < ((PC+4)+2^{25})
JALR R2	Jump and link register	Regs[R31]←PC+4; PC←Regs[R2]
JR R3	Jump register	PC←Regs[R3]
BEQZ R4,name	Branch equal zero	if (Regs[R4]==0) PC←name; ((PC+4)–2^{15}) ≤ name < ((PC+4)+2^{15})
BNEZ R4,name	Branch not equal zero	if (Regs[R4]!=0) PC←name; ((PC+4)–2^{15}) ≤ name < ((PC+4)+2^{15})

FIGURE 2.24 Typical control-flow instructions in DLX. All control instructions, except jumps to an address in a register, are PC-relative. If the register operand is R0, BEQZ will always branch, but the compiler will usually prefer to use a jump with a longer offset over this "unconditional branch."

Floating-point instructions manipulate the floating-point registers and indicate whether the operation to be performed is single or double precision. The operations MOVF and MOVD copy a single-precision (MOVF) or double-precision (MOVD) floating-point register to another register of the same type. The operations MOVFP2I and MOVI2FP move data between a single floating-point register and an integer register; moving a double-precision value to two integer registers requires two instructions. Integer multiply and divide that work on 32-bit floating-point registers are also provided, as are conversions from integer to floating point and vice versa.

The floating-point operations are add, subtract, multiply, and divide; a suffix D is used for double precision and a suffix F is used for single precision (e.g., ADDD, ADDF, SUBD, SUBF, MULTD, MULTF, DIVD, DIVF). Floating-point compares set a

bit in the special floating-point status register that can be tested with a pair of branches: BFPT and BFPF, branch floating-point true and branch floating-point false.

One slightly unusual DLX characteristic is that it uses the floating-point unit for integer multiplies and divides. As we shall see in Chapters 3 and 4, the control for the slower floating-point operations is much more complicated than for integer addition and subtraction. Since the floating-point unit already handles floating point multiply and divide, it is not much harder for it to perform the relatively slow operations of integer multiply and divide. Hence DLX requires that operands to be multiplied or divided be placed in floating-point registers.

Figure 2.25 contains a list of all DLX operations and their meaning. To give an idea which instructions are popular, Figure 2.26 shows the frequency of instructions and instruction classes for five SPECint92 programs and Figure 2.27 shows the same data for five SPECfp92 programs. To give a more intuitive feeling, Figures 2.28 and 2.29 show the data graphically for all instructions that are responsible on average for more than 1% of the instructions executed.

Effectiveness of DLX

It would seem that an architecture with simple instruction formats, simple address modes, and simple operations would be slow, in part because it has to execute more instructions than more sophisticated designs. The performance equation from the last chapter reminds us that execution time is a function of more than just instruction count:

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

To see whether reduction in instruction count is offset by increases in CPI or clock cycle time, we need to compare DLX to a sophisticated alternative.

One example of a sophisticated instruction set architecture is the VAX. In the mid 1970s, when the VAX was designed, the prevailing philosophy was to create instruction sets that were close to programming languages to simplify compilers. For example, because programming languages had loops, instruction sets should have loop instructions, not just simple conditional branches; they needed call instructions that saved registers, not just simple jump and links; they needed case instructions, not just jump indirect; and so on. Following similar arguments, the VAX provided a large set of addressing modes and made sure that all addressing modes worked with all operations. Another prevailing philosophy was to minimize code size. Recall that DRAMs have grown in capacity by a factor of four every three years; thus in the mid 1970s DRAM chips contained less than 1/1000 the capacity of today's DRAMs, so code space was also critical. Code space was

Instruction type/opcode	Instruction meaning
Data transfers	**Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR**
LB,LBU,SB	Load byte, load byte unsigned, store byte
LH,LHU,SH	Load half word, load half word unsigned, store half word
LW,SW	Load word, store word (to/from integer registers)
LF,LD,SF,SD	Load SP float, load DP float, store SP float, store DP float
MOVI2S, MOVS2I	Move from/to GPR to/from a special register
MOVF, MOVD	Copy one FP register or a DP pair to another register or pair
MOVFP2I,MOVI2FP	Move 32 bits from/to FP registers to/from integer registers
Arithmetic/logical	**Operations on integer or logical data in GPRs; signed arithmetic trap on overflow**
ADD,ADDI,ADDU, ADDUI	Add, add immediate (all immediates are 16 bits); signed and unsigned
SUB,SUBI,SUBU, SUBUI	Subtract, subtract immediate; signed and unsigned
MULT,MULTU,DIV,DIVU	Multiply and divide, signed and unsigned; operands must be FP registers; all operations take and yield 32-bit values
AND,ANDI	And, and immediate
OR,ORI,XOR,XORI	Or, or immediate, exclusive or, exclusive or immediate
LHI	Load high immediate—loads upper half of register with immediate
SLL, SRL, SRA, SLLI, SRLI, SRAI	Shifts: both immediate (S__I) and variable form (S__); shifts are shift left logical, right logical, right arithmetic
S__,S__I	Set conditional: "__" may be LT,GT,LE,GE,EQ,NE
Control	**Conditional branches and jumps; PC-relative or through register**
BEQZ,BNEZ	Branch GPR equal/not equal to zero; 16-bit offset from PC+4
BFPT,BFPF	Test comparison bit in the FP status register and branch; 16-bit offset from PC+4
J, JR	Jumps: 26-bit offset from PC+4 (J) or target in register (JR)
JAL, JALR	Jump and link: save PC+4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
RFE	Return to user code from an exception; restore user mode
Floating point	**FP operations on DP and SP formats**
ADDD,ADDF	Add DP, SP numbers
SUBD,SUBF	Subtract DP, SP numbers
MULTD,MULTF	Multiply DP, SP floating point
DIVD,DIVF	Divide DP, SP floating point
CVTF2D, CVTF2I, CVTD2F, CVTD2I, CVTI2F, CVTI2D	Convert instructions: CVTx2y converts from type x to type y, where x and y are I (integer), D (double precision), or F (single precision). Both operands are FPRs.
__D,__F	DP and SP compares: "__" = LT,GT,LE,GE,EQ,NE; sets bit in FP status register

FIGURE 2.25 Complete list of the instructions in DLX. The formats of these instructions are shown in Figure 2.21. SP = single precision; DP = double precision. This list can also be found on the page preceding the back inside cover.

Instruction	compress	eqntott	espresso	gcc (cc1)	li	Integer average
load	19.8%	30.6%	20.9%	22.8%	31.3%	26%
store	5.6%	0.6%	5.1%	14.3%	16.7%	9%
add	14.4%	8.5%	23.8%	14.6%	11.1%	14%
sub	1.8%	0.3%		0.5%		0%
mul				0.1%		0%
div						0%
compare	15.4%	26.5%	8.3%	12.4%	5.4%	14%
load imm	8.1%	1.5%	1.3%	6.8%	2.4%	4%
cond branch	17.4%	24.0%	15.0%	11.5%	14.6%	17%
jump	1.5%	0.9%	0.5%	1.3%	1.8%	1%
call	0.1%	0.5%	0.4%	1.1%	3.1%	1%
return, jmp ind	0.1%	0.5%	0.5%	1.5%	3.5%	1%
shift	6.5%	0.3%	7.0%	6.2%	0.7%	4%
and	2.1%	0.1%	9.4%	1.6%	2.1%	3%
or	6.0%	5.5%	4.8%	4.2%	6.2%	5%
other (xor, not)	1.0%		2.0%	0.5%	0.1%	1%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
compare FP						0%
mov reg-reg FP						0%
other FP						0%

FIGURE 2.26 DLX instruction mix for five SPECint92 programs. Note that integer register-register move instructions are included in the add instruction. Blank entries have the value 0.0%.

de-emphasized in fixed-length instruction sets like DLX. For example, DLX address fields always use 16 bits, even when the address is very small. In contrast, the VAX allows instructions to be a variable number of bytes, so there is little wasted space in address fields.

Designers of VAX machines later performed a quantitative comparison of VAX and a DLX-like machine for implementations with comparable organizations. Their choices were the VAX 8700 and the MIPS M2000. The differing

Instruction	doduc	ear	hydro2d	mdljdp2	su2cor	FP average
load	1.4%	0.2%	0.1%	1.1%	3.6%	1%
store	1.3%	0.1%		0.1%	1.3%	1%
add	13.6%	13.6%	10.9%	4.7%	9.7%	11%
sub	0.3%		0.2%		0.7%	0%
mul						0%
div						0%
compare	3.2%	3.1%	1.2%	0.3%	1.3%	2%
load imm	2.2%		0.2%	2.2%	0.9%	1%
cond branch	8.0%	10.1%	11.7%	9.3%	2.6%	8%
jump	0.9%	0.4%		0.4%	0.1%	0%
call	0.5%	1.9%			0.3%	1%
return, jmp ind	0.6%	1.9%			0.3%	1%
shift	2.0%	0.2%	2.4%	1.3%	2.3%	2%
and	0.4%	0.1%			0.3%	0%
or		0.2%	0.1%	0.1%	0.1%	0%
other (xor, not)						0%
load FP	23.3%	19.8%	24.1%	25.9%	21.6%	23%
store FP	5.7%	11.4%	9.9%	10.0%	9.8%	9%
add FP	8.8%	7.3%	3.6%	8.5%	12.4%	8%
sub FP	3.8%	3.2%	7.9%	10.4%	5.9%	6%
mul FP	12.0%	9.6%	9.4%	13.9%	21.6%	13%
div FP	2.3%		1.6%	0.9%	0.7%	1%
compare FP	4.2%	6.4%	10.4%	9.3%	0.8%	6%
mov reg-reg FP	2.1%	1.8%	5.2%	0.9%	1.9%	2%
other FP	2.4%	8.4%	0.2%	0.2%	1.2%	2%

FIGURE 2.27 DLX instruction mix for five programs from SPECfp92. Note that integer register-register move instructions are included in the add instruction. Blank entries have the value 0.0%.

goals for VAX and MIPS have led to very different architectures. The VAX goals, simple compilers and code density, led to powerful addressing modes, powerful instructions, efficient instruction encoding, and few registers. The MIPS goals were high performance via pipelining, ease of hardware implementation, and compatibility with highly optimizing compilers. These goals led to simple instructions, simple addressing modes, fixed-length instruction formats, and a large number of registers.

Figure 2.30 shows the ratio of the number of instructions executed, the ratio of CPIs, and the ratio of performance measured in clock cycles. Since the organizations

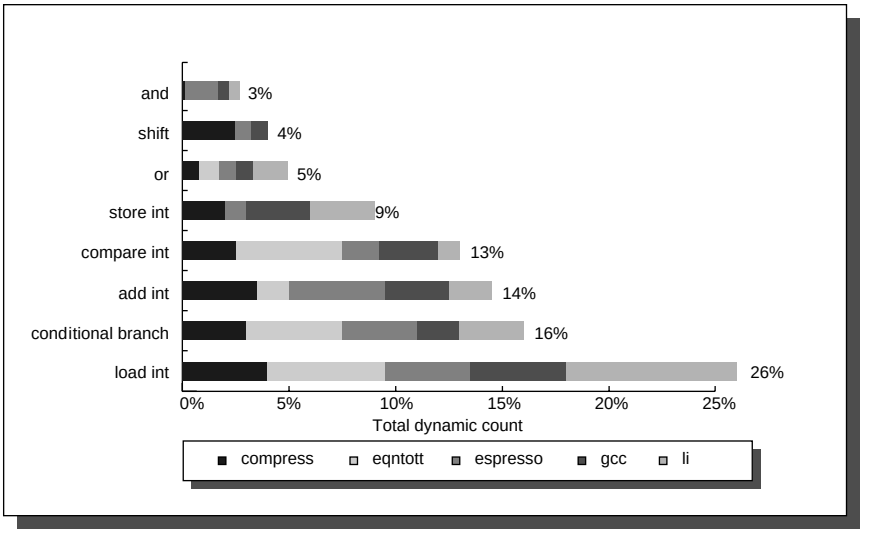

FIGURE 2.28 Graphical display of instructions executed of the five programs from SPECint92 in Figure 2.26. These instruction classes collectively are responsible on average for 92% of instructions executed.

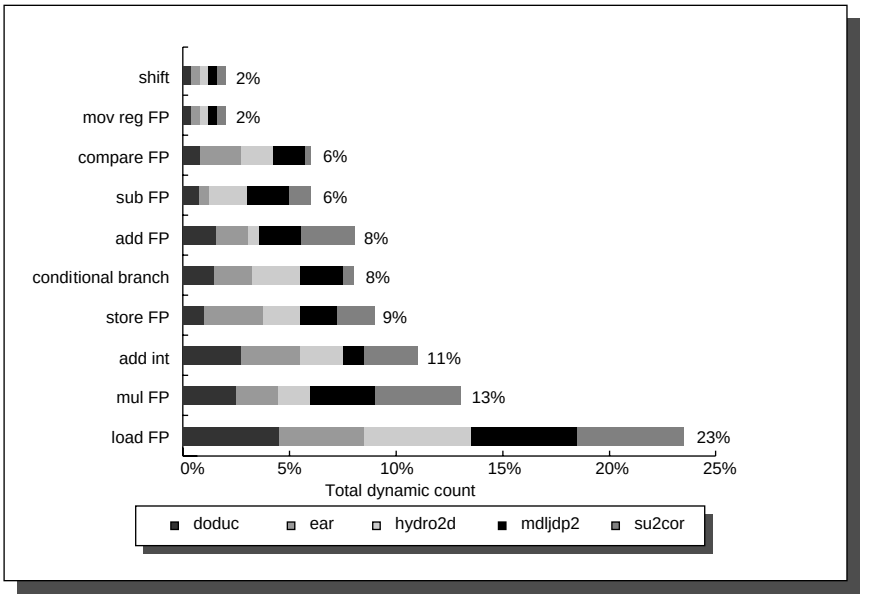

FIGURE 2.29 Graphical display of instructions executed of the five programs from SPECfp92 in Figure 2.27. These instruction classes collectively are responsible on average for just under 90% of instructions executed.

were similar, clock cycle times were assumed to be the same. MIPS executes about twice as many instructions as the VAX, while the CPI for the VAX is about six times larger than that for the MIPS. Hence the MIPS M2000 has almost three times the performance of the VAX 8700. Furthermore, much less hardware is needed to build the MIPS CPU than the VAX CPU. This cost/performance gap is the reason the company that used to make the VAX has dropped it and is now making a machine similar to DLX.

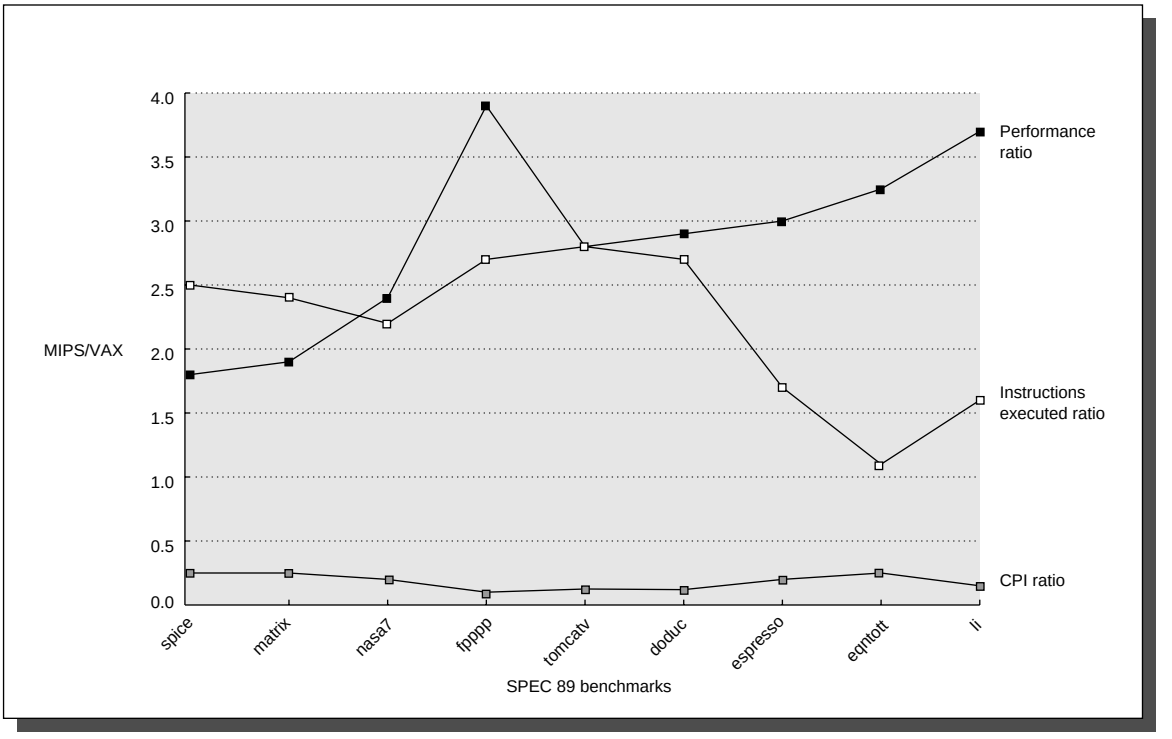

FIGURE 2.30 Ratio of MIPS M2000 to VAX 8700 in instructions executed and performance in clock cycles using SPEC89 programs. On average, MIPS executes a little over twice as many instructions as the VAX, but the CPI for the VAX is almost six times the MIPS CPI, yielding almost a threefold performance advantage. (Based on data from Bhandarkar and Clark [1991].)

2.9 Fallacies and Pitfalls

Time and again architects have tripped on common, but erroneous, beliefs. In this section we look at a few of them.

Pitfall: Designing a "high-level" instruction set feature specifically oriented to supporting a high-level language structure.

Attempts to incorporate high-level language features in the instruction set have led architects to provide powerful instructions with a wide range of flexibility. But often these instructions do more work than is required in the frequent case, or they don't exactly match the requirements of the language. Many such efforts have been aimed at eliminating what in the 1970s was called the *semantic gap*. Although the idea is to supplement the instruction set with additions that bring the hardware up to the level of the language, the additions can generate what Wulf [1981] has called a *semantic clash*:

... by giving too much semantic content to the instruction, the machine designer made it possible to use the instruction only in limited contexts. [p. 43]

More often the instructions are simply overkill—they are too general for the most frequent case, resulting in unneeded work and a slower instruction. Again, the VAX `CALLS` is a good example. `CALLS` uses a callee-save strategy (the registers to be saved are specified by the callee) *but* the saving is done by the call instruction in the caller. The `CALLS` instruction begins with the arguments pushed on the stack, and then takes the following steps:

1. Align the stack if needed.
2. Push the argument count on the stack.
3. Save the registers indicated by the procedure call mask on the stack (as mentioned in section 2.7). The mask is kept in the called procedure's code—this permits callee save to be done by the caller even with separate compilation.
4. Push the return address on the stack, then push the top and base of stack pointers for the activation record.
5. Clear the condition codes, which sets the trap enables to a known state.
6. Push a word for status information and a zero word on the stack.
7. Update the two stack pointers.
8. Branch to the first instruction of the procedure.

The vast majority of calls in real programs do not require this amount of overhead. Most procedures know their argument counts, and a much faster linkage convention can be established using registers to pass arguments rather than the stack. Furthermore, the `CALLS` instruction forces two registers to be used for linkage, while many languages require only one linkage register. Many attempts to support procedure call and activation stack management have failed to be useful, either because they do not match the language needs or because they are too general and hence too expensive to use.

The VAX designers provided a simpler instruction, JSB, that is much faster since it only pushes the return PC on the stack and jumps to the procedure. However, most VAX compilers use the more costly CALLS instructions. The call instructions were included in the architecture to standardize the procedure linkage convention. Other machines have standardized their calling convention by agreement among compiler writers and without requiring the overhead of a complex, very general-procedure call instruction.

Fallacy: There is such a thing as a typical program.

Many people would like to believe that there is a single "typical" program that could be used to design an optimal instruction set. For example, see the synthetic benchmarks discussed in Chapter 1. The data in this chapter clearly show that programs can vary significantly in how they use an instruction set. For example, Figure 2.31 shows the mix of data transfer sizes for four of the SPEC92 programs: It would be hard to say what is typical from these four programs. The variations are even larger on an instruction set that supports a class of applications, such as decimal instructions, that are unused by other applications.

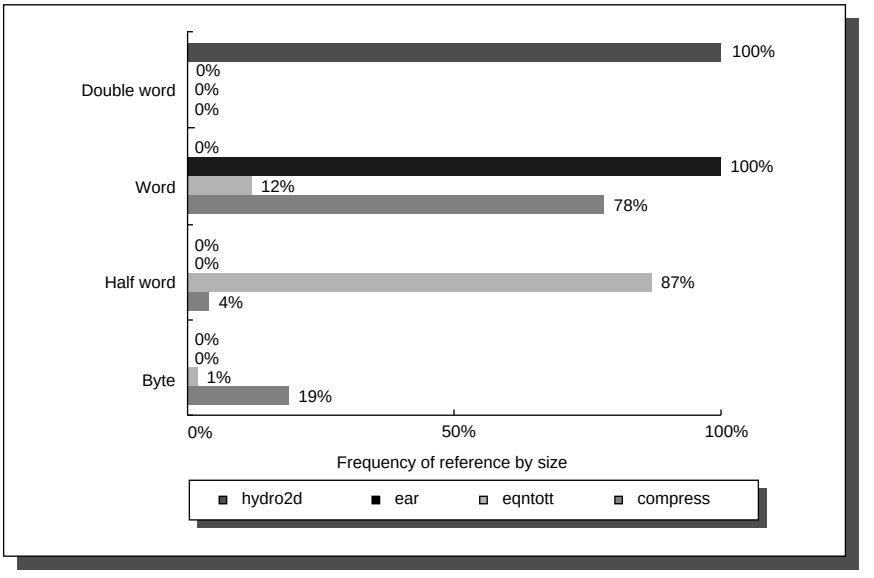

FIGURE 2.31 Data reference size of four programs from SPEC92. Although you can calculate an average size, it would be hard to claim the average is typical of programs.

Fallacy: An architecture with flaws cannot be successful.

The 80x86 provides a dramatic example: The architecture is one only its creators could love (see Appendix D). Succeeding generations of Intel engineers have

tried to correct unpopular architectural decisions made in designing the 80x86. For example, the 80x86 supports segmentation, whereas all others picked paging; the 80x86 uses extended accumulators for integer data, but other machines use general-purpose registers; and it uses a stack for floating-point data when everyone else abandoned execution stacks long before. Despite these major difficulties, the 80x86 architecture—because of its selection as the microprocessor in the IBM PC—has been enormously successful.

Fallacy: You can design a flawless architecture.

All architecture design involves trade-offs made in the context of a set of hardware and software technologies. Over time those technologies are likely to change, and decisions that may have been correct at the time they were made look like mistakes. For example, in 1975 the VAX designers overemphasized the importance of code-size efficiency, underestimating how important ease of decoding and pipelining would be 10 years later. Almost all architectures eventually succumb to the lack of sufficient address space. However, avoiding this problem in the long run would probably mean compromising the efficiency of the architecture in the short run.

2.10 Concluding Remarks

The earliest architectures were limited in their instruction sets by the hardware technology of that time. As soon as the hardware technology permitted, architects began looking for ways to support high-level languages. This search led to three distinct periods of thought about how to support programs efficiently. In the 1960s, stack architectures became popular. They were viewed as being a good match for high-level languages—and they probably were, given the compiler technology of the day. In the 1970s, the main concern of architects was how to reduce software costs. This concern was met primarily by replacing software with hardware, or by providing high-level architectures that could simplify the task of software designers. The result was both the high-level-language computer architecture movement and powerful architectures like the VAX, which has a large number of addressing modes, multiple data types, and a highly orthogonal architecture. In the 1980s, more sophisticated compiler technology and a renewed emphasis on machine performance saw a return to simpler architectures, based mainly on the load-store style of machine.

Today, there is widespread agreement on instruction set design. However, in the next decade we expect to see change in the following areas:

- The 32-bit address instruction sets are being extended to 64-bit addresses, expanding the width of the registers (among other things) to 64 bits. Appendix C gives three examples of architectures that have gone from 32 bits to 64 bits.

- Given the popularity of software for the 80x86 architecture, many companies are looking to see if changes to load-store instruction sets can significantly improve performance when emulating the 80x86 architecture.
- In the next two chapters we will see that conditional branches can limit the performance of aggressive computer designs. Hence there is interest in replacing conditional branches with conditional completion of operations, such as conditional move (see Chapter 4).
- Chapter 5 explains the increasing role of memory hierarchy in performance of machines, with a cache miss on some machines taking almost as many instruction times as page faults took on earlier machines. Hence there are investigations into hiding the cost of cache misses by prefetching and by allowing caches and CPUs to proceed while servicing a miss (see Chapter 5).
- Appendix A describes new operations to enhance floating-point performance, such as operations that perform a multiply and an add. Support for quadruple precision, at least for data transfer, may also be coming down the line.

Between 1970 and 1985 many thought the primary job of the computer architect was the design of instruction sets. As a result, textbooks of that era emphasize instruction set design, much as computer architecture textbooks of the 1950s and 1960s emphasized computer arithmetic. The educated architect was expected to have strong opinions about the strengths and especially the weaknesses of the popular machines. The importance of binary compatibility in quashing innovations in instruction set design was unappreciated by many researchers and textbook writers, giving the impression that many architects would get a chance to design an instruction set.

The definition of computer architecture today has been expanded to include design and evaluation of the full computer system—not just the definition of the instruction set—and hence there are plenty of topics for the architect to study. (You may have guessed this the first time you lifted this book.) Hence the bulk of this book is on design of computers versus instruction sets. Readers interested in instruction set architecture may be satisfied by the appendices: Appendix C compares four popular load-store machines with DLX. Appendix D describes the most widely used instruction set, the Intel 80x86, and compares instruction counts for it with that of DLX for several programs.

2.11 Historical Perspective and References

One's eyebrows should rise whenever a future architecture is developed with a stack- or register-oriented instruction set. [p. 20]

Meyers [1978]

The earliest computers, including the UNIVAC I, the EDSAC, and the IAS machines, were accumulator-based machines. The simplicity of this type of machine made it the natural choice when hardware resources were very constrained. The first general-purpose register machine was the Pegasus, built by Ferranti, Ltd. in 1956. The Pegasus had eight general-purpose registers, with R0 always being zero. Block transfers loaded the eight registers from the drum.

In 1963, Burroughs delivered the B5000. The B5000 was perhaps the first machine to seriously consider software and hardware-software trade-offs. Barton and the designers at Burroughs made the B5000 a stack architecture (as described in Barton [1961]). Designed to support high-level languages such as ALGOL, this stack architecture used an operating system (MCP) written in a high-level language. The B5000 was also the first machine from a U.S. manufacturer to support virtual memory. The B6500, introduced in 1968 (and discussed in Hauck and Dent [1968]), added hardware-managed activation records. In both the B5000 and B6500, the top two elements of the stack were kept in the CPU and the rest of the stack was kept in memory. The stack architecture yielded good code density, but only provided two high-speed storage locations. The authors of both the original IBM 360 paper [Amdahl, Blaauw, and Brooks 1964] and the original PDP-11 paper [Bell et al. 1970] argue against the stack organization. They cite three major points in their arguments against stacks:

1. Performance is derived from fast registers, not the way they are used.
2. The stack organization is too limiting and requires many swap and copy operations.
3. The stack has a bottom, and when placed in slower memory there is a performance loss.

Stack-based machines fell out of favor in the late 1970s and, except for the Intel 80x86 floating-point architecture, essentially disappeared. For example, except for the 80x86, none of the machines listed in the SPEC reports uses a stack.

The term *computer architecture* was coined by IBM in the early 1960s. Amdahl, Blaauw, and Brooks [1964] used the term to refer to the programmer-visible portion of the IBM 360 instruction set. They believed that a *family* of machines of the same architecture should be able to run the same software. Although this idea may seem obvious to us today, it was quite novel at that time. IBM, even though it was the leading company in the industry, had *five* different architectures before the 360. Thus, the notion of a company standardizing on a single architecture was a radical one. The 360 designers hoped that six different divisions of IBM could be brought together by defining a common architecture. Their definition of architecture was

... the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.

The term “machine language programmer” meant that compatibility would hold, even in assembly language, while “timing independent” allowed different implementations.

The IBM 360 was the first machine to sell in large quantities with both byte addressing using 8-bit bytes and general-purpose registers. The 360 also had register-memory and limited memory-memory instructions.

In 1964, Control Data delivered the first supercomputer, the CDC 6600. As Thornton [1964] discusses, he, Cray, and the other 6600 designers were the first to explore pipelining in depth. The 6600 was the first general-purpose, load-store machine. In the 1960s, the designers of the 6600 realized the need to simplify architecture for the sake of efficient pipelining. This interaction between architectural simplicity and implementation was largely neglected during the 1970s by microprocessor and minicomputer designers, but it was brought back in the 1980s.

In the late 1960s and early 1970s, people realized that software costs were growing faster than hardware costs. McKeeman [1967] argued that compilers and operating systems were getting too big and too complex and taking too long to develop. Because of inferior compilers and the memory limitations of machines, most systems programs at the time were still written in assembly language. Many researchers proposed alleviating the software crisis by creating more powerful, software-oriented architectures. Tanenbaum [1978] studied the properties of high-level languages. Like other researchers, he found that most programs are simple. He then argued that architectures should be designed with this in mind and should optimize program size and ease of compilation. Tanenbaum proposed a stack machine with frequency-encoded instruction formats to accomplish these goals. However, as we have observed, program size does not translate directly to cost/performance, and stack machines faded out shortly after this work.

Strecker’s article [1978] discusses how he and the other architects at DEC responded to this by designing the VAX architecture. The VAX was designed to simplify compilation of high-level languages. Compiler writers had complained about the lack of complete orthogonality in the PDP-11. The VAX architecture was designed to be highly orthogonal and to allow the mapping of a high-level-language statement into a single VAX instruction. Additionally, the VAX designers tried to optimize code size because compiled programs were often too large for available memories.

The VAX-11/780 was the first machine announced in the VAX series. It is one of the most successful and heavily studied machines ever built. The cornerstone of DEC’s strategy was a single architecture, VAX, running a single operating system, VMS. This strategy worked well for over 10 years. The large number of papers reporting instruction mixes, implementation measurements, and analysis of the VAX make it an ideal case study [Wiecek 1982; Clark and Levy 1982]. Bhandarkar and Clark [1991] give a quantitative analysis of the disadvantages of the VAX versus a RISC machine, essentially a technical explanation for the demise of the VAX.

While the VAX was being designed, a more radical approach, called *high-level-language computer architecture* (HLLCA), was being advocated in the research community. This movement aimed to eliminate the gap between high-level languages and computer hardware—what Gagliardi [1973] called the "semantic gap"—by bringing the hardware "up to" the level of the programming language. Meyers [1982] provides a good summary of the arguments and a history of high-level-language computer architecture projects.

HLLCA never had a significant commercial impact. The increase in memory size on machines and the use of virtual memory eliminated the code-size problems arising from high-level languages and operating systems written in high-level languages. The combination of simpler architectures together with software offered greater performance and more flexibility at lower cost and lower complexity.

In the early 1980s, the direction of computer architecture began to swing away from providing high-level hardware support for languages. Ditzel and Patterson [1980] analyzed the difficulties encountered by the high-level-language architectures and argued that the answer lay in simpler architectures. In another paper [Patterson and Ditzel 1980], these authors first discussed the idea of reduced instruction set computers (RISC) and presented the argument for simpler architectures. Their proposal was rebutted by Clark and Strecker [1980].

The simple load-store machines from which DLX is derived are commonly called RISC architectures. The roots of RISC architectures go back to machines like the 6600, where Thornton, Cray, and others recognized the importance of instruction set simplicity in building a fast machine. Cray continued his tradition of keeping machines simple in the CRAY-1. However, DLX and its close relatives are built primarily on the work of three research projects: the Berkeley RISC processor, the IBM 801, and the Stanford MIPS processor. These architectures have attracted enormous industrial interest because of claims of a performance advantage of anywhere from two to five times over other machines using the same technology.

Begun in 1975, the IBM project was the first to start but was the last to become public. The IBM machine was designed as an ECL minicomputer, while the university projects were both MOS-based microprocessors. John Cocke is considered to be the father of the 801 design. He received both the Eckert-Mauchly and Turing awards in recognition of his contribution. Radin [1982] describes the highlights of the 801 architecture. The 801 was an experimental project that was never designed to be a product. In fact, to keep down cost and complexity, the machine was built with only 24-bit registers.

In 1980, Patterson and his colleagues at Berkeley began the project that was to give this architectural approach its name (see Patterson and Ditzel [1980]). They built two machines called RISC-I and RISC-II. Because the IBM project was not widely known or discussed, the role played by the Berkeley group in promoting the RISC approach was critical to the acceptance of the technology. The Berkeley

group went on to build RISC machines targeted toward Smalltalk, described by Ungar et al. [1984], and LISP, described by Taylor et al. [1986].

In 1981, Hennessy and his colleagues at Stanford published a description of the Stanford MIPS machine. Efficient pipelining and compiler-assisted scheduling of the pipeline were both key aspects of the original MIPS design.

These early RISC machines—the 801, RISC-II, and MIPS—had much in common. Both university projects were interested in designing a simple machine that could be built in VLSI within the university environment. All three machines used a simple load-store architecture, fixed-format 32-bit instructions, and emphasized efficient pipelining. Patterson [1985] describes the three machines and the basic design principles that have come to characterize what a RISC machine is. Hennessy [1984] provides another view of the same ideas, as well as other issues in VLSI processor design.

In 1985, Hennessy published an explanation of the RISC performance advantage and traced its roots to a substantially lower CPI—under 2 for a RISC machine and over 10 for a VAX-11/780 (though not with identical workloads). A paper by Emer and Clark [1984] characterizing VAX-11/780 performance was instrumental in helping the RISC researchers understand the source of the performance advantage seen by their machines.

Since the university projects finished up, in the 1983–84 time frame, the technology has been widely embraced by industry. Many manufacturers of the early computers (those made before 1986) claimed that their products were RISC machines. However, these claims were often born more of marketing ambition than of engineering reality.

In 1986, the computer industry began to announce processors based on the technology explored by the three RISC research projects. Moussouris et al. [1986] describe the MIPS R2000 integer processor, while Kane's book [1986] is a complete description of the architecture. Hewlett-Packard converted their existing minicomputer line to RISC architectures; the HP Precision Architecture is described by Lee [1989]. IBM never directly turned the 801 into a product. Instead, the ideas were adopted for a new, low-end architecture that was incorporated in the IBM RT-PC and described in a collection of papers [Waters 1986]. In 1990, IBM announced a new RISC architecture (the RS 6000), which is the first superscalar RISC machine (see Chapter 4). In 1987, Sun Microsystems began delivering machines based on the SPARC architecture, a derivative of the Berkeley RISC-II machine; SPARC is described in Garner et al. [1988]. The PowerPC joined the forces of Apple, IBM, and Motorola. Appendix C summarizes several RISC architectures.

Prior to the RISC architecture movement, the major trend had been highly microcoded architectures aimed at reducing the semantic gap. DEC, with the VAX, and Intel, with the iAPX 432, were among the leaders in this approach. Today it is hard to find a computer company without a RISC product. With the 1994 announcement that Hewlett Packard and Intel will eventually have a common architecture, the end of the 1970s architectures draws near.

References

AMDAHL, G. M., G. A. BLAAUW, AND F. P. BROOKS, JR. [1964]. "Architecture of the IBM System 360," *IBM J. Research and Development* 8:2 (April), 87–101.

BARTON, R. S. [1961]. "A new approach to the functional design of a computer," *Proc. Western Joint Computer Conf.*, 393–396.

BELL, G., R. CADY, H. MCFARLAND, B. DELAGI, J. O'LAUGHLIN, R. NOONAN, AND W. WULF [1970]. "A new architecture for mini-computers: The DEC PDP-11," *Proc. AFIPS SJCC*, 657–675.

BHANDARKAR, D., AND D. W. CLARK [1991]. "Performance from architecture: Comparing a RISC and a CISC with similar hardware organizations," *Proc. Fourth Conf. on Architectural Support for Programming Languages and Operating System*s, IEEE/ACM (April), Palo Alto, Calif., 310–19.

CHOW, F. C. [1983]. *A Portable Machine-Independent Global Optimizer—Design and Measurements*, Ph.D. Thesis, Stanford Univ. (December).

CLARK, D. AND H. LEVY [1982]. "Measurement and analysis of instruction set use in the VAX-11/780," *Proc. Ninth Symposium on Computer Architecture* (April), Austin, Tex., 9–17.

CLARK, D. AND W. D. STRECKER [1980]. "Comments on 'the case for the reduced instruction set computer'," *Computer Architecture News* 8:6 (October), 34–38.

CRAWFORD, J. AND P. GELSINGER [1988]. *Programming the 80386*, Sybex Books, Alameda, Calif.

DITZEL, D. R. AND D. A. PATTERSON [1980]. "Retrospective on high-level language computer architecture," in *Proc. Seventh Annual Symposium on Computer Architecture*, La Baule, France (June), 97–104.

EMER, J. S. AND D. W. CLARK [1984]. "A characterization of processor performance in the VAX-11/780," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 301–310.

GAGLIARDI, U. O. [1973]. "Report of workshop 4–Software-related advances in computer hardware," *Proc. Symposium on the High Cost of Software*, Menlo Park, Calif., 99–120.

GARNER, R., A. AGARWAL, F. BRIGGS, E. BROWN, D. HOUGH, B. JOY, S. KLEIMAN, S. MUNCHNIK, M. NAMJOO, D. PATTERSON, J. PENDLETON, AND R. TUCK [1988]. "Scalable processor architecture (SPARC)," *COMPCON, IEEE* (March), San Francisco, 278–283.

HAUCK, E. A., AND B. A. DENT [1968]. "Burroughs' B6500/B7500 stack mechanism," *Proc. AFIPS SJCC*, 245–251.

HENNESSY, J. [1984]. "VLSI processor architecture," *IEEE Trans. on Computers* C-33:11 (December), 1221–1246.

HENNESSY, J. [1985]. "VLSI RISC processors," *VLSI Systems Design* VI:10 (October), 22–32.

HENNESSY, J., N. JOUPPI, F. BASKETT, AND J. GILL [1981]. "MIPS: A VLSI processor architecture," *Proc. CMU Conf. on VLSI Systems and Computations* (October), Computer Science Press, Rockville, Md.

KANE, G. [1986]. *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, N.J.

LEE, R. [1989]. "Precision architecture," *Computer* 22:1 (January), 78–91.

LEVY, H. AND R. ECKHOUSE [1989]. *Computer Programming and Architecture: The VAX*, Digital Press, Boston.

LUNDE, A. [1977]. "Empirical evaluation of some features of instruction set processor architecture," *Comm. ACM* 20:3 (March), 143–152.

MCKEEMAN, W. M. [1967]. "Language directed computer design," *Proc. 1967 Fall Joint Computer Conf.*, Washington, D.C., 413–417.

MEYERS, G. J. [1978]. "The evaluation of expressions in a storage-to-storage architecture," *Computer Architecture News* 7:3 (October), 20–23.

MEYERS, G. J. [1982]. *Advances in Computer Architecture*, 2nd ed., Wiley, New York.

MOUSSOURIS, J., L. CRUDELE, D. FREITAS, C. HANSEN, E. HUDSON, S. PRZYBYLSKI, T. RIORDAN, AND C. ROWEN [1986]. "A CMOS RISC processor with integrated system functions," *Proc. COMPCON, IEEE* (March), San Francisco, 191.

PATTERSON, D. [1985]. "Reduced instruction set computers," *Comm. ACM* 28:1 (January), 8–21.

PATTERSON, D. A. AND D. R. DITZEL [1980]. "The case for the reduced instruction set computer," *Computer Architecture News* 8:6 (October), 25–33.

RADIN, G. [1982]. "The 801 minicomputer," *Proc. Symposium Architectural Support for Programming Languages and Operating Systems* (March), Palo Alto, Calif., 39–47.

STRECKER, W. D. [1978]. "VAX-11/780: A virtual address extension of the PDP-11 family," *Proc. AFIPS National Computer Conf.* 47, 967–980.

TANENBAUM, A. S. [1978]. "Implications of structured programming for machine architecture," *Comm. ACM* 21:3 (March), 237–246.

TAYLOR, G., P. HILFINGER, J. LARUS, D. PATTERSON, AND B. ZORN [1986]. "Evaluation of the SPUR LISP architecture," *Proc. 13th Symposium on Computer Architecture (*June), Tokyo.

THORNTON, J. E. [1964]. "Parallel operation in Control Data 6600," *Proc. AFIPS Fall Joint Computer Conf.* 26, part 2, 33–40.

UNGAR, D., R. BLAU, P. FOLEY, D. SAMPLES, AND D. PATTERSON [1984]. "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 188–197.

WAKERLY, J. [1989]. *Microcomputer Architecture and Programming*, J. Wiley, New York.

WATERS, F., ED. [1986]. *IBM RT Personal Computer Technology*, IBM, Austin, Tex., SA 23-1057.

WIECEK, C. [1982]. "A case study of the VAX 11 instruction set usage for compiler execution," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* (March), IEEE/ACM, Palo Alto, Calif., 177–184.

WULF, W. [1981]. "Compilers and computer architecture," *Computer* 14:7 (July), 41–47.

EXERCISES

2.1 [20/15/10] <2.3,2.8> We are designing instruction set formats for a load-store architecture and are trying to decide whether it is worthwhile to have multiple offset lengths for branches and memory references. We have decided that both branch and memory references can have only 0-, 8-, and 16-bit offsets. The length of an instruction would be equal to 16 bits + offset length in bits. ALU instructions will be 16 bits. Figure 2.32 contains the data in cumulative form. Assume an additional bit is needed for the sign on the offset.

For instruction set frequencies, use the data for DLX from the average of the five benchmarks for the load-store machine in Figure 2.26. Assume that the miscellaneous instructions are all ALU instructions that use only registers.

a. [20] <2.3,2.8> Suppose offsets were permitted to be 0, 8, or 16 bits in length, including the sign bit. What is the average length of an executed instruction?

b. [15] <2.3,2.8> Suppose we wanted a fixed-length instruction and we chose a 24-bit instruction length (for everything, including ALU instructions). For every offset of longer than 8 bits, an additional instruction is required. Determine the number of

Offset bits	Cumulative data references	Cumulative branches
0	17%	0%
1	17%	0%
2	23%	24%
3	32%	49%
4	40%	64%
5	48%	79%
6	54%	87%
7	57%	93%
8	60%	98%
9	61%	99%
10	69%	100%
11	71%	100%
12	75%	100%
13	78%	100%
14	80%	100%
15	100%	100%

FIGURE 2.32 The second and third columns contain the cumulative percentage of the data references and branches, respectively, that can be accommodated with the corresponding number of bits of magnitude in the displacement. These are the average distances of all 10 programs in Figure 2.7.

instruction bytes fetched in this machine with fixed instruction size versus those fetched with a byte-variable-sized instruction as defined in part (a).

c. [10] <2.3,2.8> Now suppose we use a fixed offset length of 16 bits so that no additional instruction is ever required. How many instruction bytes would be required? Compare this result to your answer to part (b), which used 8-bit fixed offsets that used additional instruction words when larger offsets were required.

2.2 [15/10] <2.2> Several researchers have suggested that adding a register-memory addressing mode to a load-store machine might be useful. The idea is to replace sequences of

```
LOAD    R1,0(Rb)
ADD     R2,R2,R1
```

by

```
ADD     R2,0(Rb)
```

Assume the new instruction will cause the clock cycle to increase by 10%. Use the instruction frequencies for the gcc benchmark on the load-store machine from Figure 2.26. The new instruction affects only the clock cycle and not the CPI.

a. [15] <2.2> What percentage of the loads must be eliminated for the machine with the new instruction to have at least the same performance?

b. [10] <2.2> Show a situation in a multiple instruction sequence where a load of R1 followed immediately by a use of R1 (with some type of opcode) could not be replaced by a single instruction of the form proposed, assuming that the same opcode exists.

2.3 [20] <2.2> Your task is to compare the memory efficiency of four different styles of instruction set architectures. The architecture styles are

1. *Accumulator*—All operations occur between a single register and a memory location.
2. *Memory-memory*—All three operands of each instruction are in memory.
3. *Stack*—All operations occur on top of the stack. Only push and pop access memory; all other instructions remove their operands from stack and replace them with the result. The implementation uses a stack for the top two entries; accesses that use other stack positions are memory references.
4. *Load-store*—All operations occur in registers, and register-to-register instructions have three operands per instruction. There are 16 general-purpose registers, and register specifiers are 4 bits long.

To measure memory efficiency, make the following assumptions about all four instruction sets:

- The opcode is always 1 byte (8 bits).
- All memory addresses are 2 bytes (16 bits).
- All data operands are 4 bytes (32 bits).
- All instructions are an integral number of bytes in length.

There are no other optimizations to reduce memory traffic, and the variables A, B, C, and D are initially in memory.

Invent your own assembly language mnemonics and write the best equivalent assembly language code for the high-level-language fragment given. Write the four code sequences for

```
A = B + C;
B = A + C;
D = A - B;
```

Calculate the instruction bytes fetched and the memory-data bytes transferred. Which architecture is most efficient as measured by code size? Which architecture is most efficient as measured by total memory bandwidth required (code + data)?

2.4 [Discussion] <2.2–2.9> What are the *economic* arguments (i.e., more machines sold) for and against changing instruction set architecture?

2.5 [25] <2.1–2.5> Find an instruction set manual for some older machine (libraries and private bookshelves are good places to look). Summarize the instruction set with the discriminating characteristics used in Figure 2.2. Write the code sequence for this machine

for the statements in Exercise 2.3. The size of the data need not be 32 bits as in Exercise 2.3 if the word size is smaller in the older machine.

2.6 [20] <2.8> Consider the following fragment of C code:

```
for (i=0; i<=100; i++)
        {A[i] = B[i] + C;}
```

Assume that A and B are arrays of 32-bit integers, and C and i are 32-bit integers. Assume that all data values and their addresses are kept in memory (at addresses 0, 5000, 1500, and 2000 for A, B, C, and i, respectively) except when they are operated on. Assume that values in registers are lost between iterations of the loop.

Write the code for DLX; how many instructions are required dynamically? How many memory-data references will be executed? What is the code size in bytes?

2.7 [20] <App. D> Repeat Exercise 2.6, but this time write the code for the 80x86.

2.8 [20] <2.8> For this question use the code sequence of Exercise 2.6, but put the scalar data—the value of i, the value of C, and the addresses of the array variables (but not the actual array)—in registers and keep them there whenever possible.

Write the code for DLX; how many instructions are required dynamically? How many memory-data references will be executed? What is the code size in bytes?

2.9 [20] <App. D> Make the same assumptions and answer the same questions as the prior exercise, but this time write the code for the 80x86.

2.10 [15] <2.8> When designing memory systems it becomes useful to know the frequency of memory reads versus writes and also accesses for instructions versus data. Using the average instruction-mix information for DLX in Figure 2.26, find

- the percentage of all memory accesses for data
- the percentage of data accesses that are reads
- the percentage of all memory accesses that are reads

Ignore the size of a datum when counting accesses.

2.11 [18] <2.8> Compute the effective CPI for DLX using Figure 2.26. Suppose we have made the following measurements of average CPI for instructions:

Instruction	Clock cycles
All ALU instructions	1.0
Loads-stores	1.4
Conditional branches	
Taken	2.0
Not taken	1.5
Jumps	1.2

Assume that 60% of the conditional branches are taken and that all instructions in the miscellaneous category of Figure 2.26 are ALU instructions. Average the instruction frequencies of gcc and espresso to obtain the instruction mix.

2.12 [20/10] <2.3,2.8> Consider adding a new index addressing mode to DLX. The addressing mode adds two registers and an 11-bit signed offset to get the effective address.

Our compiler will be changed so that code sequences of the form

```
ADD R1, R1, R2
LW  Rd, 100(R1)(or store)
```

will be replaced with a load (or store) using the new addressing mode. Use the overall average instruction frequencies from Figure 2.26 in evaluating this addition.

a. [20] <2.3,2.8> Assume that the addressing mode can be used for 10% of the displacement loads and stores (accounting for both the frequency of this type of address calculation and the shorter offset). What is the ratio of instruction count on the enhanced DLX compared to the original DLX?

b. [10] <2.3,2.8> If the new addressing mode lengthens the clock cycle by 5%, which machine will be faster and by how much?

2.13 [25/15] <2.7> Find a C compiler and compile the code shown in Exercise 2.6 for one of the machines covered in this book. Compile the code both optimized and unoptimized.

a. [25] <2.7> Find the instruction count, dynamic instruction bytes fetched, and data accesses done for both the optimized and unoptimized versions.

b. [15] <2.7> Try to improve the code by hand and compute the same measures as in part (a) for your hand-optimized version.

2.14 [30] <2.8> Small synthetic benchmarks can be very misleading when used for measuring instruction mixes. This is particularly true when these benchmarks are optimized. In this exercise and Exercises 2.15–2.17, we want to explore these differences. These programming exercises can be done with any load-store machine.

Compile Whetstone with optimization. Compute the instruction mix for the top 20 most frequently executed instructions. How do the optimized and unoptimized mixes compare? How does the optimized mix compare to the mix for spice on the same or a similar machine?

2.15 [30] <2.8> Follow the same guidelines as the prior exercise, but this time use Dhrystone and compare it with TeX.

2.16 [30] <2.8> Many computer manufacturers now include tools or simulators that allow you to measure the instruction set usage of a user program. Among the methods in use are machine simulation, hardware-supported trapping, and a compiler technique that instruments the object-code module by inserting counters. Find a processor available to you that includes such a tool. Use it to measure the instruction set mix for one of TeX, gcc, or spice. Compare the results to those shown in this chapter.

2.17 [30] <2.3,2.8> DLX has only three operand formats for its register-register operations. Many operations might use the same destination register as one of the sources. We

could introduce a new instruction format into DLX called R_2 that has only two operands and is a total of 24 bits in length. By using this instruction type whenever an operation had only two different register operands, we could reduce the instruction bandwidth required for a program. Modify the DLX simulator to count the frequency of register-register operations with only two different register operands. Using the benchmarks that come with the simulator, determine how much more instruction bandwidth DLX requires than DLX with the R_2 format.

2.18 [25] <App. C> How much do the instruction set variations among the RISC machines discussed in Appendix C affect performance? Choose at least three small programs (e.g., a sort), and code these programs in DLX and two other assembly languages. What is the resulting difference in instruction count?