A

Instruction Set Principles

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)

A.1 Introduction

In this appendix we concentrate on instruction set architecture—the portion of the computer visible to the programmer or compiler writer. Most of this material should be review for readers of this book; we include it here for background. This appendix introduces the wide variety of design alternatives available to the instruction set architect. In particular, we focus 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 RISC-V instruction set, which is typical of RISC architectures. We conclude with fallacies and pitfalls of instruction set design.

To illustrate the principles further and to provide a comparison with RISC-V, Appendix K also gives four examples of other general-purpose RISC architectures (MIPS, Power ISA, SPARC, and Armv8), four embedded RISC processors (ARM Thumb2, RISC-V Compressed, microMIPS), and three older architectures (80x86, IBM 360/370, and VAX). Before we discuss how to classify architectures, we need to say something about instruction set measurement.

Throughout this appendix, we examine a wide variety of architectural measurements. Clearly, 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. We believe that the measurements in this appendix 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 computer would want to analyze a much larger collection of programs before making architectural decisions. The measurements shown are usually *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.

Before starting with the general principles, let's review the three application areas from Chapter 1. *Desktop computing* emphasizes the performance of programs with integer and floating-point data types, with little regard for program size. For example, code size has never been reported in the five generations of SPEC benchmarks. *Servers* today are used primarily for database, file server, and Web applications, plus some time-sharing applications for many users. Hence, floating-point performance is much less important for performance than integers and character strings, yet virtually every server processor still includes floating-point instructions. *Personal mobile devices* and *embedded applications* value cost and energy, so code size is important because less memory is both cheaper and lower energy, and some classes of instructions (such as floating point) may be optional to reduce chip costs, and a compressed version of the instructions set designed to save memory space may be used.

Thus, instruction sets for all three applications are very similar. In fact, architectures similar to RISC-V, which we focus on here, have been used successfully in desktops, servers, and embedded applications.

One successful architecture very different from RISC is the 80x86 (see Appendix K). Surprisingly, its success does not necessarily belie the advantages of a RISC instruction set. The commercial importance of binary compatibility with PC software combined with the abundance of transistors provided by Moore's Law led Intel to use a RISC instruction set internally while supporting an 80x86 instruction set externally. Recent 80x86 microprocessors, including all the Intel Core microprocessors built in the past decade, use hardware to translate from 80x86 instructions to RISC-like instructions and then execute the translated operations inside the chip. They maintain the illusion of 80x86 architecture to the programmer while allowing the computer designer to implement a RISC-style processor for performance. There remain, however, serious disadvantages for a complex instruction set like the 80x86, and we discuss these further in the conclusions.

Now that the background is set, we begin by exploring how instruction set architectures can be classified.

A.2 Classifying Instruction Set Architectures

The type of internal storage in a processor 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, and in an *accumulator architecture* one operand is implicitly the accumulator. The *general-purpose register architectures* have only explicit operands—either registers or memory locations. Figure A.1 shows a block diagram of such architectures, and Figure A.2 shows how the code sequence C = A + B would typically appear in these three classes of instruction sets. The explicit operands may be accessed directly from memory or may need to be first loaded into temporary storage, depending on the class of architecture and choice of specific instruction.

As the figures show, there are really two classes of register computers. One class can access memory as part of any instruction, called *register-memory* architecture, and the other can access memory only with load and store instructions, called *load-store* architecture. A third class, not found in computers shipping today, keeps all operands in memory and is called a *memory-memory* architecture. Some instruction set architectures have more registers than a single accumulator but place restrictions on uses of these special registers. Such an architecture is sometimes called an *extended accumulator* or *special-purpose register* computer.

Although most early computers used stack or accumulator-style architectures, virtually every new architecture designed after 1980 uses a load-store register architecture. The major reasons for the emergence of general-purpose register (GPR) computers are twofold. First, registers—like other forms of storage internal to the processor—are faster than memory. Second, registers are more efficient

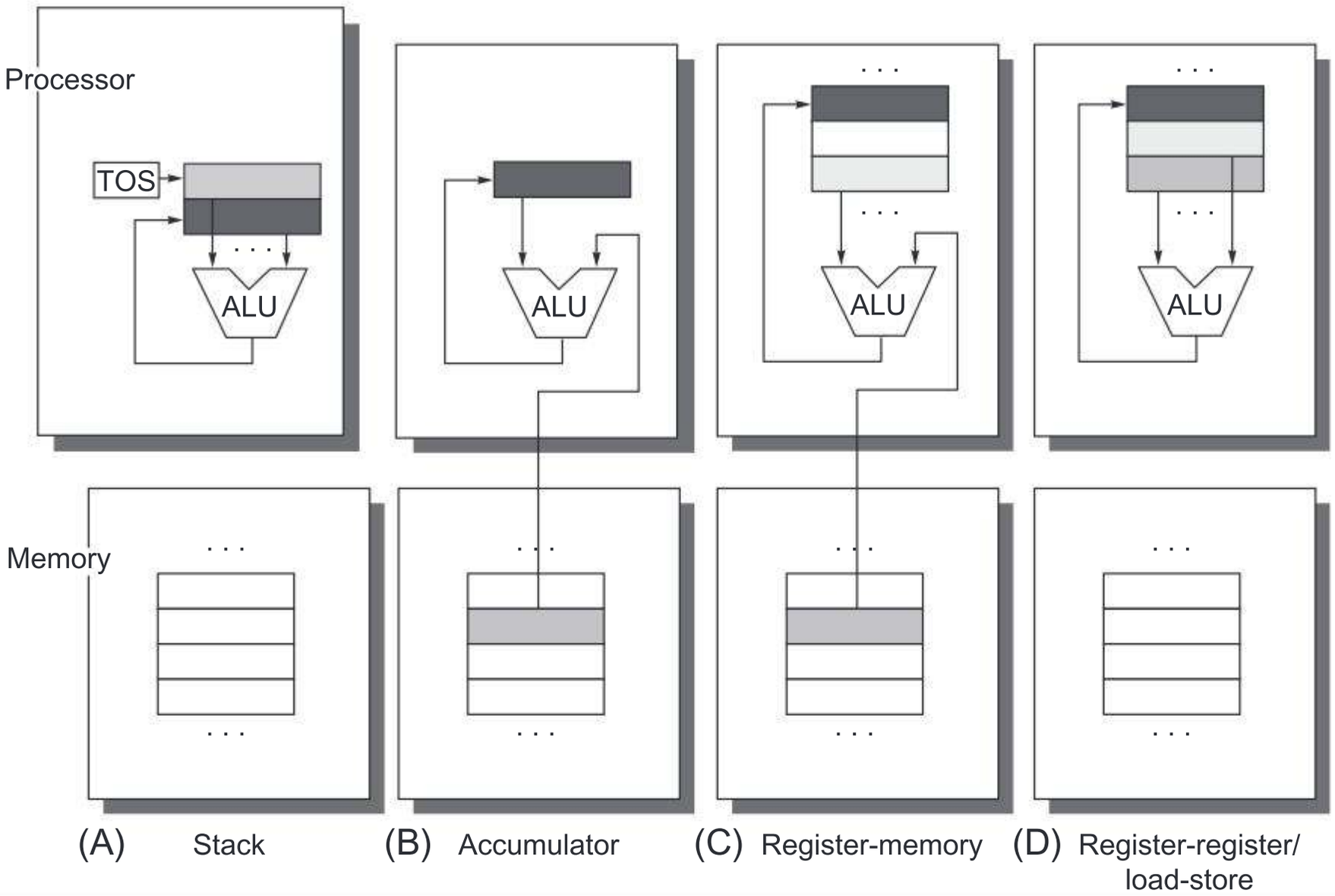


Figure A.1 Operand locations for four instruction set architecture classes. The arrows indicate whether the operand is an input or the result of the arithmetic-logical unit (ALU) operation, or both an input and result. Lighter shades indicate inputs, and the dark shade indicates the result. In (A), a top of stack (TOS) register points to the top input operand, which is combined with the operand below. The first operand is removed from the stack, the result takes the place of the second operand, and TOS is updated to point to the result. All operands are implicit. In (B), the accumulator is both an implicit input operand and a result. In (C), one input operand is a register, one is in memory, and the result goes to a register. All operands are registers in (D) and, like the stack architecture, can be transferred to memory only via separate instructions: push or pop for (A) and load or store for (D).

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

Figure A.2 The code sequence for `C = A + B` **for four classes of instruction sets.** Note that the `Add` instruction has implicit operands for stack and accumulator architectures and explicit operands for register architectures. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed. Figure A.1 shows the `Add` operation for each class of architecture.

for a compiler to use than other forms of internal storage. For example, on a register computer the expression (A * B)+(B * C)−(A * D) 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). Nevertheless, on a stack computer the hardware must evaluate the expression in only one order, because operands are hidden on the stack, and it may have to load an operand multiple times.

More importantly, registers can be used to hold variables. When variables are allocated to registers, the memory traffic reduces, the program speeds up (because registers are faster than memory), and the code density improves (because a register can be named with fewer bits than can a memory location).

As explained in Section A.8, compiler writers would prefer that all registers be equivalent and unreserved. Older computers 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 the effectiveness of 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. Modern compiler technology and its ability to effectively use larger numbers of registers has led to an increase in register counts in more recent architectures.

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 one result operand 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. Figure A.3 shows

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	ARM, MIPS, PowerPC, SPARC, RISC-V
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Figure A.3 Typical combinations of memory operands and total operands per typical ALU instruction with examples of computers. Computers with no memory reference per ALU instruction are called load-store or register-register computers. 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.

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 Appendix C)	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs, which may have some instruction cache effects
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density	Operands are not equivalent because 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 vary by operand location
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

Figure A.4 Advantages and disadvantages of the three most common types of general-purpose register computers. The notation (*m*, *n*) means *m* memory operands and *n* total operands. In general, computers with fewer alternatives simplify the compiler's task because there are fewer decisions for the compiler to make (see Section A.8). Computers with a wide variety of flexible instruction formats reduce the number of bits required to encode the program. The number of registers also affects the instruction size because you need $\log_2$ (number of registers) for each register specifier in an instruction. Thus, doubling the number of registers takes three extra bits for a register-register architecture, or about 10% of a 32-bit instruction.

combinations of these two attributes with examples of computers. Although there are seven possible combinations, three serve to classify nearly all existing computers. As we mentioned earlier, these three are load-store (also called register-register), register-memory, and memory-memory.

Figure A.4 shows the advantages and disadvantages of each of these alternatives. 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 computer with memory-memory operations could easily be ignored by the compiler and used as a load-store computer. One of the most pervasive architectural impacts is on instruction encoding and the number of instructions needed to perform a task. We see the impact of these architectural alternatives on implementation approaches in Appendix C and Chapter 3.

Summary: Classifying Instruction Set Architectures

Here and at the end of Sections A.3—A.8 we summarize those characteristics we would expect to find in a new instruction set architecture, building the foundation for the RISC-V architecture introduced in Section A.9. From this section we should clearly expect the use of general-purpose registers. Figure A.4, combined with Appendix C on pipelining, leads to the expectation of a load-store version of a general-purpose register architecture.

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

A.3 Memory Addressing

Independent of whether the architecture is load-store or allows any operand to be a memory reference, it must define how memory addresses are interpreted and how they are specified. The measurements presented here are largely, but not completely, computer independent. In some cases the measurements are significantly affected by the compiler technology. These measurements have been made using an optimizing compiler, because compiler technology plays a critical 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 computers also provide access for double words (64 bits).

There are two different conventions for ordering the bytes within a larger object. *Little Endian* byte order puts the byte whose address is "x … x000" at the least-significant position in the double word (the little end). The bytes are numbered:

7	6	5	4	3	2	1	0

Big Endian byte order puts the byte whose address is "x … x000" at the most-significant position in the double word (the big end). The bytes are numbered:

0	1	2	3	4	5	6	7

When operating within one computer, the byte order is often unnoticeable—only programs that access the same locations as both, say, words and bytes, can notice the difference. Byte order is a problem when exchanging data among computers with different orderings, however. Little Endian ordering also fails to match the normal ordering of words when strings are compared. Strings appear "SDRAWKCAB" (backwards) in the registers.

A second memory issue is that in many computers, 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 A.5 shows the addresses at which an access is aligned or misaligned.

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

	Value of three low-order bits of byte address							
Width of object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)		Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)		Misaligned				Misaligned		
4 bytes (word)			Misaligned				Misaligned	
4 bytes (word)				Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)		Misaligned						
8 bytes (double word)			Misaligned					
8 bytes (double word)				Misaligned				
8 bytes (double word)					Misaligned			
8 bytes (double word)						Misaligned		
8 bytes (double word)							Misaligned	
8 bytes (double word)								Misaligned

Figure A.5 Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide as the object. The figure shows the memory organized as 8 bytes wide. The byte offsets that label the columns specify the low-order three bits of the address.

Even if data are aligned, supporting byte, half-word, and word accesses requires an alignment network to align bytes, half words, and words in 64-bit registers. For example, in Figure A.5, suppose we read a byte from an address with its 3 low-order bits having the value 4. We will need to shift right 3 bytes to align the byte to the proper place in a 64-bit register. Depending on the instruction, the computer may also need to sign-extend the quantity. Stores are easy: only the addressed bytes in memory may be altered. On some computers a byte, half-word, and word operation does not affect the upper portion of a register. Although all the computers discussed in this book permit byte, half-word, and word accesses to memory, only the IBM 360/370, Intel 80x86, and VAX support ALU operations on register operands narrower than the full width.

Now that we have discussed alternative interpretations of memory addresses, we can discuss the ways addresses are specified by instructions, called *addressing modes*.

Addressing Modes

Given an address, we now know what bytes to access in memory. In this subsection we will look at addressing modes—how architectures specify the address

of an object they will access. Addressing modes specify constants and registers in addition to locations in memory. When a memory location is used, the actual memory address specified by the addressing mode is called the *effective address*.

Figure A.6 shows all the data addressing modes that have been used in recent computers. 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 because they don't usually have memory addresses. 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, discussed in Section A.6.

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 (+ simulates register indirect, direct addressing modes)
Register 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	`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 computers

Figure A.6 Selection of addressing modes with examples, meaning, and usage. 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). These addressing modes are only useful when the elements being accessed are adjacent in memory. RISC computers use displacement addressing to simulate register indirect with 0 for the address and to simulate direct addressing using 0 in the base register. In our measurements, we use the first name shown for each mode. The extensions to C used as hardware descriptions are defined on page A.38.

Figure A.6 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 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 computer and may increase the average clock cycles per instruction (CPI) of computers that implement those modes. Thus, the usage of various addressing modes is quite important in helping the architect choose what to include.

Figure A.7 shows the results of measuring addressing mode usage patterns in three programs on the VAX architecture. We use the old VAX architecture for a few measurements in this appendix because it has the richest set of addressing modes and the fewest restrictions on memory addressing. For example, Figure A.6 on page A.9 shows all the modes the VAX supports. Most

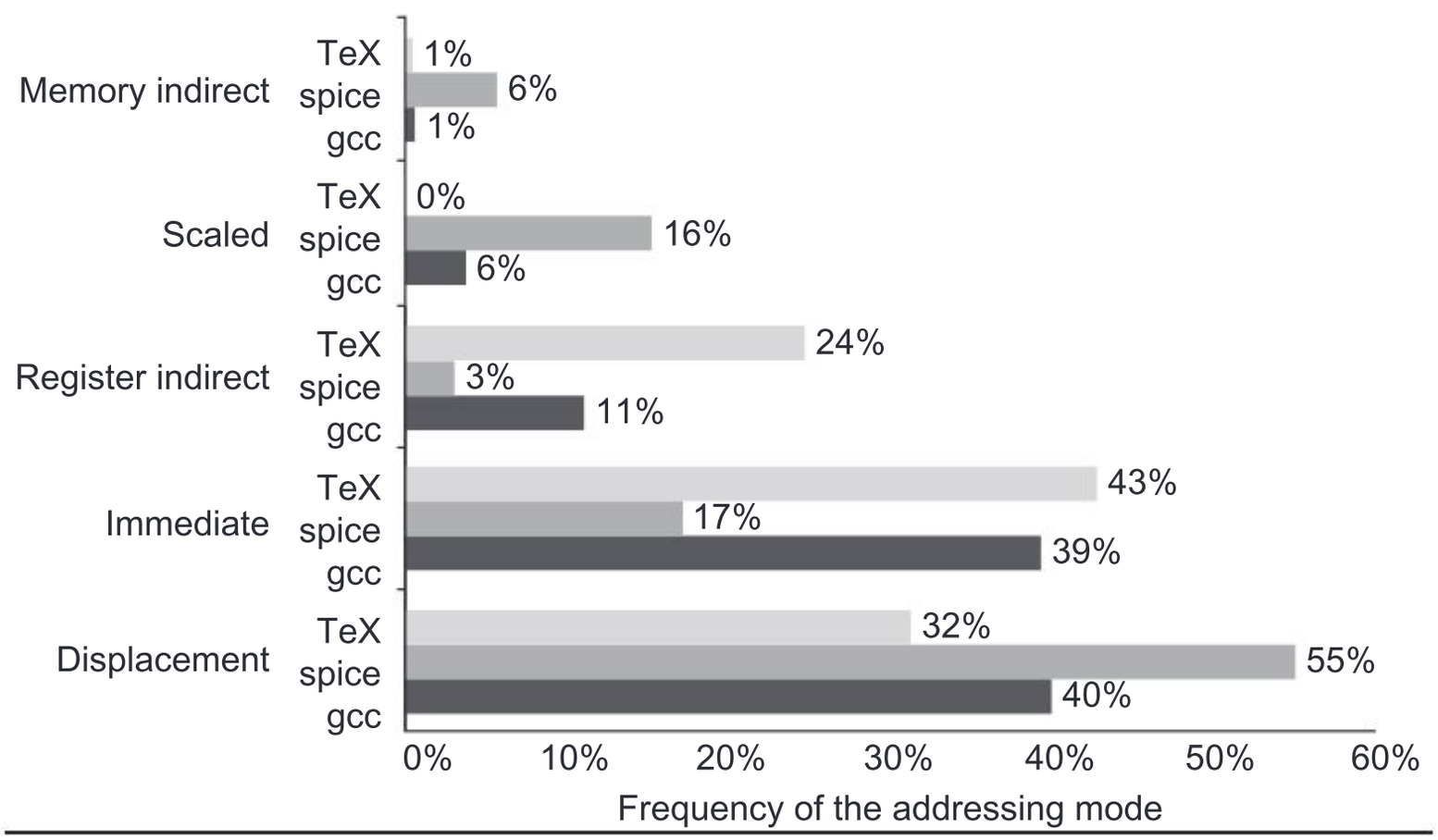


Figure A.7 Summary of use of memory addressing modes (including immediates). These major addressing modes account for all but a few percent (0%—3%) of the memory accesses. Register modes, which are not counted, account for one-half of the operand references, while memory addressing modes (including immediate) account for the other half. Of course, the compiler affects what addressing modes are used; see Section A.8. 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. Displacement mode includes all displacement lengths (8, 16, and 32 bits). The PC-relative addressing modes, used almost exclusively for branches, are not included. Only the addressing modes with an average frequency of over 1% are shown.

measurements in this appendix, however, will use the more recent register-register architectures to show how programs use instruction sets of current computers.

As Figure A.7 shows, displacement and immediate addressing dominate addressing mode usage. Let's look at some properties of these two heavily used modes.

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. Figure A.8 shows the measurements taken on the data access on a load-store architecture using our benchmark programs. We look at branch offsets in Section A.6—data accessing patterns and branches are different; little is gained by combining them, although in practice the immediate sizes are made the same for simplicity.

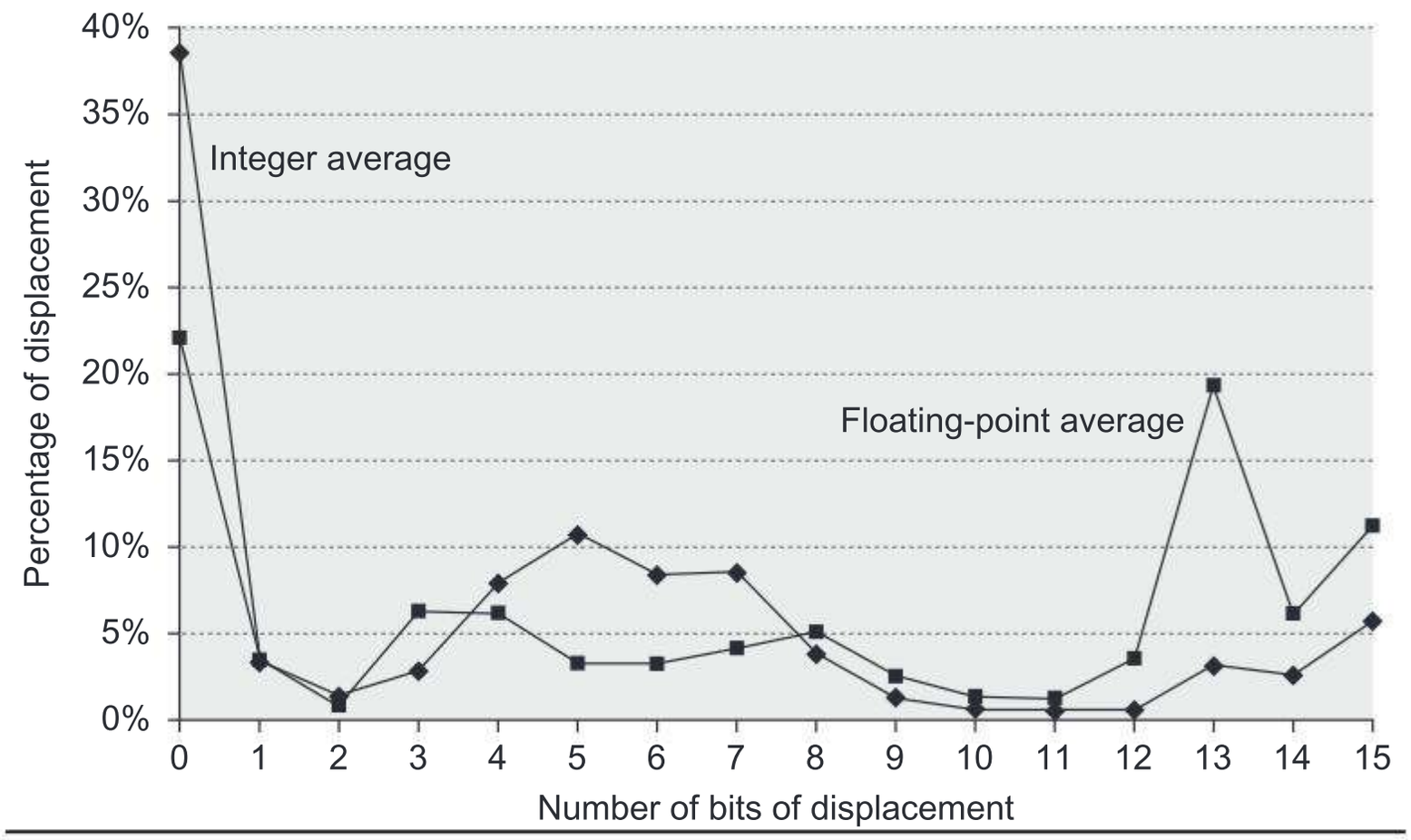


Figure A.8 Displacement values are widely distributed. There are both a large number of small values and a fair number of large values. The wide distribution of displacement values is due to multiple storage areas for variables and different displacements to access them (see Section A.8) as well as the overall addressing scheme the compiler uses. The *x*-axis is $\log_2$ of the displacement, that is, the size of a field needed to represent the magnitude of the displacement. Zero on the *x*-axis shows the percentage of displacements of value 0. The graph does not include the sign bit, which is heavily affected by the storage layout. Most displacements are positive, but a majority of the largest displacements (14+ bits) are negative. Because these data were collected on a computer with 16-bit displacements, they cannot tell us about longer displacements. These data were taken on the Alpha architecture with full optimization (see Section A.8) for SPEC CPU2000, showing the average of integer programs (CINT2000) and the average of floating-point programs (CFP2000).

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 tend to 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. Figure A.9 shows the frequency of immediates for the general classes of integer and floating-point operations in an instruction set.

Another important instruction set measurement is the range of values for immediates. Like displacement values, the size of immediate values affects instruction length. As Figure A.10 shows, small immediate values are most heavily used. Large immediates are sometimes used, however, most likely in addressing calculations.

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 indirect. Figure A.7 shows that they represent 75%–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 12–16 bits, because the caption in Figure A.8 suggests these sizes would capture 75%–99% of the displacements. Third, we would expect the size of the immediate field to be at least 8–16 bits. This claim is not substantiated by the caption of the figure to which it refers.

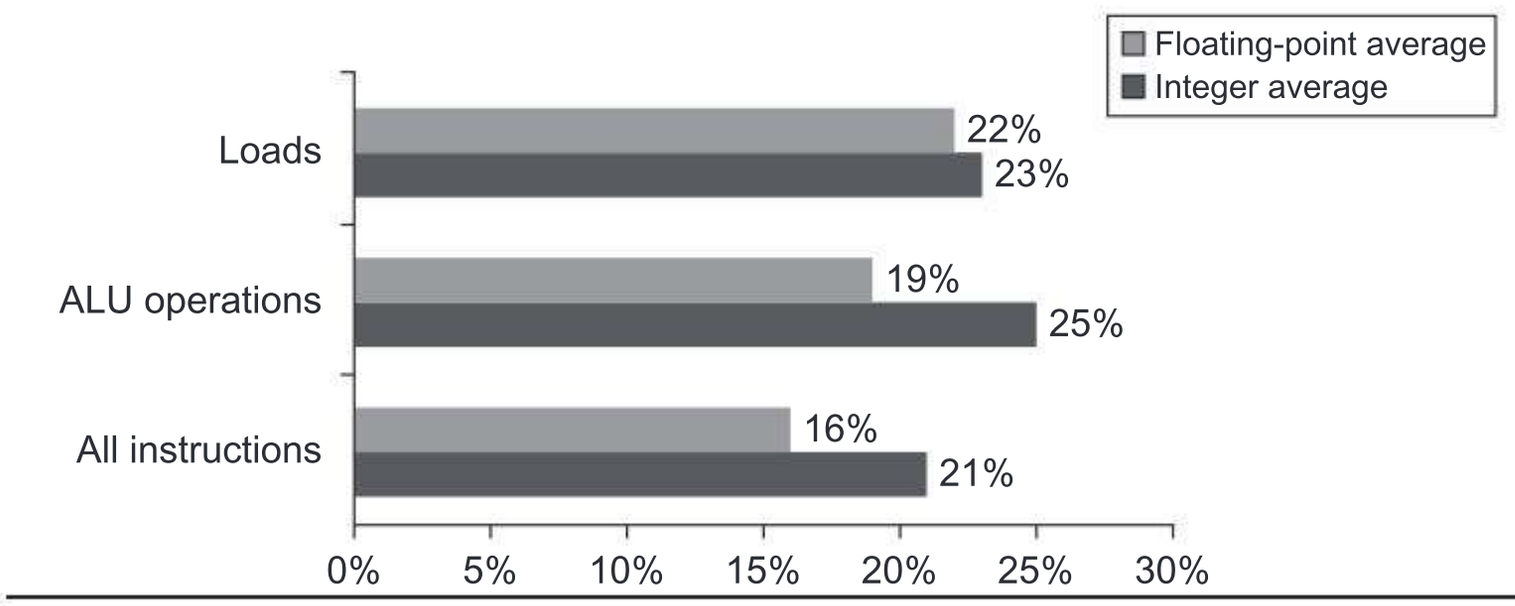


Figure A.9 About one-quarter of data transfers and ALU operations have an immediate operand. The bottom bars show that integer programs use immediates in about one-fifth of the instructions, while floating-point programs use immediates in about one-sixth of the instructions. For loads, the load immediate instruction loads 16 bits into either half of a 32-bit register. Load immediates are not loads in a strict sense because they do not access memory. Occasionally a pair of load immediates is used to load a 32-bit constant, but this is rare. (For ALU operations, shifts by a constant amount are included as operations with immediate operands.) The programs and computer used to collect these statistics are the same as in Figure A.8.

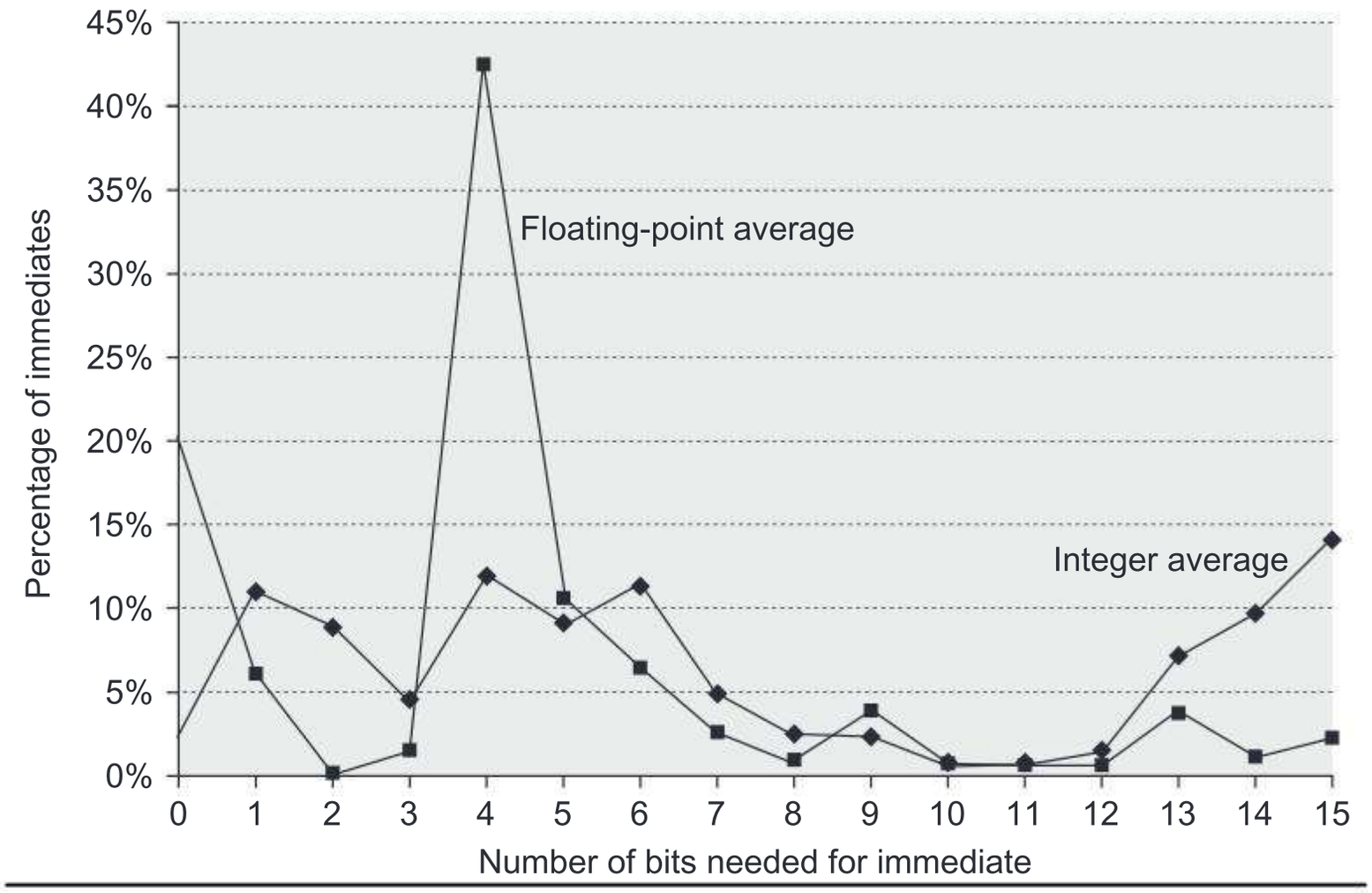


Figure A.10 The distribution of immediate values. The *x*-axis shows the number of bits needed to represent the magnitude of an immediate value—0 means the immediate field value was 0. The majority of the immediate values are positive. About 20% were negative for CINT2000, and about 30% were negative for CFP2000. These measurements were taken on an Alpha, where the maximum immediate is 16 bits, for the same programs as in Figure A.8. A similar measurement on the VAX, which supported 32-bit immediates, showed that about 20%—25% of immediates were longer than 16 bits. Thus, 16 bits would capture about 80% and 8 bits about 50%.

Having covered instruction set classes and decided on register-register architectures, plus the previous recommendations on data addressing modes, we next cover the sizes and meanings of data.

A.4 Type and Size of Operands

How is the type of an operand designated? Usually, encoding in the opcode designates the type of an operand—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. Computers with tagged data, however, can only be found in computer museums.

Let's start with desktop and server architectures. Usually the type of an operand—integer, single-precision floating point, character, and so on—effectively gives its size. Common operand types include character (8 bits), half word (16 bits), word (32 bits), single-precision floating point (also 1 word), and double-precision floating point (2 words). Integers are almost universally represented as two's complement binary numbers. Characters are usually in

ASCII, but the 16-bit Unicode (used in Java) is gaining popularity with the internationalization o f computers. Until the early 1980s, most computer manufacturers chose their own floating-point representation. Almost all computers since that time follow the same standard for floating point, the IEEE standard 754, although this level of accuracy has recently been abandoned in application-specific processors. The IEEE floating-point standard is discussed in detail in Appendix J.

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.

One reason to use decimal operands is to get results that exactly match decimal numbers, as some decimal fractions do not have an exact representation in binary. For example, 0.10_{10} is a simple fraction in decimal, but in binary it requires an infinite set of repeating digits: $0.0001100110\overline{0}11_{\ldots 2}$. Thus, calculations that are exact in decimal can be close but inexact in binary, which can be a problem for financial transactions. (See Appendix J to learn more about precise arithmetic.)

The SPEC benchmarks use byte or character, half-word (short integer), word (integer and single precision floating point), double-word (long integer), and floating-point data types. Figure A.11 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 computer have a 64-bit access path, or would taking two cycles to access a double word be satisfactory? As we saw earlier, byte accesses

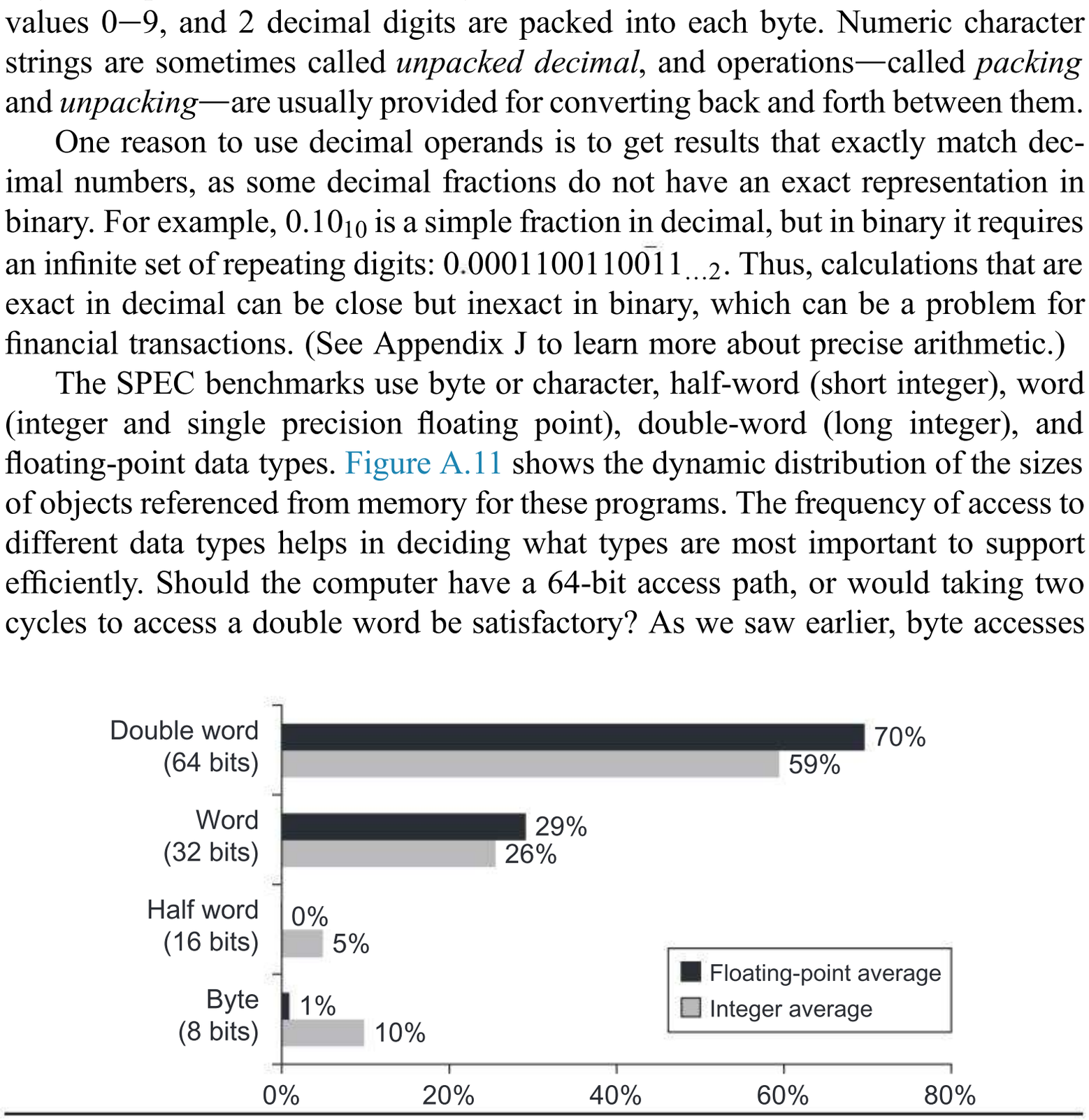


Figure A.11 Distribution of data accesses by size for the benchmark programs. The double-word data type is used for double-precision floating point in floating-point programs and for addresses, because the computer uses 64-bit addresses. On a 32-bit address computer the 64-bit addresses would be replaced by 32-bit addresses, and so almost all double-word accesses in integer programs would become single-word accesses.

require an alignment network: how important is it to support bytes as primitives? Figure A.11 uses memory references 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.

A.5 Operations in the Instruction Set

The operators supported by most instruction set architectures can be categorized as in Figure A.12. 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 A.13 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.

As mentioned before, the instructions in Figure A.13 are found in every computer for every application——desktop, server, embedded——with the variations of operations in Figure A.12 largely depending on which data types the instruction set includes.

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Loads-stores (move instructions on computers 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, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

Figure A.12 Categories of instruction operators and examples of each. All computers 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 computers 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 computer 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.

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 A.13 The top 10 instructions for the 80x86. Simple instructions dominate this list and are responsible for 96% of the instructions executed. These percentages are the average of the five SPECint92 programs.

A.6 Instructions for Control Flow

Because the measurements of branch and jump behavior are fairly independent of other measurements and applications, we now examine the use of control flow instructions, which have little in common with the operations of the previous sections.

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, computers introduced additional names. Throughout this book we will use *jump* when the change in control is unconditional and *branch* when the change is conditional.

We can distinguish four different types of control flow change:

- Conditional branches
- Jumps
- Procedure calls
- 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. Figure A.14 shows the frequencies of these control flow instructions for a load-store computer running our benchmarks.

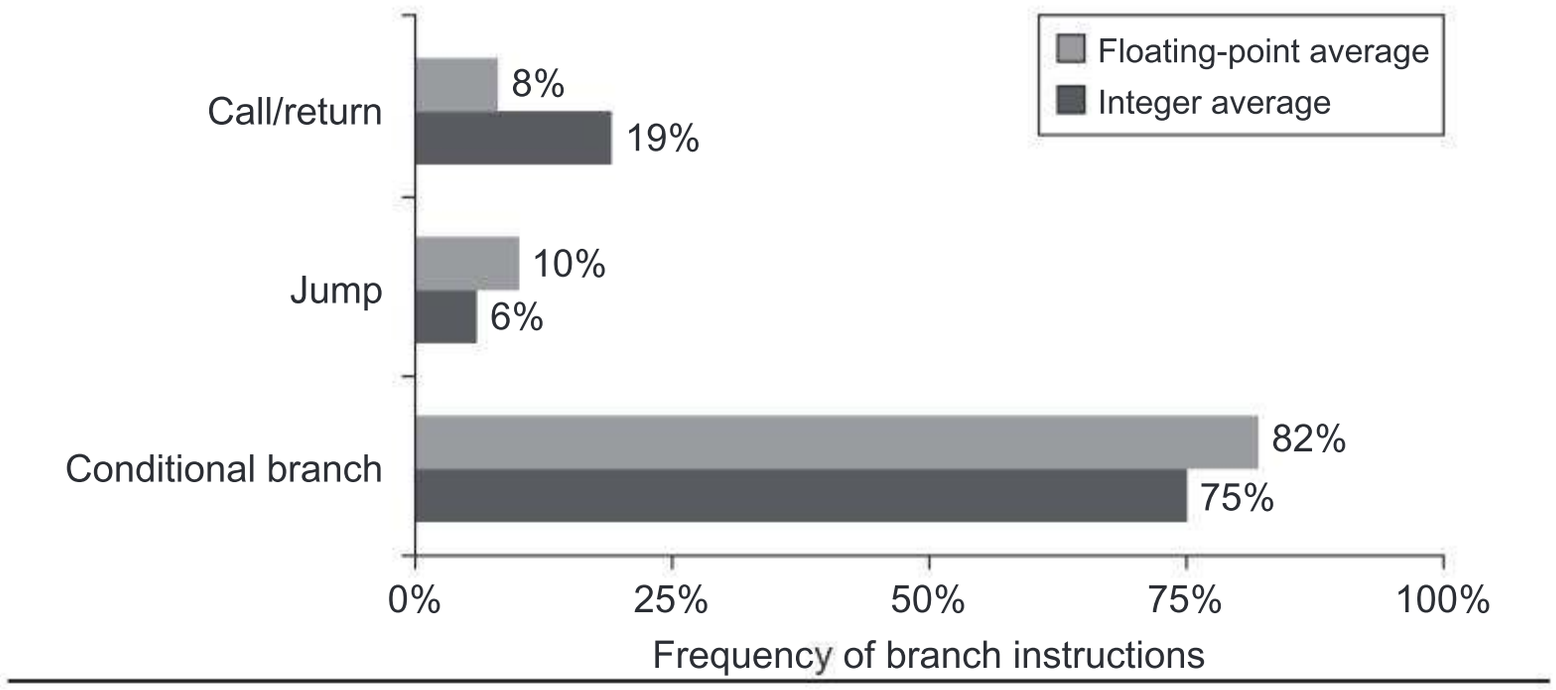


Figure A.14 Breakdown of control flow instructions into three classes: calls or returns, jumps, and conditional branches. Conditional branches clearly dominate. Each type is counted in one of three bars. The programs and computer used to collect these statistics are the same as those in Figure A.8.

Addressing Modes for Control Flow Instructions

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, because 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* (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 dynamically during execution.

To implement returns and indirect jumps when 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 four other important features:

- *Case* or *switch* statements, found in most programming languages (which select among one of several alternatives).
- *Virtual functions* or *methods* in object-oriented languages like C++ or Java (which allow different routines to be called depending on the type of the argument).

- *High-order functions* or *function pointers* in languages like C or C++ (which allow functions to be passed as arguments, giving some of the flavor of object-oriented programming).
- *Dynamically shared libraries* (which allow a library to be loaded and linked at runtime only when it is actually invoked by the program rather than loaded and linked statically before the program is run).

In all four 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, an important 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 A.15 shows the distribution of displacements for PC-relative branches in instructions. About 75% of the branches are in the forward direction.

Conditional Branch Options

Because most changes in control flow are branches, deciding how to specify the branch condition is important. Figure A.16 shows the three primary techniques in use today and their advantages and disadvantages.

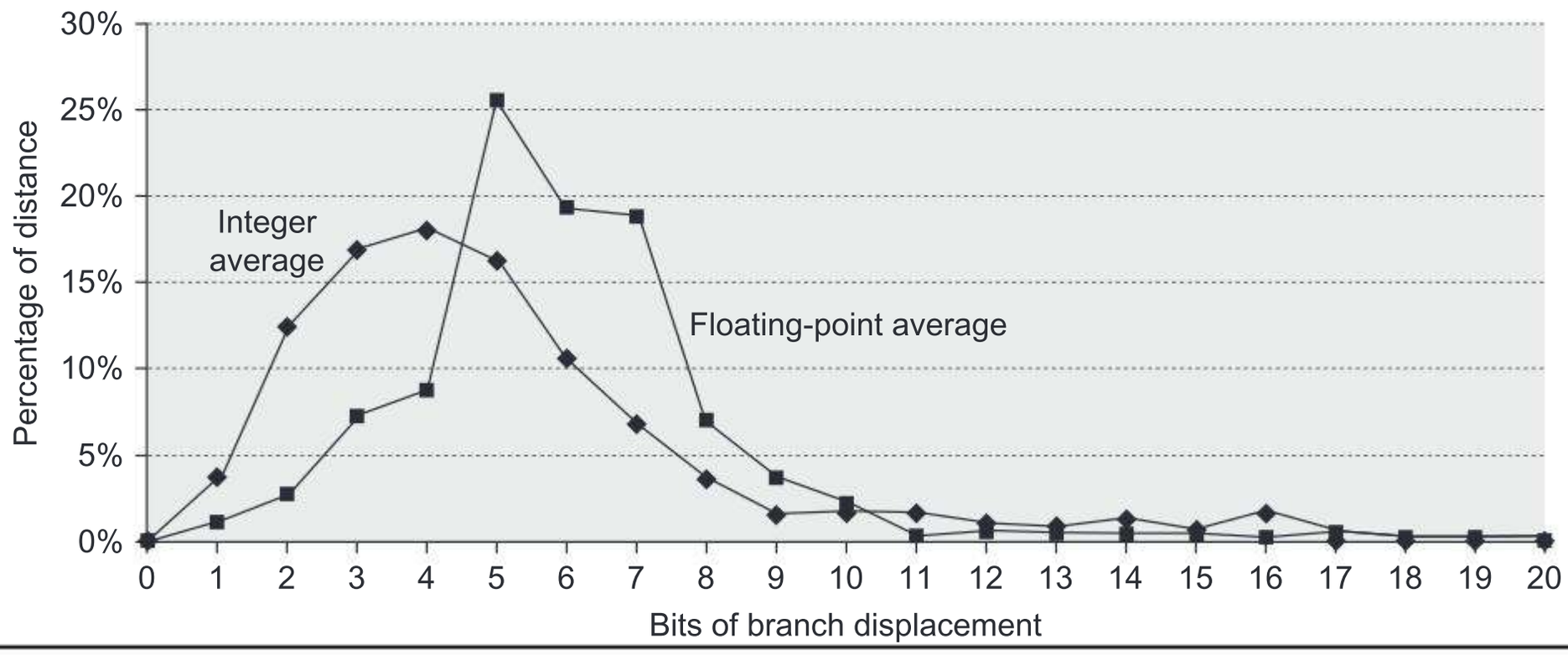


Figure A.15 Branch distances in terms of number of instructions between the target and the branch instruction. The most frequent branches in the integer programs are to targets that can be encoded in 4–8 bits. This result tells us that short displacement fields often suffice for branches and that the designer can gain some encoding density by having a shorter instruction with a smaller branch displacement. These measurements were taken on a load-store computer (Alpha architecture) with all instructions aligned on word boundaries. An architecture that requires fewer instructions for the same program, such as a VAX, would have shorter branch distances. However, the number of bits needed for the displacement may increase if the computer has variable-length instructions to be aligned on any byte boundary. The programs and computer used to collect these statistics are the same as those in Figure A.8.

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructionsbecause they pass information from one instruction to a branch
Condition register/ limited comparison	Alpha, MIPS	Tests arbitrary register with the result of a simple comparison (equality or zero tests)	Simple	Limited compare may affect critical path or require extra comparison for general condition
Compare and branch	PA-RISC, VAX, RISC-V	Compare is part of the branch. Fairly general compares are allowed (greater then, less then)	One instruction rather than two for a branch	May set critical path for branch instructions

Figure A.16 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. Computers with compare and branch often limit the set of compares and use a separate operation and register for more complex compares. Often, different techniques are used for branches based on floating-point comparison versus those based on integer comparison. This dichotomy is reasonable because the number of branches that depend on floating-point comparisons is much smaller than the number depending on integer comparisons.

One of the most noticeable properties of branches is that a large number of the comparisons are simple 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 A.17 shows the frequency of different comparisons used for conditional branching.

Procedure Invocation Options

Procedure calls and returns include control transfer and possibly some state saving; at a minimum the return address must be saved somewhere, sometimes in a special link register or just a GPR. Some older architectures provide a mechanism to save many registers, while newer architectures require the compiler to generate stores and loads for each register saved and restored.

There are two basic conventions in use to save registers: either at the call site or inside the procedure being called. *Caller saving* means that the calling procedure must save the registers that it wants preserved for access after the call, and thus the called procedure need not worry about registers. *Callee saving* is the opposite: the called procedure must save the registers it wants to use, leaving the caller unrestrained. 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

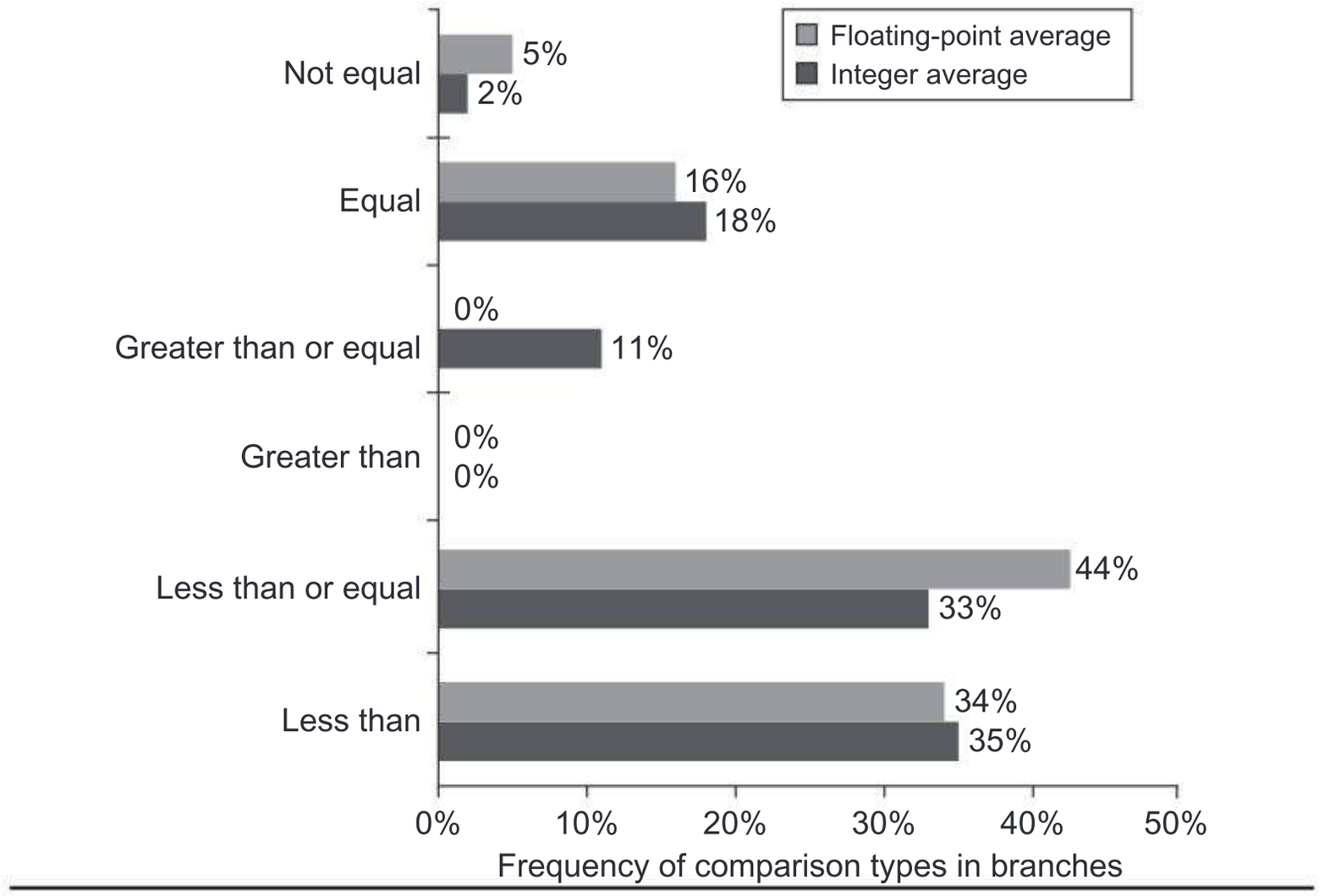


Figure A.17 Frequency of different types of compares in conditional branches. Less than (or equal) branches dominate this combination of compiler and architecture. These measurements include both the integer and floating-point compares in branches. The programs and computer used to collect these statistics are the same as those in Figure A.8.

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. Suppose P2 may not touch *x* but can call another procedure, P3, that may access *x*, yet P2 and P3 are compiled separately. 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, most real systems today use a combination of the two mechanisms. This convention is specified in an application binary interface (ABI) that sets down the basic rules as to which registers should be caller saved and which should be callee saved. Later in this appendix we will examine the mismatch between sophisticated instructions for automatically saving registers and the needs of the compiler.

Summary: Instructions for Control Flow

Control flow instructions are some of the most frequently executed instructions. Although there are many options for conditional branches, we would expect

branch addressing in a new architecture to be able to jump to hundreds of instructions either above or below the branch. This requirement suggests 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.

We have now completed our instruction architecture tour at the level seen by an assembly language programmer or compiler writer. We are leaning toward a load-store architecture with displacement, immediate, and register indirect addressing modes. These data are 8-, 16-, 32-, and 64-bit integers and 32- and 64-bit floating-point data. The instructions include simple operations, PC-relative conditional branches, jump and link instructions for procedure call, and register indirect jumps for procedure return (plus a few other uses).

Now we need to select how to represent this architecture in a form that makes it easy for the hardware to execute.

A.7 Encoding an Instruction Set

Clearly, the choices mentioned herein will affect how the instructions are encoded into a binary representation for execution by the processor. This representation affects not only the size of the compiled program but also the implementation of the processor, 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 older computers have one to five operands with 10 addressing modes for each operand (see Figure A.6). 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 are load-store computers 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.

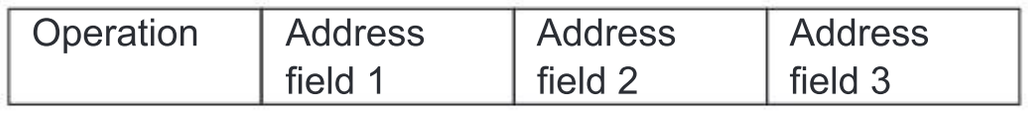
When encoding the instructions, the number of registers and the number of addressing modes both have a significant impact on the size of instructions, as the register field and addressing mode field may appear many times in a single instruction. In fact, for most instructions many more bits are consumed 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.

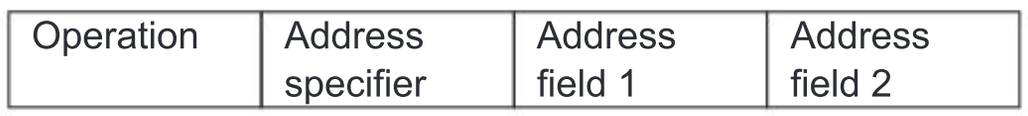
3. A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation. (The value of easily decoded instructions is discussed in Appendix C and Chapter 3.) As a minimum, the architect wants instructions to be in multiples of bytes, rather than an arbitrary bit length. Many desktop and server architects have chosen to use a fixed-length instruction to gain implementation benefits while sacrificing average code size.

Figure A.18 shows three popular choices for encoding the instruction set. The first we call *variable*, because 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*, because it combines the operation and the 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 processor. Variable tries to use as

(A) Variable (e.g., Intel 80x86, VAX)



(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)



(C) Hybrid (e.g., RISC V Compressed (RV32IC), IBM 360/370, microMIPS, Arm Thumb2)

Figure A.18 Three basic variations in instruction encoding: variable length, fixed length, and hybrid. The variable format can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand. It generally enables the smallest code representation, because unused fields need not be included. The fixed format always has the same number of operands, with the addressing modes (if options exist) specified as part of the opcode. It generally results in the largest code size. Although the fields tend not to vary in their location, they will be used for different purposes by different instructions. The hybrid approach has 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.

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.

Let's look at an 80x86 instruction to see an example of the variable encoding:

```
add EAX,1000(EBX)
```

The name `add` means a 32-bit integer add instruction with two operands, and this opcode takes 1 byte. An 80x86 address specifier is 1 or 2 bytes, specifying the source/destination register (`EAX`) and the addressing mode (displacement in this case) and base register (`EBX`) for the second operand. This combination takes 1 byte to specify the operands. When in 32-bit mode (see Appendix K), the size of the address field is either 1 byte or 4 bytes. Because1000 is bigger than 2^8, the total length of the instruction is

$$1 + 1 + 4 = 6 \text{ bytes}$$

The length of 80x86 instructions varies between 1 and 17 bytes. 80x86 programs are generally smaller than the RISC architectures, which use fixed formats (see Appendix K).

Given these two poles of instruction set design of variable and fixed, the third alternative immediately springs to mind: reduce the variability in size and work of the variable architecture but provide multiple instruction lengths to reduce code size. This *hybrid* approach is the third encoding alternative, and we'll see examples shortly.

Reduced Code Size in RISCs

As RISC computers started being used in embedded applications, the 32-bit fixed format became a liability because cost, and hence smaller code, are important. In response, several manufacturers offered a new hybrid version of their RISC instruction sets, with both 16-bit and 32-bit instructions. The narrow instructions support fewer operations, smaller address and immediate fields, fewer registers, and the two-address format rather than the classic three-address format of RISC computers. RISC-V offers such an extension, called RV32IC, the C standing for compressed. Common instruction occurrences, such as intermediates with small values and common ALU operations with the source and destination register being identical, are encoded in 16-bit formats. Appendix K gives two other examples, the ARM Thumb and microMIPS, which both claim a code size reduction of up to 40%.

In contrast to these instruction set extensions, IBM simply compresses its standard instruction set and then adds hardware to decompress instructions as they are fetched from memory on an instruction cache miss. Thus, the instruction cache contains full 32-bit instructions, but compressed code is kept in main memory, ROMs, and the disk. The advantage of a compressed format, such as RV32IC, microMIPS and Thumb2 is that instruction caches act as if they are about 25% larger, while IBM's CodePack means that compilers need not be

changed to handle different instruction sets and instruction decoding can remain simple.

CodePack starts with run-length encoding compression on any PowerPC program and then loads the resulting compression tables in a 2 KB table on chip. Hence, every program has its own unique encoding. To handle branches, which are no longer to an aligned word boundary, the PowerPC creates a hash table in memory that maps between compressed and uncompressed addresses. Like a TLB (see Chapter 2), it caches the most recently used address maps to reduce the number of memory accesses. IBM claims an overall performance cost of 10%, resulting in a code size reduction of 35%–40%.

Summary: Encoding an Instruction Set

Decisions made in the components of instruction set design discussed in previous sections determine whether 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. RISC-V, MIPS, and ARM all have an instruction set extension that uses 16-bit instruction, as well as 32-bit; applications with serious code size constraints can opt to use the 16-bit variant to decrease code size. Appendix E gives 13 examples of the results of architects' choices. In Appendix C and Chapter 3, the impact of variability on performance of the processor will be discussed further.

We have almost finished laying the groundwork for the RISC-V instruction set architecture that will be introduced in Section A.9. Before we do that, however, it will be helpful to take a brief look at compiler technology and its effect on program properties.

A.8 Cross-Cutting Issues: The Role of Compilers

Today almost all programming is done in high-level languages for desktop and server applications. This development means that because most instructions executed are the output of a compiler, an instruction set architecture is essentially a compiler target. In earlier times for these applications, architectural decisions were often made to ease assembly language programming or for a specific kernel. Because the compiler will significantly affect the performance of a computer, understanding compiler technology today is critical to designing and efficiently implementing an instruction set.

Once 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 architecture from its implementation. This separation is essentially impossible with today's desktop compilers and computers. Architectural choices affect the quality of the code that can be generated for a computer and the complexity of building a good compiler for it, for better or for worse.

In this section, we discuss the critical goals in the instruction set primarily from the compiler viewpoint. It starts with a review of the anatomy of current compilers. Next we discuss how compiler technology affects the decisions of the architect, and how the architect can make it hard or easy for the compiler to produce good code. We conclude with a review of compilers and multimedia operations, which unfortunately is a bad example of cooperation between compiler writers and architects.

The Structure of Recent Compilers

To begin, let's look at what optimizing compilers are like today. Figure A.19 shows the structure of recent compilers.

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

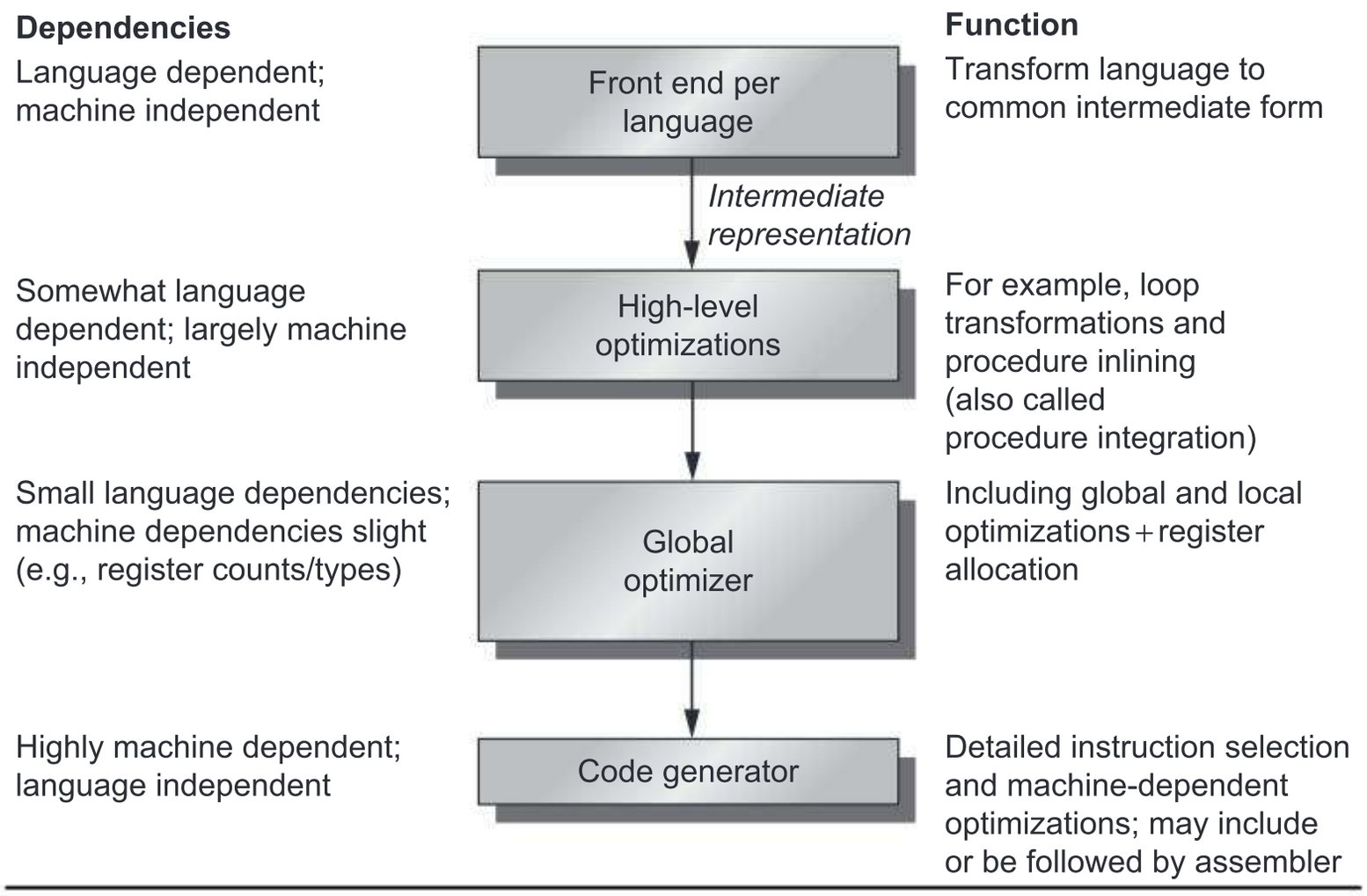


Figure A.19 Compilers typically consist of two to four passes, with more highly optimizing compilers having more passes. This structure maximizes the probability that a program compiled at various levels of optimization will produce the same output when given the same input. The optimizing passes are designed to be optional and may be skipped when faster compilation is the goal and lower-quality code is acceptable. 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*.) Because the optimizing passes are separated, multiple languages can use the same optimizing and code generation passes. Only a new front end is required for a new language.

progressively lower-level representations. Eventually it reaches 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 A.19, we can see that certain high-level optimizations are performed long before it is known what the resulting code will look like. Once such a transformation is made, the compiler can't afford to go back and revisit all steps, possibly undoing transformations. Such iteration 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 common 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:

- *High-level optimizations* are often done on the source with output fed to later optimization passes.
- *Local optimizations* optimize code only within a straight-line code fragment (called a *basic block* by compiler people).
- *Global optimizations* extend the local optimizations across branches and introduce a set of transformations aimed at optimizing loops.
- *Register allocation* associates registers with operands.
- *Processor-dependent optimizations* attempt to take advantage of specific architectural knowledge.

Register Allocation

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—of the optimizations. Register allocation algorithms today 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. Roughly speaking, the problem is how to use a limited set of colors so that no two adjacent nodes in a dependency graph have the same color. The emphasis in the approach is to achieve 100% register allocation of active variables. The problem of coloring a graph in general can take exponential time as a function of the size of the graph (NP-complete). There are heuristic algorithms, however, that work well in practice, yielding close allocations that run in near-linear time.

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.

Impact of Optimizations on Performance

It is sometimes difficult to separate some of the simpler optimizations—local and processor-dependent optimizations—from transformations done in the code generator. Examples of typical optimizations are given in Figure A.20. The last column of Figure A.20 indicates the frequency with which the listed optimizing transforms were applied to the source program.

Figure A.21 shows the effect of various optimizations on instructions executed for two programs. In this case, optimized programs executed roughly 25%–90% fewer instructions than unoptimized programs. The figure illustrates the importance of looking at optimized code before suggesting new instruction set features, because a compiler might completely remove the instructions the architect was trying to improve.

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 or shrunk on procedure call or return, respectively. Objects on the stack are addressed

Optimization name	Explanation	Percentage of the total number of optimizing transforms
High-level	*At or near the source level; processor-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%
Processor-dependent	*Depends on processor 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 A.20 Major types of optimizations and examples in each class. These data tell us about the relative frequency of occurrence of various optimizations. 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. 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. Processor-dependent optimizations are usually done in a code generator, and none of those was measured in this experiment. The percentage is the portion of the static optimizations that are of the specified type. Data from Chow, F.C., 1983. A Portable Machine-Independent Global Optimizer—Design and Measurements (Ph.D. thesis). Stanford University, Palo Alto, CA (collected using the Stanford UCODE compiler).

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.

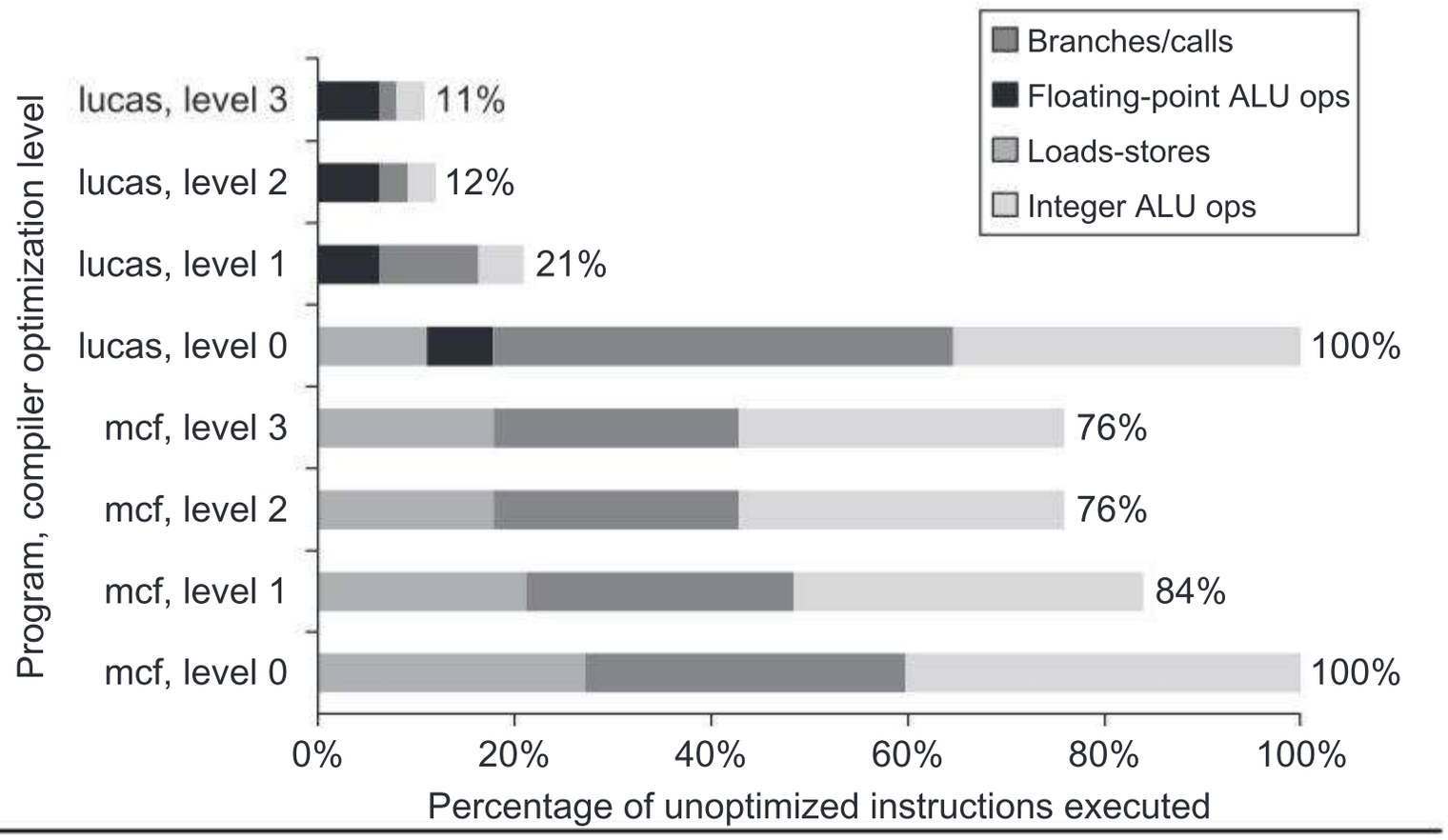


Figure A.21 Change in instruction count for the programs lucas and mcf from the SPEC2000 as compiler optimization levels vary. Level 0 is the same as unoptimized code. Level 1 includes local optimizations, code scheduling, and local register allocation. Level 2 includes global optimizations, loop transformations (software pipelining), and global register allocation. Level 3 adds procedure integration. These experiments were performed on Alpha compilers.

- 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.

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*—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; some 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 decisions are made one step at a time about which code sequence is best.

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.

- *Provide 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 one addressing mode can be applied, all addressing modes are applicable. This regularity 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. Compilers for special-purpose register architectures typically get stuck in this dilemma. This restriction can result in the compiler finding itself with lots of available registers, but none of the right kind!
- *Provide primitives, not solutions*—Special features that "match" a language construct or a kernel function 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. An example of how such attempts have failed is given in Section A.10.
- *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 Chapter 1—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.

- *Provide instructions that bind the quantities known at compile time as constants*—A compiler writer hates the thought of the processor 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 (see Section A.10).

Compiler Support (or Lack Thereof) for Multimedia Instructions

Alas, the designers of the SIMD instructions (see Section 4.3 in Chapter 4) basically ignored the previous subsection. These instructions tend to be solutions, not primitives; they are short of registers; and the data types do not match existing programming languages. Architects hoped to find an inexpensive solution that would help some users, but often only a few low-level graphics library routines use them.

The SIMD instructions are really an abbreviated version of an elegant architecture style that has its own compiler technology. As explained in Section 4.2, *vector architectures* operate on vectors of data. Invented originally for scientific codes, multimedia kernels are often vectorizable as well, albeit often with shorter vectors. As Section 4.3 suggests, we can think of Intel's MMX and SSE or PowerPC's AltiVec, or the RISC-V P extension, as simply short vector computers: MMX with vectors of eight 8-bit elements, four 16-bit elements, or two 32-bit elements, and AltiVec with vectors twice that length. They are implemented as simply adjacent, narrow elements in wide registers.

These microprocessor architectures build the vector register size into the architecture: the sum of the sizes of the elements is limited to 64 bits for MMX and 128 bits for AltiVec. When Intel decided to expand to 128-bit vectors, it added a whole new set of instructions, called streaming SIMD extension (SSE).

A major advantage of vector computers is hiding latency of memory access by loading many elements at once and then overlapping execution with data transfer. The goal of vector addressing modes is to collect data scattered about memory, place them in a compact form so that they can be operated on efficiently, and then place the results back where they belong.

Vector computers include *strided addressing* and *gather/scatter addressing* (see Section 4.2) to increase the number of programs that can be vectorized. Strided addressing skips a fixed number of words between each access, so sequential addressing is often called *unit stride addressing*. Gather and scatter find their

addresses in another vector register: think of it as register indirect addressing for vector computers. From a vector perspective, in contrast, these short-vector SIMD computers support only unit strided accesses: memory accesses load or store all elements at once from a single wide memory location. Because the data for multimedia applications are often streams that start and end in memory, strided and gather/scatter addressing modes are essential to successful vectorization (see Section 4.7).

Example As an example, compare a vector computer with MMX for color representation conversion of pixels from RGB (red, green, blue) to YUV (luminosity chrominance), with each pixel represented by 3 bytes. The conversion is just three lines of C code placed in a loop:

```
Y = (9798*R + 19235*G + 3736*B) / 32768;
U = (-4784*R 9437*G + 4221*B) / 32768 + 128;
V = (20218*R 16941*G 3277*B) / 32768 + 128;
```

A 64-bit-wide vector computer can calculate 8 pixels simultaneously. One vector computer for media with strided addresses takes

- 3 vector loads (to get RGB)
- 3 vector multiplies (to convert R)
- 6 vector multiply adds (to convert G and B)
- 3 vector shifts (to divide by 32,768)
- 2 vector adds (to add 128)
- 3 vector stores (to store YUV)

The total is 20 instructions to perform the 20 operations in the previous C code to convert 8 pixels (Kozyrakis, 2000). (Because a vector might have 32 64-bit elements, this code actually converts up to 32×8 or 256 pixels.)

In contrast, Intel's website shows that a library routine to perform the same calculation on 8 pixels takes 116 MMX instructions plus 6 80x86 instructions (Intel, 2001). This six-fold increase in instructions is due to the large number of instructions to load and unpack RGB pixels and to pack and store YUV pixels, because there are no strided memory accesses.

Having short, architecture-limited vectors with few registers and simple memory addressing modes makes it more difficult to use vectorizing compiler technology. Hence, these SIMD instructions are more likely to be found in hand-coded libraries than in compiled code.

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—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. Alas, SIMD extensions are more an example of good marketing than of outstanding achievement of hardware–software co-design.

A.9 Putting It All Together: The RISC-V Architecture

In this section we describe the load-store architecture called RISC-V. RISC-V is a freely licensed open standard, similar to many of the RISC architectures, and based on observations similar to those covered in the last sections. (In Section M.3 we discuss how and why these architectures became popular.) RISC-V builds on 30 years of experience with RISC architectures and "cleans up" most of the short-term inclusions and omissions, leading to an architecture that is easier and more efficient to implement. RISC-V provides a both a 32-bit and a 64-bit instruction set, as well as a variety of extensions for features like floating point; these extensions can be added to either the 32-bit or 64-bit base instruction set. We discuss a 64-bit version of RISC-V, RV64, which is a superset of the 32-bit version RV32.

Reviewing our expectations from each section, for desktop and server applications:

- Section A.2—Use general-purpose registers with a load-store architecture.
- Section A.3—Support these addressing modes: displacement (with an address offset size of 12–16 bits), immediate (size 8–16 bits), and register indirect.
- Section A.4—Support these data sizes and types: 8-, 16-, 32-, and 64-bit integers and 64-bit IEEE 754 floating-point numbers.
- Section A.5—Support these simple instructions, because they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and shift.
- Section A.6—Compare equal, compare not equal, compare less, branch (with a PC-relative address at least 8 bits long), jump, call, and return.
- Section A.7—Use fixed instruction encoding if interested in performance, and use variable instruction encoding if interested in code size. In some low-end, embedded applications, with small or only one-level caches, larger code size may have significant performance implications. ISAs that provide a compressed instruction set extension provide a way of addressing this difference.
- Section A.8—Provide at least 16, and preferably 32, general-purpose registers, be sure all addressing modes apply to all data transfer instructions, and aim for

a minimalist instruction set. This section didn't cover floating-point programs, but they often use separate floating-point registers. The justification is to increase the total number of registers without raising problems in the instruction format or in the speed of the general-purpose register file. This compromise, however, is not orthogonal.

We introduce RISC-V by showing how it follows these recommendations. Like its RISC predecessors, RISC-V emphasizes

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

RISC-V provides a good architectural model for study, not only because of the popularity of this type of processor, but also because it is an easy architecture to understand. We will use this architecture again in Appendix C and in Chapter 3, and it forms the basis for a number of exercises and programming projects.

RISC-V Instruction Set Organization

The RISC-V instruction set is organized as three base instruction sets that support 32-bit or 64-bit integers, and a variety of optional extensions to one of the base instruction sets. This allows RISC-V to be implemented for a wide range of potential applications from a small embedded processor with a minimal budget for logic and memory that likely costs $1 or less, to high-end processor configurations with full support for floating point, vectors, and multiprocessor configurations. Figure A.22 summarizes the three base instruction sets and the instruction set extensions with their basic functionality. For purposes of this text, we use RV64IMAFD (also known as RV64G, for short) in examples. RV32G is the 32-bit subset of the 64-bit architecture RV64G.

Registers for RISC-V

RV64G has 32 64-bit general-purpose registers (GPRs), named x0, x1, … , x31. GPRs are also sometimes known as *integer registers*. Additionally, with the F and D extensions for floating point that are part of RV64G, come a set of 32 floating-point registers (FPRs), named f0, f1, … , f31, which can hold 32 single-precision (32-bit) values or 32 double-precision (64-bit) values. (When holding one single-precision number, the other half of the FPR is unused.) Both single- and double-precision floating-point operations (32-bit and 64-bit) are provided.

The value of x0 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 general-purpose registers. An example is the floating-point status register, used to hold

Name of base or extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers
RV32E	Base 32-bit instruction set but with only 16 registers; intended for very low-end embedded applications
RV64I	Base 64-bit instruction set; all registers are 64-bits, and instructions to move 64-bit from/to the registers (LD and SD) are added
M	Adds integer multiply and divide instructions
A	Adds atomic instructions needed for concurrent processing; see Chapter 5
F	Adds single precision (32-bit) IEEE floating point, includes 32 32-bit floating point registers, instructions to load and store those registers and operate on them
D	Extends floating point to double precision, 64-bit, making the registers 64-bits, adding instructions to load, store, and operate on the registers
Q	Further extends floating point to add support for quad precision, adding 128-bit operations
L	Adds support for 64- and 128-bit decimal floating point for the IEEE standard
C	Defines a compressed version of the instruction set intended for small-memory-sized embedded applications. Defines 16-bit versions of common RV32I instructions
V	A future extension to support vector operations (see Chapter 4)
B	A future extension to support operations on bit fields
T	A future extension to support transactional memory
P	An extension to support packed SIMD instructions: see Chapter 4
RV128I	A future base instruction set providing a 128-bit address space

Figure A.22 RISC-V has three base instructions sets (and a reserved spot for a future fourth); all the extensions extend one of the base instruction sets. An instruction set is thus named by the base name followed by the extensions. For example, RISC-V64IMAFD refers to the base 64-bit instruction set with extensions M, A, F, and D. For consistency of naming and software, this combination is given the abbreviated name: RV64G, and we use RV64G through most of this text.

information about the results of floating-point operations. There are also instructions for moving between an FPR and a GPR.

Data Types for RISC-V

The data types are 8-bit bytes, 16-bit half words, 32-bit words, and 64-bit double-words for integer data and 32-bit single precision and 64-bit double precision for floating point. Half words were added because they are found in languages like C

and are popular in some programs, such as the operating systems, concerned about size of data structures. They will also become more popular if Unicode becomes widely used.

The RV64G operations work on 64-bit integers and 32- or 64-bit floating point. Bytes, half words, and words are loaded into the general-purpose registers with either zeros or the sign bit replicated to fill the 64 bits of the GPRs. Once loaded, they are operated on with the 64-bit integer operations.

Addressing Modes for RISC-V Data Transfers

The only data addressing modes are immediate and displacement, both with 12-bit fields. Register indirect is accomplished simply by placing 0 in the 12-bit displacement field, and limited absolute addressing with a 12-bit field is accomplished by using register 0 as the base register. Embracing zero gives us four effective modes, although only two are supported in the architecture.

RV64G memory is byte addressable with a 64-bit address and uses Little Endian byte numbering. As it is a load-store architecture, all references between memory and either GPRs or FPRs are through loads or stores. Supporting the data types mentioned herein, memory accesses involving GPRs can be to a byte, half word, word, or double word. The FPRs may be loaded and stored with single-precision or double-precision numbers. Memory accesses need not be aligned; however, it may be that unaligned accesses run extremely slow. In practice, programmers and compilers would be stupid to use unaligned accesses.

RISC-V Instruction Format

Because RISC-V has just two addressing modes, these can be encoded into the opcode. Following the advice on making the processor easy to pipeline and decode, all instructions are 32 bits with a 7-bit primary opcode. Figure A.23 shows the instruction layout of the four major instruction types. These formats are simple while providing 12-bit fields for displacement addressing, immediate constants, or PC-relative branch addresses.

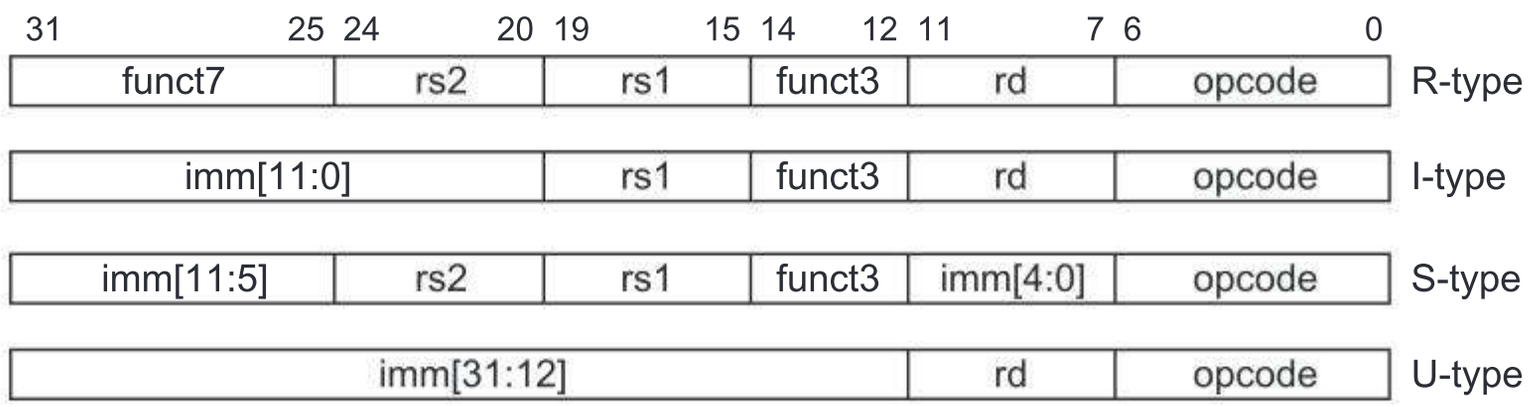


Figure A.23 The RISC-V instruction layout. There are two variations on these formata, called the SB and UJ formats; they deal with a slightly different treatment for immediate fields.

Instruction format	Primary use	rd	rs1	rs2	Immediate
R-type	Register-register ALU instructions	Destination	First source	Second source	
I-type	ALU immediates Load	Destination	First source base register		Value displacement
S-type	Store Compare and branch		Base register first source	Data source to store second source	Displacement offset
U-type	Jump and link Jump and link register	Register destination for return PC	Target address for jump and link register		Target address for jump and link

Figure A.24 The use of instruction fields for each instruction type. Primary use shows the major instructions that use the format. A blank indicates that the corresponding field is not present in this instruction type. The I-format is used for both loads and ALU immediates, with the 12-bit immediate holding either the value for an immediate or the displacement for a load. Similarly, the S-format encodes both store instructions (where the first source register is the base register and the second contains the register source for the value to store) and compare and branch instructions (where the register fields contain the sources to compare and the immediate field specifies the offset of the branch target). There are actually two other formats: SB and UJ that follow the same basic organization as S and J, but slightly modify the interpretation of the immediate fields.

The instruction formats and the use of the instruction fields is described in Figure A.24. The opcode specifies the general instruction type (ALU instruction, ALU immediate, load, store, branch, or jump), while the funct fields are used for specific operations. For example, an ALU instruction is encoded with a single opcode with the funct field dictating the exact operation: add, subtract, and, etc. Notice that several formats encode multiple types of instructions, including the use of the I-format for both ALU immediates and loads, and the use of the S-format for stores and conditional branches.

RISC-V Operations

RISC-V (or more properly RV64G) supports the list of simple operations recommended herein 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 x0 has no effect. Figure A.25 gives examples of the load and store instructions. Single-precision floating-point numbers occupy half a floating-point register. Conversions between single and double precision must be done explicitly. The floating-point format is IEEE 754 (see Appendix J). A list of all the RV64G instructions appears in Figure A.28 (page A.42).

Example instruction	Instruction name	Meaning
`ld x1,80(x2)`	Load doubleword	Regs[x1] ← Mem[80+Regs[x2]]
`lw x1,60(x2)`	Load word	Regs[x1] $\leftarrow_{64}$ Mem[60+Regs[x2]]$_0$)^{32} ## Mem[60+Regs[x2]]
`lwu x1,60(x2)`	Load word unsigned	Regs[x1] $\leftarrow_{64}$ 0^{32} ## Mem[60+Regs[x2]]
`lb x1,40(x3)`	Load byte	Regs[x1] $\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{56} ## Mem[40+Regs[x3]]
`lbu x1,40(x3)`	Load byte unsigned	Regs[x1] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[x3]]
`lh x1,40(x3)`	Load half word	Regs[x1] $\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{48} ## Mem[40+Regs[x3]]
`flw f0,50(x3)`	Load FP single	Regs[f0] $\leftarrow_{64}$ Mem[50+Regs[x3]] ## 0^{32}
`fld f0,50(x2)`	Load FP double	Regs[f0] $\leftarrow_{64}$ Mem[50+Regs[x2]]
`sd x2,400(x3)`	Store double	Mem[400+Regs[x3]] $\leftarrow_{64}$ Regs[x2]
`sw x3,500(x4)`	Store word	Mem[500+Regs[x4]] $\leftarrow_{32}$ Regs[x3]$_{32..63}$
`fsw f0,40(x3)`	Store FP single	Mem[40+Regs[x3]] $\leftarrow_{32}$ Regs[f0]$_{0..31}$
`fsd f0,40(x3)`	Store FP double	Mem[40+Regs[x3]] $\leftarrow_{64}$ Regs[f0]
`sh x3,502(x2)`	Store half	Mem[502+Regs[x2]] $\leftarrow_{16}$ Regs[x3]$_{48..63}$
`sb x2,41(x3)`	Store byte	Mem[41+Regs[x3]] $\leftarrow_{8}$ Regs[x2]$_{56..63}$

Figure A.25 The load and store instructions in RISC-V. Loads shorter than 64 bits are available in both sign-extended and zero-extended forms. All memory references use a single addressing mode. Of course, both loads and stores are available for all the data types shown. Because RV64G supports double precision floating point, all single precision floating point loads must be aligned in the FP register, which are 64-bits wide.

To understand these figures we need to introduce a few additional extensions to our C description language used initially on page A-9:

- A subscript is appended to the symbol ← 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[x4]$_0$ yields the sign bit of x4) or a subrange (e.g., Regs[x3]$_{56..63}$ yields the least-significant byte of x3).
- 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^{48} yields a field of zeros of length 48 bits).
- The symbol ## is used to concatenate two fields and may appear on either side of a data transfer, and the symbols $\ll$ and $\gg$ shift the first operand left or right by the amount of the second operand.

Example instrucmtion	Instruction name	Meaning
`add x1,x2,x3`	Add	`Regs[x1]←Regs[x2]+Regs[x3]`
`addi x1,x2,3`	Add immediate unsigned	`Regs[x1]←Regs[x2]+3`
`lui x1,42`	Load upper immediate	$\texttt{Regs[x1]} \leftarrow 0^{32}\#\#42\#\#0^{12}$
`sll x1,x2,5`	Shift left logical	`Regs[x1]←Regs[x2]<<5`
`slt x1,x2,x3`	Set less than	`if (Regs[x2]<Regs[x3]) Regs[x1]←1 else Regs[x1]←0`

Figure A.26 The basic ALU instructions in RISC-V are available both with register-register operands and with one immediate operand. LUI uses the U-format that employs the rs1 field as part of the immediate, yielding a 20-bit immediate.

As an example, assuming that x8 and x10 are 32-bit registers:

$$\texttt{Regs[x10]} \leftarrow_{64} (\texttt{Mem[Regs[x8]]}_0)^{32} \#\# \texttt{Mem[Regs[R8]]}$$

means that the word at the memory location addressed by the contents of register x8 is sign-extended to form a 64-bit quantity that is stored into register x10.

All ALU instructions are register-register instructions. Figure A.26 gives some examples of the arithmetic/logical instructions. The operations include simple arithmetic and logical operations: add, subtract, AND, OR, XOR, and shifts. Immediate forms of all these instructions are provided using a 12-bit sign-extended immediate. The operation `LUI` (load upper immediate) loads bits 12–31 of a register, sign-extends the immediate field to the upper 32-bits, and sets the low-order 12-bits of the register to 0. `LUI` allows a 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 herein, x0 is used to synthesize popular operations. Loading a constant is simply an add immediate where the source operand is x0, and a register-register move is simply an add (or an or) where one of the sources is x0. (We sometimes use the mnemonic li, standing for load immediate, to represent the former, and the mnemonic mv for the latter.)

RISC-V Control Flow Instructions

Control is handled through a set of jumps and a set of branches, and Figure A.27 gives some typical branch and jump instructions. The two jump instructions (jump and link and jump and link register) are unconditional transfers and always store the "link," which is the address of the instruction sequentially following the jump instruction, in the register specified by the rd field. In the event that the link address is not needed, the rd field can simply be set to x0, which results in a typical unconditional jump. The two jump instructions are differentiated by whether the address is computed by adding an immediate field to the PC or b y adding the

Example instruction	Instruction name	Meaning
`jal x1,offset`	Jump and link	`Regs[x1]←PC+4; PC←PC+(offset<<1)`
`jalr x1,x2,offset`	Jump and link register	`Regs[x1]←PC+4; PC←Regs[x2]+offset`
`beq x3,x4,offset`	Branch equal zero	`if (Regs[x3]==Regs[x4]) PC←PC+(offset<<1)`
`bgt x3,x4,name`	Branch not equal zero	`if (Regs[x3]>Regs[x4]) PC←PC+(offset<<1)`

Figure A.27 Typical control flow instructions in RISC-V. All control instructions, except jumps to an address in a register, are PC-relative.

immediate field to the contents of a register. The offset is interpreted as a half word offset for compatibility with the compressed instruction set, R64C, which includes 16-bit instructions.

All branches are conditional. The branch condition is specified by the instruction, and any arithmetic comparison (equal, greater than, less than, and their inverses) is permitted. The branch-target address is specified with a 12-bit signed offset that is shifted left one place (to get 16-bit alignment) and then added to the current program counter. Branches based on the contents of the floating point registers are implemented by executing a floating point comparison (e.g., feq.d or fle.d), which sets an integer register to 0 or 1 based on the comparison, and then executing a beq or bne with x0 as an operand.

The observant reader will have noticed that there are very few 64-bit only instructions in RV64G. Primarily, these are the 64-bit loads and stores and versions of 32-bit, 16-bit, and 8-bit loads that do not sign extend (the default is to sign-extend). To support 32-bit modular arithmetic without additional instructions, there are versions of the instructions that ignore the upper 32 bits of a 64-bit register, such as add and subtract word (addw, subw). Amazingly, everything else just works.

RISC-V Floating-Point Operations

Floating-point instructions manipulate the floating-point registers and indicate whether the operation to be performed is single or double precision. The floating-point operations are add, subtract, multiply, divide, square root, as well as fused multiply-add and multiply-subtract. All floating point instructions begin with the letter `f` and use the suffix `d` for double precision and `s` for single precision (e.g., `fadd.d, fadd.s, fmul.d, fmul.s, fmadd.d fmadd.s`). Floating-point compares set an integer register based on the comparison, similarly to the integer instruction set-less-than and set-great-than.

In addition to floating-point loads and stores (`flw, fsw, fld, fsd`), instructions are provided for converting between different FP precisions, for moving between integer and FP registers (fmv), and for converting between floating point and integer (fcvt, which uses the integer registers for source or destination as appropriate).

Figure A.28 contains a list of nearly all the RV64G instructions and a summary of their meaning.

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP; only memory address mode is 12-bit displacement + contents of a GPR*
`lb, lbu, sb`	Load byte, load byte unsigned, store byte (to/from integer registers)
`lh, lhu, sh`	Load half word, load half word unsigned, store half word (to/from integer registers)
`lw, lwu, sw`	Load word, store word (to/from integer registers)
`ld, sd`	Load doubleword, store doubleword
Arithmetic/logical	*Operations on data in GPRs. Word versions ignore upper 32 bits*
`add, addi, addw, addiw, sub, subi, subw, subiw`	Add and subtract, with both word and immediate versions
`slt, sltu, slti, sltiu`	set-less-than with signed and unsigned, and immediate
`and, or, xor, andi, ori, xori`	and, or, xor, both register-register and register-immediate
`lui`	Load upper immediate: loads bits 31..12 of a register with the immediate value. Upper 32 bits are set to 0
`auipc`	Sums an immediate and the upper 20-bits of the PC into a register; used for building a branch to any 32-bit address
`sll, srl, sra, slli, srli, srai, sllw,slliw, srli, srliw, srai, sraiw`	Shifts: logical shift left and right and arithmetic shift right, both immediate and word versions (word versions leave the upper 32 bit untouched)
`mul, mulw, mulh, mulhsu, mulhu, div,divw, divu, rem, remu, remw, remuw`	Integer multiply, divide, and remainder, signed and unsigned with support for 64-bit products in two instructions. Also word versions
Control	*Conditional branches and jumps; PC-relative or through register*
`beq, bne, blt, bge, bltu, bgeu`	Branch based on compare of two registers, equal, not equal, less than, greater or equal, signed and unsigned
`jal,jalr`	Jump and link address relative to a register or the PC
Floating point	*All FP operation appear in double precision (.d) and single (.s)*
`flw, fld, fsw, fsd`	Load, store, word (single precision), doubleword (double precision)
`fadd, fsub, fmult, fiv, fsqrt, fmadd, fmsub, fnmadd, fnmsub, fmin, fmax, fsgn, fsgnj, fsjnx`	Add, subtract, multiply, divide, square root, multiply-add, multiply-subtract, negate multiply-add, negate multiply-subtract, maximum, minimum, and instructions to replace the sign bit. For single precision, the opcode is followed by: .s, for double precision: .d. Thus fadd.s, fadd.d
`feq, flt, fle`	Compare two floating point registers; result is 0 or 1 stored into a GPR
`fmv.x.*, fmv.*.x`	Move between the FP register abd GPR, "*" is s or d
`fcvt.*.l, fcvt.l.*, fcvt.*.lu, fcvt.lu.*, fcvt.*.w, fcvt.w.*, fcvt.*.wu, fcvt.wu.*`	Converts between a FP register and integer register, where "*" is S or D for single or double precision. Signed and unsigned versions and word, doubleword versions

Figure A.28 A list of the vast majority of instructions in RV64G. This list can also be found on the back inside cover. This table omits system instructions, synchronization and atomic instructions, configuration instructions, instructions to reset and access performance counters, about 10 instructions in total.

Program	Loads	Stores	Branches	Jumps	ALU operations
astar	28%	6%	18%	2%	46%
bzip	20%	7%	11%	1%	54%
gcc	17%	23%	20%	4%	36%
gobmk	21%	12%	14%	2%	50%
h264ref	33%	14%	5%	2%	45%
hmmer	28%	9%	17%	0%	46%
libquantum	16%	6%	29%	0%	48%
mcf	35%	11%	24%	1%	29%
omnetpp	23%	15%	17%	7%	31%
perlbench	25%	14%	15%	7%	39%
sjeng	19%	7%	15%	3%	56%
xalancbmk	30%	8%	27%	3%	31%

Figure A.29 RISC-V dynamic instruction mix for the SPECint2006 programs. Omnetpp includes 7% of the instructions that are floating point loads, stores, operations, or compares; no other program includes even 1% of other instruction types. A change in gcc in SPECint2006, creates an anomaly in behavior. Typical integer programs have load frequencies that are 1/5 to 3x the store frequency. In gcc, the store frequency is actually higher than the load frequency! This arises because a large fraction of the execution time is spent in a loop that clears memory by storing x0 (not where a compiler like gcc would usually spend most of its execution time!). A store instruction that stores a register pair, which some other RISC ISAs have included, would address this issue.

RISC-V Instruction Set Usage

To give an idea of which instructions are popular, Figure A.29 shows the frequency of instructions and instruction classes for the SPECint2006 programs, using RV32G.

A.10 Fallacies and Pitfalls

Architects have repeatedly 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. However, often these instructions do more work than is required in the frequent case, or they don't exactly match the requirements of some languages. 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 et al. (1981) have called a *semantic clash*:

> … by giving too much semantic content to the instruction, the computer 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 A.8). The mask is kept in the called procedure's code—this permits the callee to specify the registers to be saved by the caller even with separate compilation.
4. Push the return address on the stack, and then push the top and base of stack pointers (for the activation record).
5. Clear the condition codes, which sets the trap enable 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 in memory. 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 because 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 computers 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.*

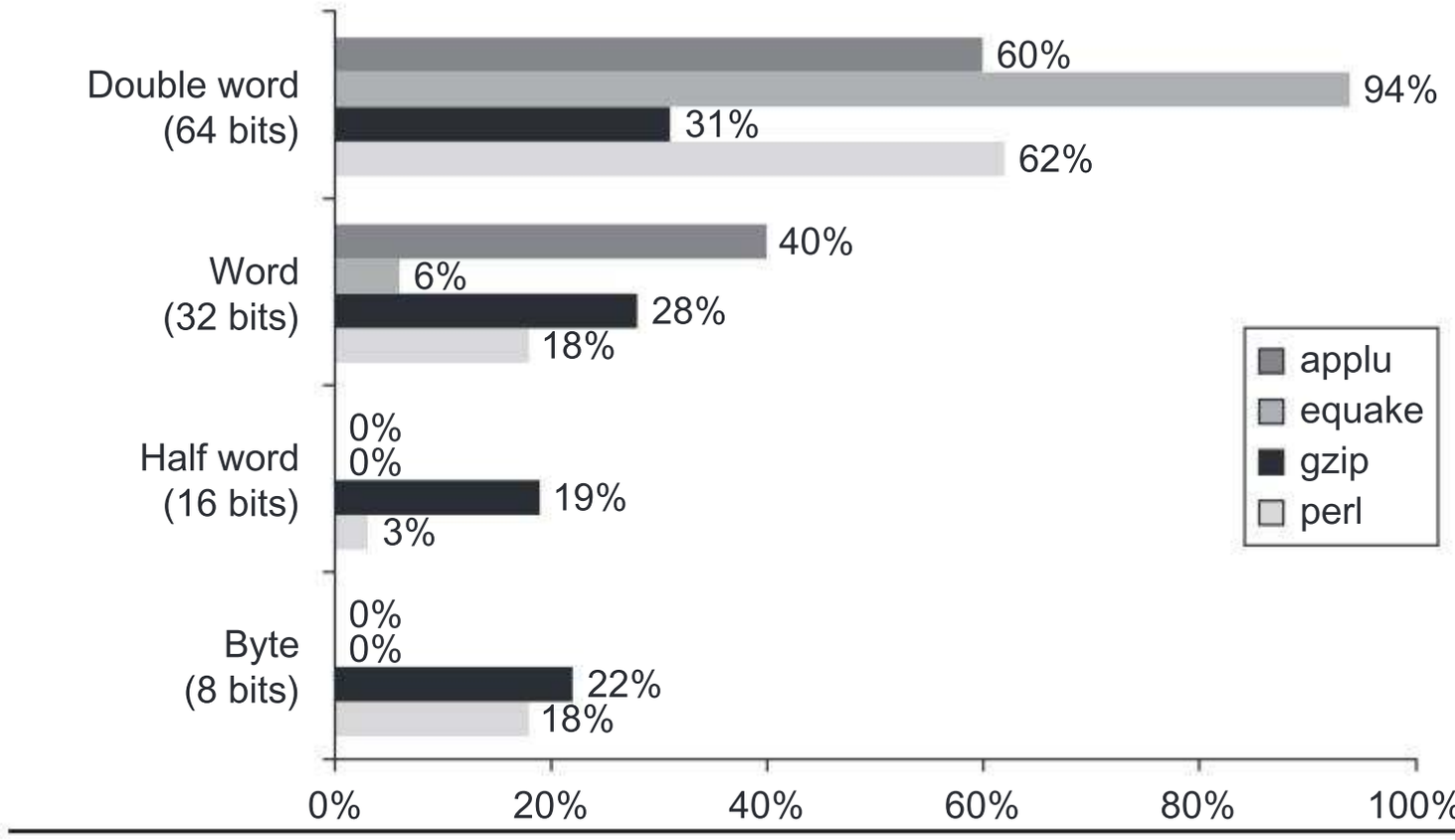


Figure A.30 Data reference size of four programs from SPEC2000. Although you can calculate an average size, it would be hard to claim the average is typical of programs.

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 appendix clearly show that programs can vary significantly in how they use an instruction set. For example, Figure A.30 shows the mix of data transfer sizes for four of the SPEC2000 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.

Pitfall *Innovating at the instruction set architecture to reduce code size without accounting for the compiler.*

Figure A.31 shows the relative code sizes for four compilers for the MIPS instruction set. Whereas architects struggle to reduce code size by 30%–40%, different compiler strategies can change code size by much larger factors. Similar to performance optimization techniques, the architect should start with the tightest code the compilers can produce before proposing hardware innovations to save space.

Fallacy *An architecture with flaws cannot be successful.*

The 80x86 provides a dramatic example: the instruction set architecture is one only its creators could love (see Appendix K). 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; it uses extended accumulators for integer data, but other processors use general-purpose registers; and it uses a stack for floating-point data, when everyone else abandoned execution stacks long before.

Compiler	Apogee software version 4.1	Green Hills Multi2000 Version 2.0	Algorithmics SDE4.0B	IDT/c 7.2.1
Architecture	MIPS IV	MIPS IV	MIPS 32	MIPS 32
Processor	NEC VR5432	NEC VR5000	IDT 32334	IDT 79RC32364
Autocorrelation kernel	1.0	2.1	1.1	2.7
Convolutional encoder kernel	1.0	1.9	1.2	2.4
Fixed-point bit allocation kernel	1.0	2.0	1.2	2.3
Fixed-point complex FFT kernel	1.0	1.1	2.7	1.8
Viterbi GSM decoder kernel	1.0	1.7	0.8	1.1
Geometric mean of five kernels	1.0	1.7	1.4	2.0

Figure A.31 Code size relative to Apogee Software Version 4.1 C compiler for Telecom application of EEMBC benchmarks. The instruction set architectures are virtually identical, yet the code sizes vary by factors of 2. These results were reported February—June 2000.

Despite these major difficulties, the 80x86 architecture has been enormously successful. The reasons are threefold: first, its selection as the microprocessor in the initial IBM PC makes 80x86 binary compatibility extremely valuable. Second, Moore's Law provided sufficient resources for 80x86 microprocessors to translate to an internal RISC instruction set and then execute RISC-like instructions. This mix enables binary compatibility with the valuable PC software base and performance on par with RISC processors. Third, the very high volumes of PC microprocessors mean Intel can easily pay for the increased design cost of hardware translation. In addition, the high volumes allow the manufacturer to go up the learning curve, which lowers the cost of the product.

The larger die size and increased power for translation may be a liability for embedded applications, but it makes tremendous economic sense for the desktop. And its cost-performance in the desktop also makes it attractive for servers, with its main weakness for servers being 32-bit addresses, which was resolved with a 64-bit address extension.

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 five years later. An example in the RISC camp is delayed branch (see Appendix K). It was a simple matter to control pipeline hazards with five-stage pipelines, but a challenge for processors with longer pipelines that issue multiple instructions per clock cycle. In addition, almost all architectures eventually

succumb to the lack of sufficient address space. This is one reason that RISC-V has planned for the possibility of 128-bit addresses, although it may be decades before such capability is needed.

In general, avoiding such flaws in the long run would probably mean compromising the efficiency of the architecture in the short run, which is dangerous, since a new instruction set architecture must struggle to survive its first few years.

A.11 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, computer 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 processor performance saw a return to simpler architectures, based mainly on the load-store style of computer.

The following instruction set architecture changes occurred in the 1990s:

- *Address size doubles*—The 32-bit address instruction sets for most desktop and server processors were extended to 64-bit addresses, expanding the width of the registers (among other things) to 64 bits. Appendix K gives three examples of architectures that have gone from 32 bits to 64 bits.
- *Optimization of conditional branches via conditional execution*—In Chapter 3, we see that conditional branches can limit the performance of aggressive computer designs. Hence, there was interest in replacing conditional branches with conditional completion of operations, such as conditional move (see Appendix H), which was added to most instruction sets.
- *Optimization of cache performance via prefetch*—Chapter 2 explains the increasing role of memory hierarchy in the performance of computers, with a cache miss on some computers taking as many instruction times as page faults took on earlier computers. Hence, prefetch instructions were added to try to hide the cost of cache misses by prefetching (see Chapter 2).
- *Support for multimedia*—Most desktop and embedded instruction sets were extended with support for multimedia applications.

- *Faster floating-point operations*—Appendix J describes operations added to enhance floating-point performance, such as operations that perform a multiply and an add and paired single execution, which are part of RISC-V.

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 computers. 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 not just the processor—and hence there are plenty of topics for the architect to study. In fact, the material in this appendix was a central point of the book in its first edition in 1990, but now is included in an appendix primarily as reference material!

Appendix K may satisfy readers interested in instruction set architecture; it describes a variety of instruction sets, which are either important in the marketplace today or historically important, and it compares nine popular load-store computers with RISC-V.

A.12 Historical Perspective and References

Section M.4 (available online) features a discussion on the evolution of instruction sets and includes references for further reading and exploration of related topics.

Exercises by Gregory D. Peterson

A.1 [10]<A.9>Compute the effective CPI for an implementation of an embedded RISC-V CPU using Figure A.29. Assume we have made the following measurements of average CPI for each of the instruction types:

Instruction	Clock cycles
All ALU operations	1.0
Loads	5.0
Stores	3.0
Branches	
Taken	5.0
Not taken	3.0
Jumps	3.0

Average the instruction frequencies of astar and gcc to obtain the instruction mix.

A.2 [10] < A.9 > Compute the effective CPI for RISC-V using Figure A.29 and the table above. Average the instruction frequencies of bzip and hmmer to obtain the instruction mix. You may assume that all other instructions (for instructions not accounted for by the types in Table A.29) require 3.0 clock cycles each.

A.3 [10] < A.9 > Compute the effective CPI for an implementation of a RISC-V CPU using Figure A.29. Assume we have made the following measurements of average CPI for each of the instruction types:

Instruction	Clock cycles
All ALU operations	1.0
Loads	3.5
Stores	2.8
Branches	
Taken	4.0
Not taken	2.0
Jumps	2.4

Average the instruction frequencies of gobmk and mcf to obtain the instruction mix. You may assume that all other instructions (for instructions not accounted for by the types in Table A.29) require 3.0 clock cycles each.

A.4 [10] < A.9 > Compute the effective CPI for RISC-V using Figure A.29 and the table above. Average the instruction frequencies of perlbench and sjeng to obtain the instruction mix.

A.5 [10] < A.8 > Consider this high-level code sequence of three statements:

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

Use the technique of copy propagation (see Figure A.20) to transform the code sequence to the point where no operand is a computed value. Note the instances in which the transformation has reduced the computational work of a statement and those cases where the work has increased. What does this suggest about the technical challenge faced in trying to satisfy the desire for optimizing compilers?

A.6 [30] < A.8 > Compiler optimizations may result in improvements to code size and/or performance. Consider one or more of the benchmark programs from the SPEC CPU2017 or the EEMBC benchmark suites. Use the RISC-V processor or a processor available to you along with the GNU C compiler to optimize the benchmark program(s) using no optimization, −O1, −O2, and −O3. Compare the performance and size of the resulting programs. Also compare your results to Figure A.21.

A.7 [20/20/20/25/10]<A.2, A.9>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 64-bit integers, and C and i are 64-bit integers. Assume that all data values and their addresses are kept in memory (at addresses 1000, 3000, 5000, and 7000 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. Assume all addresses and words are 64 bits.

a. [20]<A.2, A.9>Write the code for RISC-V. How many instructions are required dynamically? How many memory-data references will be executed? What is the code size in bytes?

b. [20]<A.2>Write the code for x86. How many instructions are required dynamically? How many memory-data references will be executed? What is the code size in bytes?

c. [20]<A.2>Write the code for a stack machine. Assume all operations occur on top of the stack. Push and pop are the only instructions that access memory; all others remove their operands from the stack and replace them with the result. The implementation uses a hardwired stack for only the top two stack entries, which keeps the processor circuit very small and low in cost. Additional stack positions are kept in memory locations, and accesses to these stack positions require memory references. How many instructions are required dynamically? How many memory-data references will be executed?

d. [25]<A.2, A.9>Instead of the code fragment above, write a routine for computing a matrix multiplication for dense, single precision matrices, also known as SGEMM. For input matrices of size 100×100, how many instructions are required dynamically? How many memory-data references will be executed?

e. [10]<A.2, A.9>As the matrix size increases, how does this affect the number of instructions executed dynamically or the number of memory-data references?

A.8 [25/25]<A.2, A.8, A.9>Consider the following fragment of C code:

```
for(p=0; p<8; p++) {
      Y[p]=(9798*R[p]+19235*G[p]+3736*B[p])/32768;
      U[p]=(-4784*R[p]-9437*G[p]+4221*B[p])/32768+128;
      V[p]=(20218*R[p]-16941*G[p]-3277*B[p])/32768+128;
}
```

Assume that R, G, B, Y, U, and V are arrays of 64-bit integers. Assume that all data values and their addresses are kept in memory (at addresses 1000,

2000, 3000, 4000, 5000, and 6000 for R, G, B, Y, U, and V, respectively) except when they are operated on. Assume that values in registers are lost between iterations of the loop. Assume all addresses and words are 64 bits.

a. [25]<A.2, A.9>Write the code for RISC-V. How many instructions are required dynamically? How many memory-data references will be executed? What is the code size in bytes?

b. [25]<A.2>Write the code for x86. How many instructions are required dynamically? How many memory-data references will be executed? What is the code size in bytes? Compare your results to the multimedia instructions (MMX) and vector implementations discussed in the A.8.

A.9 [10/10/10/10]<A.2, A.7>For the following, we consider instruction encoding for instruction set architectures.

a. [10]<A.2, A.7>Consider the case of a processor with an instruction length of 14 bits and with 64 general-purpose registers so the size of the address fields is 6 bits. Is it possible to have instruction encodings for the following?

- 3 two-address instructions
- 63 one-address instructions
- 45 zero-address instructions

b. [10]<A.2, A.7>Assuming the same instruction length and address field sizes as above, determine if it is possible to have

- 3 two-address instructions
- 65 one-address instructions
- 35 zero-address instructions

Explain your answer.

c. [10]<A.2, A.7>Assume the same instruction length and address field sizes as above. Further assume there are already 3 two-address and 24 zero-address instructions. What is the maximum number of one-address instructions that can be encoded for this processor?

d. [10]<A.2, A.7>Assume the same instruction length and address field sizes as above. Further assume there are already 3 two-address and 65 zero-address instructions. What is the maximum number of one-address instructions that can be encoded for this processor?

A.10 [10/15]<A.2>For the following assume that integer values A, B, C, D, E, and F reside in memory. Also assume that instruction operation codes are represented in 8 bits, memory addresses are 64 bits, and register addresses are 6 bits.

a. [10]<A.2>For each instruction set architecture shown in Figure A.2, how many addresses, or names, appear in each instruction for the code to compute C = A + B, and what is the total code size?

b. [15] < A.2 > Some of the instruction set architectures in Figure A.2 destroy operands in the course of computation. This loss of data values from processor internal storage has performance consequences. For each architecture in Figure A.2, write the code sequence to compute:

```
C=A+B
D=A - E
F=C+D
```

In your code, mark each operand that is destroyed during execution and mark each "overhead" instruction that is included just to overcome this loss of data from processor internal storage. What is the total code size, the number of bytes of instructions and data moved to or from memory, the number of overhead instructions, and the number of overhead data bytes for each of your code sequences?

A.11 [15] < A.2, A.7, A.9 > The design of RISC-V provides for 32 general-purpose registers and 32 floating-point registers. If registers are good, are more registers better? List and discuss as many trade-offs as you can that should be considered by instruction set architecture designers examining whether to, and how much to, increase the number of RISC-V registers.

A.12 [5] < A.3 > Consider a C struct that includes the following members:

```
struct foo {
    char a;
    bool b;
    int c;
    double d;
    short e;
    float f;
    double g;
    char *cptr;
    float *fptr;
    int x;
};
```

Note that for C, the compiler must keep the elements of the struct in the same order as given in the struct definition. For a 32-bit machine, what is the size of the foo struct? What is the minimum size required for this struct, assuming you may arrange the order of the struct members as you wish? What about for a 64-bit machine?

A.13 [30] < A.7 > 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 techniques that instrument the object code module by inserting counters in software

or using built-in hardware counters. Pick a processor and tools to instrument user programs. (The open source RISC-V architecture supports a collection of tools. Tools such as the Performance API (PAPI) work with x86 processors.) Use the processor and tools to measure the instruction set mix for one of the SPEC CPU2017 benchmarks. Compare the results to those shown in this chapter.

A.14 [30] < A.8 > Newer processors such as Intel's i7 Kaby Lake include support for AVX2 vector/multimedia instructions. Write a dense matrix multiply function using single-precision values and compile it with different compilers and optimization flags. Linear algebra codes using Basic Linear Algebra Subroutine (BLAS) routines such as SGEMM include optimized versions of dense matrix multiply. Compare the code size and performance of your code to that of BLAS SGEMM. Explore what happens when using double-precision values and DGEMM.

A.15 [30] < A.8 > For the SGEMM code developed above for the i7 processor, include the use of AVX2 intrinsics to improve the performance. In particular, try to vectorize your code to better utilize the AVX hardware. Compare the code size and performance to the original code. Compare your results to Intel's Math Kernel Library (MKL) implementation for SGEMM.

A.16 [30] < A.7, A.9 > The RISC-V processor is open source and boasts an impressive collection of implementations, simulators, compilers, and other tools. See riscv.org for an overview of tools, including spike, a simulator for RISC-V processors. Use spike or another simulator to measure the instruction set mix for some SPEC CPU2017 benchmark programs.

A.17 [35/35/35/35] < A.2−A.8 > gcc targets most modern instruction set architectures (see www.gnu.org/software/gcc/). Create a version of gcc for several architectures that you have access to, such as x86, RISC-V, PowerPC, and ARM.

a. [35] < A.2−A.8 > Compile a subset of SPEC CPU2017 integer benchmarks and create a table of code sizes. Which architecture is best for each program?

b. [35] < A.2−A.8 > Compile a subset of SPEC CPU2017 floating-point benchmarks and create a table of code sizes. Which architecture is best for each program?

c. [35] < A.2−A.8 > Compile a subset of EEMBC AutoBench benchmarks (see www.eembc.org/home.php) and create a table of code sizes. Which architecture is best for each program?

d. [35] < A.2−A.8 > Compile a subset of EEMBC FPBench floating-point benchmarks and create a table of code sizes. Which architecture is best for each program?

A.18 [40] < A.2−A.8 > Power efficiency has become very important for modern processors, particularly for embedded systems. Create a version of gcc for two architectures that you have access to, such as x86, RISC-V, PowerPC, Atom, and ARM. (Note that the different versions of RISC-V can also be explored and compared.) Compile a subset of EEMBC benchmarks while using EnergyBench to measure

energy usage during execution. Compare code size, performance, and energy usage for the processors. Which is best for each program?

A.19 [20/15/15/20] Your task is to compare the memory efficiency of four different styles of instruction set architectures. The architecture styles are:

- *Accumulator*—All operations occur between a single register and a memory location.
- *Memory-memory*—All instruction addresses reference only memory locations.
- *Stack*—All operations occur on top of the stack. Push and pop are the only instructions that access memory; all others remove their operands from the stack and replace them with the result. The implementation uses a hardwired stack for only the top two stack entries, which keeps the processor circuit very small and low in cost. Additional stack positions are kept in memory locations, and accesses to these stack positions require memory references.
- *Load-store*—All operations occur in registers, and register-to-register instructions have three register names per instruction.

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

- All instructions are an integral number of bytes in length.
- The opcode is always one byte (8 bits).
- Memory accesses use direct, or absolute, addressing.
- The variables `A, B, C, and D` are initially in memory.

a. [20] <A.2, A.3> Invent your own assembly language mnemonics (Figure A.2 provides a useful sample to generalize), and for each architecture write the best equivalent assembly language code for this high-level language code sequence:

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

b. [15] <A.3> Label each instance in your assembly codes for part (a) where a value is loaded from memory after having been loaded once. Also label each instance in your code where the result of one instruction is passed to another instruction as an operand, and further classify these events as involving storage within the processor or storage in memory.

c. [15] <A.7> Assume that the given code sequence is from a small, embedded computer application that uses a 16-bit memory address and data operands. If a load-store architecture is used, assume it has 16 general-purpose registers. For each architecture answer the following questions: How many instruction bytes are fetched? How many bytes of data are transferred from/to memory? Which

architecture is most efficient as measured by total memory traffic (code + data)?

d. [20] <A.7> Now assume a processor with 64-bit memory addresses and data operands. For each architecture answer the questions of part (c). How have the relative merits of the architectures changed for the chosen metrics?

A.20 [30] <A.2, A.3> Use the four different instruction set architecture styles from above, but assume that the memory operations supported include register indirect as well as direct addressing. Invent your own assembly language mnemonics (Figure A.2 provides a useful sample to generalize), and for each architecture, write the best equivalent assembly language code for this fragment of C code:

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

Assume that `A` and `B` are arrays of 64-bit integers, and `C`, `D`, and `i` are 64-bit integers.

A.21 [20/20] <A.3, A.6, A.9> The size of displacement values needed for the displacement addressing mode or for PC-relative addressing can be extracted from compiled applications. Use a disassembler with one or more of the SPEC CPU2017 or EEMBC benchmarks compiled for the RISC-V processor.

a. [20] <A.3, A.9> For each instruction using displacement addressing, record the displacement value used. Create a histogram of displacement values. Compare the results to those shown in this appendix in Figure A.8.

b. [20] <A.6, A.9> For each branch instruction using PC-relative addressing, record the offset value used. Create a histogram of offset values. Compare the results to those shown in this chapter in Figure A.15.

A.22 [15/15/10/10/10/10] <A.3> The value represented by the hexadecimal number 5249 5343 5643 5055 is to be stored in an aligned 64-bit double word.

a. [15] <A.3> Using the physical arrangement of the first row in Figure A.5, write the value to be stored using Big Endian byte order. Next, interpret each byte as an ASCII character and below each byte write the corresponding character, forming the character string as it would be stored in Big Endian order.

b. [15] <A.3> Using the same physical arrangement as in part (a), write the value to be stored using Little Endian byte order, and below each byte write the corresponding ASCII character.

c. [10] <A.3> What are the hexadecimal values of all misaligned 2-byte words that can be read from the given 64-bit double word when stored in Big Endian byte order?

d. [10] < A.3 > What are the hexadecimal values of all misaligned 2-byte words that can be read from the given 64-bit double word when stored in Big Endian byte order?

e. [10] < A.3 > What are the hexadecimal values of all misaligned 2-byte words that can be read from the given 64-bit double word when stored in Little Endian byte order?

f. [10] < A.3 > What are the hexadecimal values of all misaligned 4-byte words that can be read from the given 64-bit double word when stored in Little Endian byte order?

A.23 [25,25] < A.3, A.9 > The relative frequency of different addressing modes impacts the choices of addressing modes support for an instruction set architecture. Figure A.7 illustrates the relative frequency of addressing modes for three applications on the VAX.

a. [25] < A.3 > Compile one or more programs from the SPEC CPU2017 or EEMBC benchmark suites to target the x86 architecture. Using a disassembler, inspect the instructions and the relative frequency of various addressing modes. Create a histogram to illustrate the relative frequency of the addressing modes. How do your results compare to Figure A.7?

b. [25] < A.3, A.9 > Compile one or more programs from the SPEC CPU2017 or EEMBC benchmark suites to target the RISC-V architecture. Using a disassembler, inspect the instructions and the relative frequency of various addressing modes. Create a histogram to illustrate the relative frequency of the addressing modes. How do your results compare to Figure A.7?

A.24 [Discussion] < A.2−A.12 > Consider typical applications for desktop, server, cloud, and embedded computing. How would instruction set architecture be impacted for machines targeting each of these markets?