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 information

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.

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 doublewords 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.

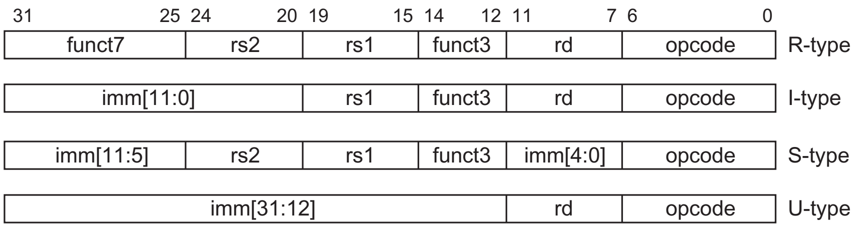

Figure A.23 The RISC-V instruction layout. There are two variations on these formata, called the SB and UJ formats; they deal with a slightly different treatment for immediate fields.

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

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 $\leftarrow$ whenever the length of the datum being transferred might not be clear. Thus, $\leftarrow_n$ means transfer an *n*-bit quantity. We use $x, y \leftarrow z$ to indicate that z should be transferred to x and y.
- A subscript is used to indicate selection of a bit from a field. Bits are labeled from the most-significant bit starting at 0. The subscript may be a single digit (e.g., $\texttt{Regs[x4]}_0$ yields the sign bit of x4) or a subrange (e.g., $\texttt{Regs [x3]}_{56..63}$ 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	Regs[x1]←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 by adding the immediate

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.

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