1 Why RISC-V?

Leonardo da Vinci (1452-1519) was a Renaissance architect, engineer, sculptor, and painter of the Mona Lisa.

Simplicity is the ultimate sophistication.

—Leonardo da Vinci

1.1 Introduction

The goal for RISC-V ("RISC five") is to become a universal *instruction set architecture* (*ISA*):

- It should suit all sizes of processors, from the tiniest embedded controller to the fastest high-performance computer.
- It should work well with a wide variety of popular software stacks and programming languages.
- It should accommodate all implementation technologies: Field-Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), full-custom chips, and even future device technologies.
- It should be efficient for all microarchitecture styles: microcoded or hardwired control; in-order, decoupled, or out-of-order pipelines; single or superscalar instruction issue; and so on.
- It should support extensive specialization to act as a base for customized accelerators, which rise in importance as Moore's Law fades.
- It should be stable, in that the base ISA should not change. More importantly, the ISA cannot be discontinued, as has happened in the past to proprietary ISAs such as the AMD Am29000, the Digital Alpha, the Digital VAX, the Hewlett Packard PA-RISC, the Intel i860, the Intel i960, the Motorola 88000, and the Zilog Z8000.

We add sidebars in the margins to offer hopefully interesting commentary. For example, RISC-V was originally developed for internal use in UC Berkeley research and courses. It became open because outsiders started using it on their own. The RISC-V architects learned about the external interest when they started receiving complaints about ISA changes in their coursework, which was on the web. Only after the architects understood the need did they try to make it an open ISA standard.

RISC-V is unusual not only because it is a recent ISA—born this decade when most alternatives date from the 1970s or 1980s—but also because it is an *open* ISA. Unlike practically all prior architectures, its future is free from the fate or the whims of any single corporation, which has doomed many ISAs in the past. It belongs instead to an open, non-profit foundation. The goal of the RISC-V Foundation is to maintain the stability of RISC-V, evolve it slowly and carefully, solely for technical reasons, and try to make it as popular for hardware as Linux is for operating systems. As a sign of its vitality, Figure 1.1 lists the largest corporate members of the RISC-V Foundation.

>$50B		>$5B, <$50B		>$0.5B, <$5B	
Google	USA	BAE Systems	UK	AMD	USA
Huawei	China	MediaTek	Taiwan	Andes Technology	China
IBM	USA	Micron Tech.	USA	C-SKY Microsystems	China
Microsoft	USA	Nvidia	USA	Integrated Device Tech.	USA
Samsung	Korea	NXP Semi.	Netherlands	Mellanox Technology	Israel
		Qualcomm	USA	Microsemi Corp.	USA
		Western Digital	USA		

Figure 1.1: The corporate members of the RISC-V Foundation as of the Sixth RISC-V Workshop in May 2017 ranked by annual sales. The left column companies all exceed $US 50B in annual sales, the middle column companies sell less than $US 50B but more than $US 5B, and the sales of those in the right column are less than $US 5B but more than $US 0.5B. The foundation includes another 25 smaller companies, 5 startup companies (Antmicro Ltd, Blockstream, Esperanto Technologies, Greenwaves Technologies, and SiFive), 4 nonprofit organizations (CSEM, Draper Laboratory, ICT, and lowRISC), and 6 universities (ETH Zurich, IIT Madras, National University of Defense Technology, Princeton, and UC Berkeley). Most of the 60 organizations have their headquarters outside the US. To learn more, see www.riscv.org.

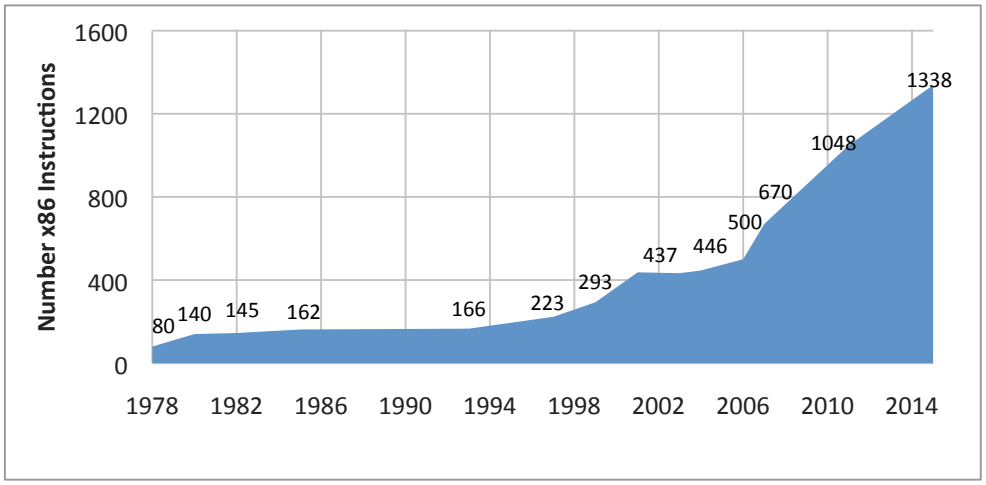


Figure 1.2: Growth of x86 instruction set over its lifetime. x86 started with 80 instructions in 1978. It grew 16X to 1338 instructions by 2015, and it's still growing. Amazingly, this graph is conservative. An Intel blog puts the count at 3600 instructions in 2015 [Rodgers and Uhlig 2017], which would raise the x86 rate to one new instruction *every four days* between 1978 and 2015. We count assembly language instructions, and they presumably count machine language instructions. As Chapter 8 explains, a large part of the growth is because the x86 ISA relies on SIMD instructions for data level parallelism.

```
The AL register is the default source and destination.
If the low 4-bits of AL register are > 9,
   or the auxiliary carry flag AF = 1,
 Then
    Add 6 to low 4-bits of AL and discard overflow
    Increment the high byte of AL
    Carry flag CF = 1
    Auxiliary carry flag AF = 1
 Else
    CF = AF = 0
Upper 4-bits of AL = 0
```

Figure 1.3: Description of the x86-32 *ASCII Adjust after Addition* (`aaa`) instruction. It performs computer arithmetic in Binary Coded Decimal (BCD), which has fallen into the dustbin of information technology history. The x86 also has three related instructions for subtraction (`aas`), multiplication (`aam`), and division (`aad`). As each is a one-byte instruction, they collectively occupy 1.6% $(4/256)$ of the precious opcode space.

1.2 Modular vs. Incremental ISAs

> *Intel was betting its future on a high-end microprocessor, but that was still years away. To counter Zilog, Intel developed a stop-gap processor and called it the 8086. It was intended to be short-lived and not have any successors, but that's not how things turned out. The high-end processor ended up being late to market, and when it did come out, it was too slow. So the 8086 architecture lived on—it evolved into a 32-bit processor and eventually into a 64-bit one. The names kept changing (80186, 80286, i386, i486, Pentium), but the underlying instruction set remained intact.*
>
> —Stephen P. Morse, architect of the 8086 [Morse 2017]

The conventional approach to computer architecture is *incremental* ISAs, where new processors must implement not only new ISA extensions but also all extensions of the past. The purpose is to maintain *backwards binary-compatibility* so that binary versions of decades-old programs can still run correctly on the latest processor. This requirement, when combined with the marketing appeal of announcing new instructions with a new generation of processors, has led to ISAs that grow substantially in size with age. For example, Figure 1.2 shows the growth in the number of instructions for a dominant ISA today: the 80x86. It dates back to 1978, yet it has added about *three instructions per month* over its long lifetime.

This convention means that every implementation of the x86-32 (the name we use for the 32-bit address version of x86) must implement the mistakes of past extensions, even when they no longer make sense. For example, Figure 1.3 describes the ASCII Adjust after Addition (`aaa`) instruction of the x86, which has long outlived its usefulness.

As an analogy, suppose a restaurant serves only a fixed-price meal, which starts out as a small dinner of just a hamburger and a milkshake. Over time, it adds fries, and then an ice cream sundae, followed by salad, pie, wine, vegetarian pasta, steak, beer, ad infinitum until it becomes a gigantic banquet. It may make little sense in total, but diners can find whatever they've ever eaten in a past meal at that restaurant. The bad news is that diners must pay the rising cost of the expanding banquet for each dinner.

Beyond being recent and open, RISC-V is unusual since, unlike almost all prior ISAs, it is *modular*. At the core is a base ISA, called *RV32I*, which runs a full software stack. RV32I is

frozen and will never change, which gives compiler writers, operating system developers, and assembly language programmers a stable target. The modularity comes from optional standard extensions that hardware can include or not depending on the needs of the application. This modularity enables very small and low energy implementations of RISC-V, which can be critical for embedded applications. By informing the RISC-V compiler what extensions are included, it can generate the best code for that hardware. The convention is to append the extension letters to the name to indicate which are included. For example, RV32IMFD adds the multiply (RV32M), single-precision floating point (RV32F), and double-precision floating point extensions (RV32D) to the mandatory base instructions (RV32I).

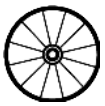
If software uses an omitted RISC-V instruction from an optional extension, the hardware traps and executes the desired function in software as part of a standard library.

Returning to our analogy, RISC-V offers a menu instead of a buffet; the chef need cook only what the customers want—not a feast for every meal—and the customers pay only for what they order. RISC-V has no need to add instructions simply for the marketing sizzle. The RISC-V Foundation decides when to add a new option to the menu, and they will do so only for solid technical reasons after an extended open discussion by a committee of hardware and software experts. Even when new choices appear on the menu, they remain optional and not a new requirement for all future implementations, like incremental ISAs.

1.3 ISA Design 101

Before introducing the RISC-V ISA, it will be helpful to understand the underlying principles and trade-offs that a computer architect must make while designing an ISA. Below is a list of the seven measures, along with icons we'll put in page margins to highlight instances when RISC-V addresses them in the following chapters. (The back cover of the print book has a legend for the icons.)

- cost (US dollar coin icon)
- simplicity (wheel)
- performance (speedometer)
- isolation of architecture from implementation (detached halves of a circle)
- room for growth (accordion)
- program size (opposing arrows compressing line)
- ease of programming / compiling / linking (children's blocks for "as easy as ABC").




To illustrate what we mean, in this section we'll show some choices from older ISAs that look unwise in retrospect and where RISC-V often made much better decisions.

Cost. Processors are implemented as integrated circuits, commonly called *chips* or *dies*. They are called dies because they start life as a piece of a single round wafer, which is *diced* into many individual pieces. Figure 1.4 shows a wafer of RISC-V processors. The cost is very sensitive to the area of the die:

$$cost \approx f(die\ area^2)$$

Obviously, the smaller the die, the more dies per wafer, and most of the cost of the die is the processed wafer itself. Less obvious is that the smaller the die, the higher the *yield*, the

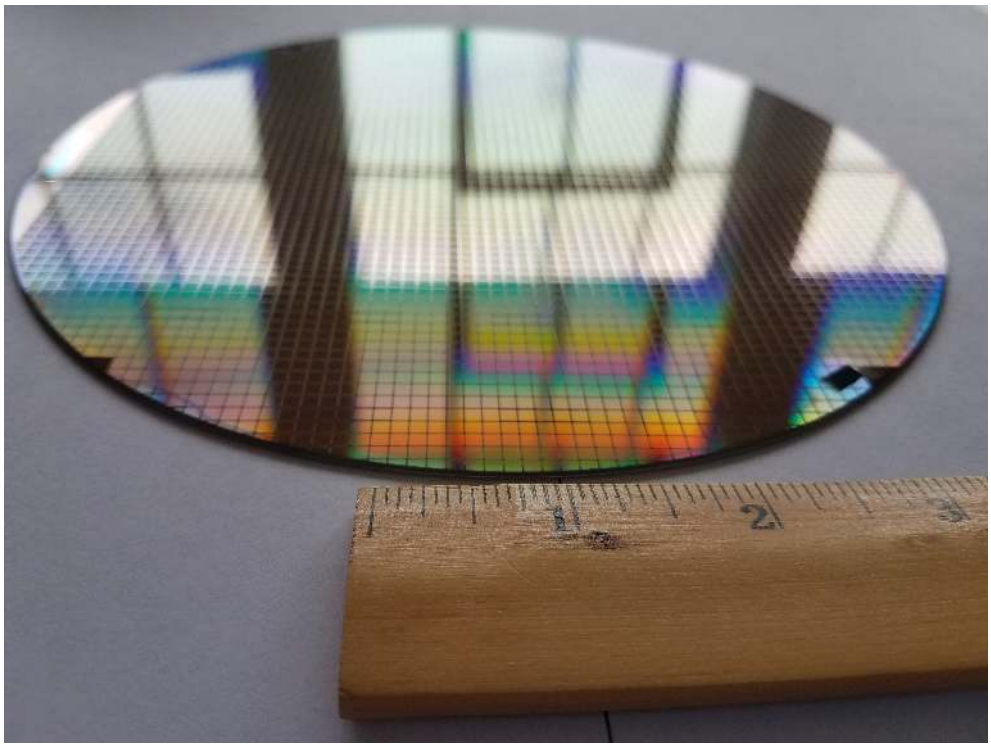


Figure 1.4: An 8-inch diameter wafer of RISC-V dies designed by SiFive. It has two types of RISC-V dies using an older, larger processing line. An FE310 die is 2.65 mm×2.72 mm and a SiFive test die that is 2.89 mm×2.72 mm. The wafer contains 1846 of the former and 1866 of the latter, totaling 3712 chips.

fraction of manufactured dies that work. The reason is that the silicon manufacturing will result in small flaws scattered about the wafer, so the smaller the die, the lower the fraction that will be flawed.

An architect wants to keep the ISA simple to shrink the size of processors that implement it. As we shall see in the following chapters, the RISC-V ISA is much simpler ISA than the ARM-32 ISA. As a concrete example of the impact of simplicity, let's compare a RISC-V Rocket processor to an ARM-32 Cortex-A5 processor in the same technology (TSMC40GPLUS) using the same-sized caches (16 KiB). The RISC-V die is 0.27 mm^2 versus 0.53 mm^2 for ARM-32. Around twice the area, the ARM-32 Cortex-A5 die costs approximately 4X (2^2) as much as RISC-V Rocket die. Even a 10% smaller die reduces cost by a factor of 1.2 (1.1^2).

Simplicity. Given the cost sensitivity to complexity, architects want a simple ISA to reduce die area. Simplicity also reduces chip design time and verification time, which can be much of the cost of development of the chip. These costs must be added to the cost of the chip, with this overhead dependent on the number of chips shipped. Simplicity also reduces the cost of documentation and the difficulty of getting customers to understand how to use the ISA.

High-end processors can gain performance by combining simple instructions together without burdening all lower-end implementations with a larger, more complicated ISA. This technique is called *macro-fusion*, as it fuses "macro" instructions together.

Below is a glaring example of ISA complexity from ARM-32:

```
ldmiaeq SP!, {R4-R7, PC}
```

The instruction stands for LoaD Multiple, Increment-Address, on EQual. It performs 5 data loads and writes to 6 registers but executes only if the EQ condition code is set. Moreover, it writes a result to the PC, so it is also performing a conditional branch. Quite a handful!

A simple processor can be helpful for embedded applications since it is easier to predict execution time. Assembly-language programmers of microcontrollers often want to maintain exact timing, so they rely on code taking a predictable number of clock cycles that they can count by hand.

Ironically, simple instructions are much more likely to be used than complex ones. For example, x86-32 includes an `enter` instruction, which was intended to be the first instruction executed on entering a procedure to create a stack frame for it (see Chapter 3). Most compilers instead use only these two simple x86-32 instructions:

```
push ebp      # Push the frame pointer onto the stack
mov  ebp, esp # Copy the stack pointer to the frame pointer
```

Performance. Except for the tiny chips for embedded applications, architects are typically concerned about performance as well as cost. Performance can be factored into three terms:

$$\frac{instructions}{program} \times \frac{average\ clock\ cycles}{instruction} \times \frac{time}{clock\ cycle} = \frac{time}{program}$$

The last factor is the inverse of the clock rate, so a 1 GHz clock rate means the time per clock cycle is 1 ns ($1/10^9$).

Even if a simple ISA might execute more instructions per program than a complex ISA, it can more than make up for that by having a faster clock cycle or average fewer clock cycles per instruction (CPI).

For example, for the CoreMark benchmark [Gal-On and Levy 2012] (100,000 iterations), the performance on the ARM-32 Cortex-A9 is

$$\frac{32.27\ B\ instructions}{program} \times \frac{0.79\ clock\ cycles}{instruction} \times \frac{0.71\ ns}{clock\ cycle} = \frac{18.15\ secs}{program}$$

The average number of clock cycles can be less than 1 because the A9 and BOOM [Celio et al. 2015] are so-called *superscalar* processors, which execute more than one instruction per clock cycle.

For the BOOM implementation of RISC-V, the equation is

$$\frac{29.51\ B\ instructions}{program} \times \frac{0.72\ clock\ cycles}{instruction} \times \frac{0.67\ ns}{clock\ cycle} = \frac{14.26\ secs}{program}$$

The ARM processor didn't execute fewer instructions than RISC-V in this case. As we shall see, the simple instructions are also the most popular instructions, so ISA simplicity can win in all metrics. For this program, the RISC-V processor gains nearly 10% in each of the three factors, which results in a performance advantage of almost 30%. If a simpler ISA also results in a smaller chip, its cost-performance will be excellent.

Isolation of Architecture from Implementation. The original distinction between *architecture* and *implementation*, which goes back to the 1960s, is that architecture is what a machine language programmer needs to know to write a correct program, but not the performance of that program. The temptation for an architect is to include instructions in an ISA that help performance or cost of one implementation at a particular time, but burden different or future implementations.

Pipelined processors today anticipate branch outcomes using hardware predictors, which can exceed 90% accuracy and work with any pipeline length. They only need a mechanism to flush and restart the pipeline when they mispredict.

For the MIPS-32 ISA, the regrettable example was the *delayed branch*. Conditional branches cause problems in pipelined execution because the processor wants to have the next instruction to execute already in the pipeline, but it can't decide whether it wants the next sequential one (if the branch isn't taken) or the one at the branch target address (if it is taken). For their first microprocessor with a 5-stage pipeline, this indecision could have caused a one clock-cycle stall of the pipeline. MIPS-32 solved this problem by redefining branch to occur in the instruction *after* the next one. Thus, the following instruction is *always* executed. The job of the programmer or compiler writer was to put something useful into the *delay slot*.

Alas, this "solution" didn't help later MIPS-32 processors with many more pipeline stages (hence many more instructions fetched before the branch outcome is computed), but it made life harder for MIPS-32 programmers, compiler writers, and processor designers ever after, since incremental ISAs demand backwards compatibility (see Section 1.2). In addition, it makes the MIPS-32 code much harder to understand (see Figure 2.10 on page 29).

While architects shouldn't put features that *help* just one implementation at a point in time, they also shouldn't put in features that *hinder* some implementations. For example, ARM-32 and some other ISAs have a Load Multiple instruction, as mentioned on the previous page. These instructions can improve performance of single-instruction issue pipelined designs, but hurt multiple-instruction issue pipelines. The reason is that the straightforward implementation precludes scheduling the individual loads of a Load Multiple in parallel with other instructions, reducing instruction throughput of such processors.

Room for Growth. With ending of Moore's Law, the only path forward for major improvements in cost-performance is to add custom instructions for specific domains, such as deep learning, augmented reality, combinatorial optimization, graphics, and so. That means it's important today for an ISA to reserve opcode space for future enhancements.

In the 1970s and 1980s, when Moore's Law was in full force, there was little thought of saving opcode space for future accelerators. Architects instead valued larger address and immediate fields to reduce the number of instructions executed per program, the first factor in the performance equation on the prior page.

The ARM-32 instruction `ldmiaeq` mentioned above is even more complicated, since when it branches it can also change instruction set modes between ARM-32 and Thumb/Thumb-2.

An example of the impact of paucity of opcode space was when the architects of ARM-32 later tried to reduce code size by adding 16-bit length instructions to the formerly uniform 32-bit length ISA. There was simply no room left. Thus, the only solution was to create a new ISA first with 16-bit instructions (Thumb) and later a new ISA with both 16-bit and 32-bit instructions (Thumb-2) using a mode bit to switch between ARM ISAs. To change modes, the programmer or compiler branches to a byte address with a 1 in the least-significant bit, which worked because 16-bit and 32-bit instructions should have 0 in that bit.

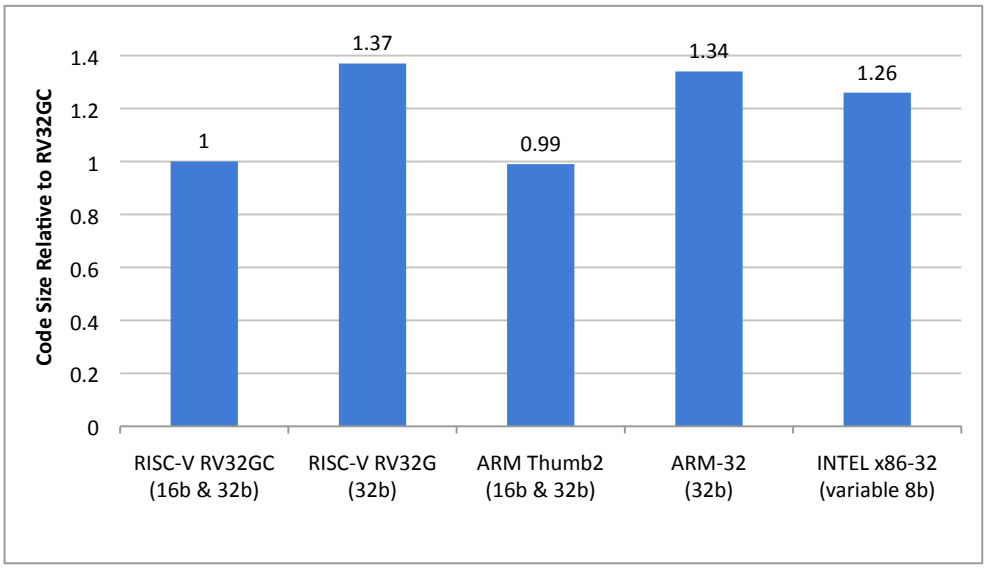
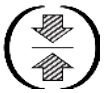


Figure 1.5: Relative program sizes for RV32G, ARM-32, x86-32, RV32C, and Thumb-2. The last two ISAs are aimed at small code size. The programs were the SPEC CPU2006 benchmarks using the GCC compilers. The small size advantage of Thumb-2 over RV32C is due to the code size savings of Load and Store Multiple on procedure entry. RV32C excludes them to maintain the one-to-one mapping to instructions of RV32G, which omits Load and Store Multiple to reduce implementation complexity for high-end processors (see below). Chapter 7 explains RV32C. RV32G indicates a popular combination of RISC-V extensions (RV32M, RV32F, RV32D, and RV32A), properly called RV32IMAFD. [Waterman 2016]

Program Size. The smaller the program, the smaller the area on a chip needed for the program memory, which can be a significant cost for embedded devices. Indeed, that issue inspired ARM architects to retroactively add shorter instructions in the Thumb and Thumb-2 ISAs. Smaller programs also lead to fewer misses in instruction caches, which saves power since off-chip DRAM accesses use much more energy than on-chip SRAM accesses, and improves performance as well. Small code size can be one of the goals of ISA architects.

The x86-32 ISA has instructions as short as 1 byte and as long as 15 bytes. One would expect that the byte-variable length instructions of the x86 should certainly lead to smaller programs than ISAs limited to 32-bit length instructions, like ARM-32 and RISC-V. Logically, 8-bit variable length instructions should also be smaller than ISAs that offer only 16-bit and 32-bit instructions, like Thumb-2 and RISC-V using the RV32C extension (see Chapter 7). Figure 1.5 shows that, while ARM-32 and RISC-V code is 6% to 9% larger than code for x86-32 when all instructions are 32 bits long, surprisingly x86-32 is 26% *larger* than the compressed versions (RV32C and Thumb-2) that offer both 16-bit and 32-bit instructions.

While a new ISA using 8-bit variable instructions would likely lead to smaller code than RV32C and Thumb-2, the architects of the first x86 in the 1970s had different concerns. Moreover, given the requirement of backwards binary-compatibility of an incremental ISA (Section 1.2), the hundreds of new x86-32 instructions are longer than one might expect, since they bear the burden of a one- or two-byte prefix to squeeze them into the limited free opcode space of the original x86.



One example 15-byte x86-32 instruction is `lock add dword ptr ds:[esi+ecx*4 +0x12345678], 0xefcdab89`. It assembles into (in hexadecimal): 67 66 f0 3e 81 84 8e 78 56 34 12 89 ab cd ef. The last 8 bytes are 2 addresses and the first 7 bytes specify atomic memory operation, the add operation, 32-bit data, the data segment register, the 2 address registers, and scaled indexed addressing mode. An example 1-byte instruction is `inc eax` that assembles into 40.

Ease of programming, compiling, and linking. Since data in a register is so much faster to access than data in memory, it is critical for compilers to do a good job at register allocation. That task is much easier when there are many registers rather than fewer. In that light, ARM-32 has 16 registers and x86-32 has only 8. Most modern ISAs, including RISC-V, have a relatively generous 32 integer registers. More registers surely make life easier for compilers and assembly language programmers.

Another issue for compilers and assembly language programmers is figuring out the speed of a code sequence. As we shall see, RISC-V instructions are typically at most one clock cycle per instruction (ignoring cache misses), while as we saw earlier both ARM-32 and x86-32 have instructions that take many clock cycles even when everything fits in the cache. Moreover, unlike ARM-32 and RISC-V, x86-32 arithmetic instructions can have operands in memory instead of requiring all operands to be in registers. Complex instructions and operands in memory make it difficult for processor designers to deliver performance predictability.

It's useful for an ISA to support *position independent code* (*PIC*), because it supports dynamic linking (see Section 3.5), since shared library code can reside at different addresses in different programs. PC-relative branches and data addressing are a boon to PIC. While nearly all ISAs provide PC-relative branches, x86-32 and MIPS-32 omit PC-relative data addressing.

■ ***Elaboration: ARM-32, MIPS-32, and x86-32***

Elaborations are optional sections that readers can delve into if they are interested in a topic, but you don't need to read them to understand the rest of the book. For example, our ISA names aren't the official ones. The 32-bit-address ARM ISA has many versions, with the first in 1986 and the latest called ARMv7 in 2005. ARM-32 generally refers to the ARMv7 ISA. MIPS also had many 32-bit versions, but we're referring to the original, called MIPS I. ("MIPS32" is a different, later ISA than what we call MIPS-32.) Intel's first 16-bit address architecture was the 8086 in 1978, which the 80386 ISA expanded to 32-bit addresses in 1985. Our x86-32 notation generally refers to the IA-32, the 32-bit-address version of its x86 ISA. Given the myriad variants of these ISAs, we find our nonstandard terminology least confusing.

1.4 An Overview of this Book

This book assumes you have seen other instruction sets before RISC-V. If not, look at our related introductory architecture book based on RISC-V [Patterson and Hennessy 2017].

Chapter 2 introduces RV32I, the frozen base integer instructions that are the heart of RISC-V. Chapter 3 explains the remaining RISC-V assembly language beyond that introduced in Chapter 2, including calling conventions and some clever tricks for linking. Assembly language includes all of the proper RISC-V instructions plus some useful instructions that are outside RISC-V. These *pseudoinstructions*, which are clever variations of real instructions, make it easier to write assembly language programs without having to complicate the ISA.

The next three chapters explain the standard RISC-V extensions that, when added to RV32I, we collectively call RV32G (G is for general):

- Chapter 4: Multiply and Divide (RV32M)
- Chapter 5: Floating Point (RV32F and RV32D)
- Chapter 6: Atomic (RV32A)

The RISC-V "reference card" on pages 3 and 4 is a handy summary of *all* RISC-V instructions in this book: RV32G, RV64G, and RV32/64V.

The reference card is also called the *green card* because of the shade of the background color of the one-page cardboard summary of ISAs from the 1960s. We kept the background white for legibility instead of green for historical accuracy.

Chapter 7 describes the optional compressed extension RV32C, an excellent example of the elegance of RISC-V. By restricting the 16-bit instructions to be short versions of existing 32-bit RV32G instructions, they are almost free. The assembler can pick the instruction size, allowing the assembly language programmer and the compiler to be oblivious to RV32C. The hardware decoder to translate 16-bit RV32C instructions into 32-bit RV32G instructions needs just 400 gates, which is a few percent of even the simplest implementation of RISC-V.

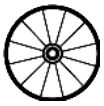
Chapter 8 introduces RV32V, the vector extension. Vector instructions are another example of ISA elegance as compared to the numerous, brute-force *Single Instruction Multiple Data* (*SIMD*) instructions of ARM-32, MIPS-32, and x86-32. Indeed, hundreds of the instructions added to x86-32 in Figure 1.2 were SIMD, and hundreds more are coming. RV32V is even simpler than most vector ISAs, as it associates the data type and length with the vector registers instead of embedding them in the opcodes. RV32V may be the most compelling reason for switching from a conventional SIMD-based ISA to RISC-V.

Chapter 9 shows the 64-bit address version of RISC-V, RV64G. As the chapter explains, the RISC-V architects needed only to widen the registers and add a few `word`, `doubleword`, or `long` versions of RV32G instructions to extend the address from 32 to 64 bits.

Chapter 10 explains the system instructions, showing how RISC-V handles paging and the Machine, User, and Supervisor privilege modes.

The last chapter gives a quick description of the remaining extensions that are currently under consideration by the RISC-V Foundation.

Next comes the largest section of the book, Appendix A, an instruction set summary in alphabetical order. It defines the full RISC-V ISA with all extensions mentioned above and all pseudoinstructions in about 50 pages, a testimony to the simplicity of RISC-V.



We end the book with an index.

1.5 Concluding Remarks

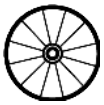
> *It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations ... The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.*
>
> —[von Neumann et al. 1947, 1947]

A previous version of John von Neumann's well-written report was so influential that this style of computer is commonly called a *von Neumann architecture*, although this report was based on the work of others. It was written three years before the first stored program computer was operational!

RISC-V is a recent, clean-slate, minimalist, and open ISA informed by mistakes of past ISAs. The goal of the RISC-V architects is for it to be effective for all computing devices, from the smallest to the fastest. Following von Neumann's 70-year-old advice, this ISA emphasizes simplicity to keep costs low while having plenty of registers and transparent instruction speed to help compilers and assembly language programmers map actually important problems to appropriate, quick code.

ISA	Pages	Words	Hours to read	Weeks to read
RISC-V	236	76,702	6	0.2
ARM-32	2736	895,032	79	1.9
x86-32	2198	2,186,259	182	4.5

Figure 1.6: Number of pages and words of ISA manuals [Waterman and Asanović 2017a], [Waterman and Asanović 2017b], [Intel Corporation 2016], [ARM Ltd. 2014]. Hours and weeks to complete assumes reading at 200 words per minute for 40 hours a week. Based in part of Figure 1 of [Baumann 2017].



One indication of complexity is the size of the documentation. Figure 1.6 shows the size of the instruction set manuals for RISC-V, ARM-32, and x86-32 measured in pages and words. If you read manuals as a full-time job—8 hours a day for 5 days a week—it would take half a month to make a single pass over the ARM-32 manual and a full month for the x86-32. At this level of intricacy, perhaps no single person fully understands ARM-32 or x86-32. Using this common-sense metric, RISC-V is $\frac{1}{12}$ complexity of the ARM-32 and $\frac{1}{10}$ to $\frac{1}{30}$ the complexity of x86-32. Indeed, the summary of RISC-V ISA including all extensions is only two pages (see the Reference Card).

This minimal, open ISA was unveiled in 2011 and is now backed by a foundation that will evolve it by adding optional extensions based strictly on technical justifications after a prolonged debate. The openness enables free, shared implementations of RISC-V, which lowers costs and the odds of unwanted malicious secrets being hidden in a processor.

However, hardware alone does not a system make. Software development costs likely dwarf hardware development costs, so while stable hardware is important, stable software is more so. It needs operating systems, boot-loaders, reference software, and popular software tools. The foundation offers stability for the overall ISA, and the frozen base means that the RV32I core that is the target for the software stack will never change. By its broad adoption and openness, RISC-V can challenge the dominance of the prevailing proprietary ISAs.

Elegant is a word rarely applied to ISAs, but after reading this book, you may agree with us that it applies to RISC-V. We'll highlight features that we believe indicate elegance with a Mona Lisa icon in the margins.

1.6 To Learn More

ARM Ltd. ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition, 2014. URL `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/`.

A. Baumann. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 132–137. ACM, 2017.

C. Celio, D. Patterson, and K. Asanovic. The Berkeley Out-of-Order Machine (BOOM): an industry-competitive, synthesizable, parameterized RISC-V processor. *Tech. Rep. UCB/EECS-2015–167, EECS Department, University of California, Berkeley*, 2015.

S. Gal-On and M. Levy. Exploring CoreMark - a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012.

Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference*. September 2016.

S. P. Morse. The Intel 8086 chip and the future of microprocessor design. *Computer*, 50(4): 8–9, 2017.

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

S. Rodgers and R. Uhlig. X86: Approaching 40 and still going strong, 2017.

J. L. von Neumann, A. W. Burks, and H. H. Goldstine. Preliminary discussion of the logical design of an electronic computing instrument. *Report to the U.S. Army Ordnance Department*, 1947.

A. Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html`.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10*. May 2017a. URL `https://riscv.org/specifications/privileged-isa/`.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017b. URL `https://riscv.org/specifications/`.

Notes

[1] `http://parlab.eecs.berkeley.edu`

2 RV32I: RISC-V Base Integer ISA

Frances Elizabeth "Fran" Allen (1932-) was bestowed the Turing Award primarily for her work on optimizing compilers. The Turing Award is the greatest prize in Computer Science.

. . . the only way to realistically realize the performance goals and make them accessible to the user was to design the compiler and the computer at the same time. In this way features would not be put in the hardware which the software could not use . . .

—Frances Elizabeth "Fran" Allen, 1981

2.1 Introduction

Figure 2.1 is a one-page graphical representation of the RV32I base instruction set. You can see the full RV32I instruction set by concatenating the underlined letters from left to right for each diagram. The set notation using { } lists the possible variations of the instruction, using either underlined letters or the underscore character _, which means no letter for this variation. For example

$$\underline{\text{s}}\text{et } \underline{\text{l}}\text{ess } \underline{\text{t}}\text{han } \left\{ \begin{array}{l} \_ \\ \underline{\text{i}}\text{mmediate} \end{array} \right\} \left\{ \begin{array}{l} \_ \\ \underline{\text{u}}\text{nsigned} \end{array} \right\}$$

represents these four RV32I instructions: `slt`, `slti`, `sltu`, `sltiu`.

The goal of these diagrams, which will be the first figure of the following chapters, is give a quick, insightful overview of the instructions of a chapter.

2.2 RV32I Instruction formats

Figure 2.2 shows the six base instruction formats: R-type for register-register operations; I-type for short immediates and loads; S-type for stores; B-type for conditional branches; U-type for long immediates; and J-type for unconditional jumps. Figure 2.3 lists the opcodes of the RV32I instructions in Figure 2.1 using the formats of Figure 2.2.

Even the instruction formats demonstrate several examples where the simpler RISC-V ISA improves cost-performance. First, there are only six formats and all instructions are 32 bits long, which simplifies instruction decoding. ARM-32 and particularly x86-32 have numerous formats, which make decoding expensive in low-end implementations and a performance challenge for medium and high-end processor designs. Second, RISC-V instructions offer three register operands, rather than having one field shared for source and destination, as with x86-32. When an operation naturally has three distinct operands but the ISA provides