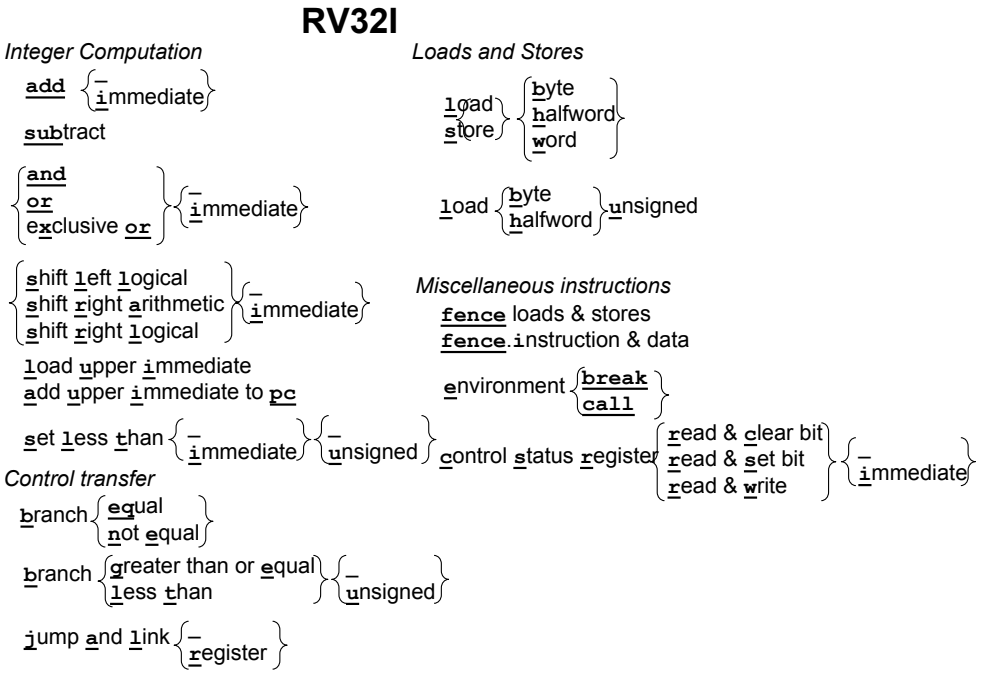


Figure 2.1: Diagram of the RV32I instructions. The underlined letters are concatenated from left to right to form RV32I instructions. The curly bracket notation { } means each vertical item in the set is a different variation of the instruction. The underscore _ within a set means that one option is simply the instruction name so far without a letter from this set. For example, the notation near the upper left-hand corner represents the following six instructions: `and`, `or`, `xor`, `andi`, `ori`, `xori`.

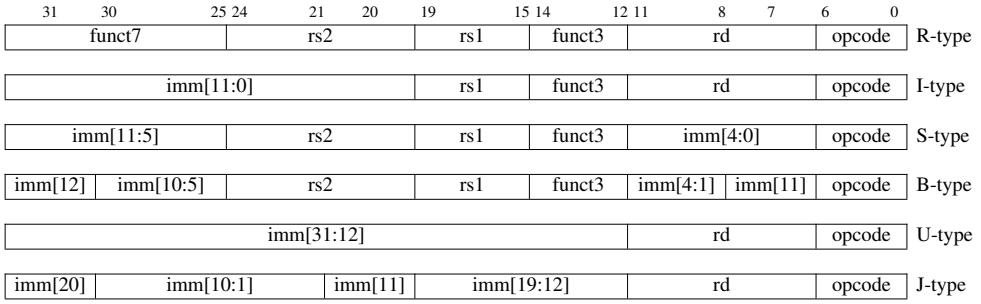


Figure 2.2: RISC-V instruction formats. We label each immediate subfield with the bit position (imm[*x*]) in the immediate value being produced, rather than the bit position in the instruction's immediate field as is usually done. Chapter 10 explains how the control status register instructions use the I-type format slightly differently. (Figure 2.2 of Waterman and Asanović 2017 is the basis of this figure).

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred	succ	00000	000	00000	0001111	I fence
0000 0000	0000	00000	001	00000	0001111	I fence.i
000000000000		00000	000	00000	1110011	I ecall
000000000001		00000	000	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I csrrsi
csr		zimm	111	rd	1110011	I csrrci

Figure 2.3: RV32I opcode map has instruction layout, opcodes, format type, and names. (Table 19.2 of [Waterman and Asanović 2017] is the basis of this figure.)

only a two-operand instruction, the compiler or assembly language programmer must use an extra move instruction to preserve the destination operand. Third, in RISC-V the specifiers of the registers to be read and written are always in the same location in all instructions, which means the register accesses can begin before decoding the instruction. Many other ISAs reuse a field as a source in some instructions and as a destination in others (e.g., ARM-32 and MIPS-32), which forces addition of extra hardware to be placed in a potentially time-critical path to select the proper field. Fourth, the immediate fields in these formats are always sign extended, and the sign bit is always in the most significant bit of the instruction. This decision means sign extension of the immediate, which may also be on a critical timing path, can proceed before decoding the instruction.

Sign-extended immediates even help logical instructions. For example, x & 0xffffffff0 uses only the single instruction `andi` in RISC-V, but it requires two instructions in MIPS-32 (`addiu` to load the constant, then `and`), since MIPS zero-extends logical immediates. ARM-32 needed to add an additional instruction, `bic`, that performs rx & immediate to compensate for zeroextending immediates.

■ ***Elaboration: B- and J-type formats?***

As mentioned below, the immediate field is rotated 1 bit for branch instructions, a variation of the S format that we relabel the B format. The immediate field of jump instructions rotated 12 bits for jump instructions, a variation of the U format relabeled J format. Hence, there are a really four basic formats, but we can conservatively count RISC-V as having six formats.

To help programmers, a bit pattern of all zeros is an illegal RV32I instruction. Thus, erroneous jumps into zeroed memory regions will immediately trap, an aid to debugging. Similarly, the bit pattern of all ones is an illegal instruction, which will trap other common errors such as unprogrammed non-volatile memory devices, disconnected memory buses, or broken memory chips.

To leave ample room for ISA extensions, the base RV32I ISA uses less than 1/8-th of the encoding space in the 32-bit instruction word. The architects also carefully picked the RV32I opcodes so that instructions with common datapath operations share as many of the same opcode bit values as possible, which simplifies the control logic. Finally, as we shall see, the branch and jump addresses in the B and J formats must be shifted left 1 bit so as to multiply the addresses by 2, thereby giving branches and jumps a greater range. RISC-V rotates the bits in the immediate operands from a natural placement to reduce the instruction signal fanout and immediate multiplexing cost by almost a factor for two, which again simplifies datapath logic on low-end implementations.

RISC-V implementations all use the same opcodes for the optional extensions such as RV32M, RV32F, and so on. Non-standard extensions that are unique to processor are restricted to a reserved opcode space in RISC-V.

What's Different? We'll end each section in this and following chapters with description on how RISC-V differs from other ISAs. The contrast is often what RISC-V is missing. Architects demonstrate good taste by the features they omit as well as by what they include.

The ARM-32 12-bit immediate field is not simply a constant but an input to a function that produces a constant: 8 bits are zero-extended to full width and then rotated right by the value in the 4 remaining bits multiplied by 2. The hope was encoding more useful constants in 12 bits would reduce the number of executed instructions. ARM-32 also dedicates a precious four bits in most instruction formats to conditional execution. Despite being infrequently used, conditional execution adds to the complexity of *out-of-order processors*.

■ ***Elaboration: Out-of-order processors***

are high-speed, pipelined processors that execute instructions opportunistically instead of in lock-step program order. A critical feature of such processors is *register renaming*, which maps the register names in the program onto a larger number of internal physical registers. The problem with conditional execution is that the registers in these instructions must be allocated to internal physical registers whether or not the condition holds, yet internal physical register availability is a critical performance resource for out-of-order processors.

2.3 RV32I Registers

Figure 2.4 lists the RV32I registers and their names as determined by the RISC-V application binary interface (ABI). We will use the ABI names in our code examples to make them easier to read. To the joy of assembly language programmers and compiler writers, RV32I has 31 registers plus x0, which always has the value 0. ARM-32 has merely 16 registers while x86-32 has only 8!

What's Different? Dedicating a register to zero is a surprisingly large factor in simplifying the RISC-V ISA. Figure 3.3 on page 36 in Chapter 3 gives many examples of operations that are native instructions in ARM-32 and x86-32, which don't have a zero register, but can be synthesized from RV32I instructions simply by using the zero register as an operand.

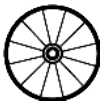
Pipelining is used by all but the cheapest processors today to get good performance. Like an industrial assembly line, they get higher throughput by overlapping the execution of many instructions at once. To pull this off, the processors predict branch outcomes, which they can do with more than 90% accuracy. When they mispredict, they re-execute instructions. Early microprocessors had a 5-stage pipeline, which meant 5 instructions overlapped execution. Recent ones have more than 10 pipeline stages.

The PC is one of ARM-32's 16 registers, which means that any instruction that changes a register may also as a side effect be a branch instruction. The PC as a register complicates hardware branch prediction, whose accuracy is vital for good pipelined-performance, since every instruction might be a branch instead of 10–20% of instructions executed in programs for typical ISAs. It also means one less general-purpose register.

2.4 RV32I Integer Computation

Appendix A gives details of all of the RISC-V instructions, including formats and opcodes. In this section, and similar sections of the following chapters, we give an ISA overview that should be sufficient for knowledgeable assembly language programmers, as well as highlight the features that demonstrate the seven ISA metrics from Chapter 1.

The simple arithmetic instructions (`add`, `sub`), logical instructions (`and`, `or`, `xor`), and shift instructions (`sll`, `srl`, `sra`) in Figure 2.1 are just as you would expect in any ISA. They read two 32-bit values from registers and write a 32-bit result to the destination register. RV32I also offers immediate versions of these instructions. Unlike ARM-32, immediates are always sign-extended so that they can be negative when needed, which is why there is no need for an immediate version of `sub`.



Programs can generate a Boolean value from the result of a comparison. To accommodate such cases, RV32I offers a *set less than* instruction, which sets the destination register to 1 if the first operand is less than the second, or 0 otherwise. As one would expect, there is a signed version (`slt`) and an unsigned version (`sltu`) for signed and unsigned integers as well as immediate versions for both (`slti`, `sltiu`). As we shall see, while RV32I branches can check for all relationships between two registers, some conditional expressions involve relationships between many pairs of registers. The compiler or assembly language programmer could use `slt` and the logical instructions `and`, `or`, `xor` to resolve more elaborate conditional expressions.

The two remaining integer computation instructions Figure 2.1 help with assembly and linking. *Load upper immediate* (`lui`) loads a 20-bit constant into the most significant 20 bits of a register. It can be followed by a standard immediate instruction to create a 32-bit constant from only two 32-bit RV32I instructions. *Add upper immediate to PC* (`auipc`) supports two-instruction sequences to access arbitrary offsets from the PC for both control-flow transfers and data accesses. The combination of an `auipc` and the 12-bit immediate in a `jalr` (see below) can transfer control to any 32-bit PC-relative address, while an `auipc` plus the 12-bit immediate offset in regular load or store instructions can access any 32-bit PC-relative data address.

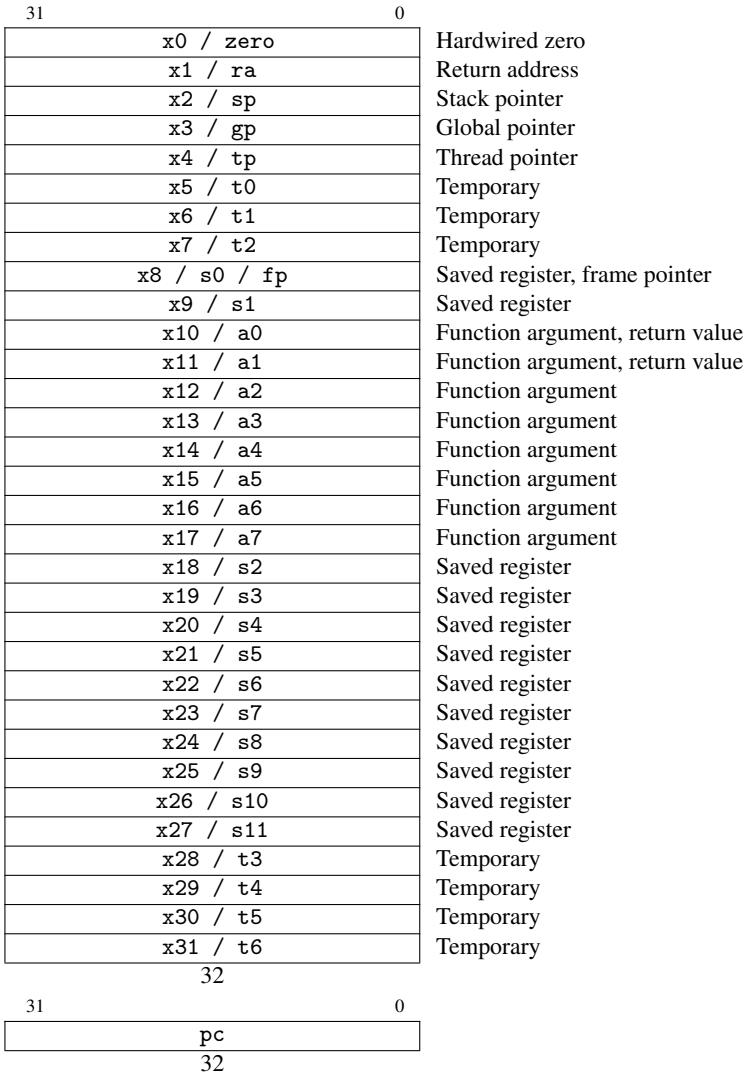
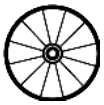


Figure 2.4: The registers of RV32I. Chapter 3 explains the RISC-V calling convention, the rationale behind the various pointers (sp, gp, tp, fp), Saved registers (s0-s11), and Temporaries (t0-t6). (Figure 2.1 and Table 20.1 of [Waterman and Asanović 2017] is the basis of this figure.)



What's Different? First, there are no byte or half-word integer computation operations. The operations are always the full register width. Memory accesses take orders of magnitude more energy than arithmetic operations, so narrow data accesses can save significant energy, but narrow operations do not. ARM-32 has the unusual feature of having an option to shift one of the operands in most arithmetic-logic operations, which complicates the datapath and is rarely needed [Hohl and Hinds 2016]; RV32I has separate shift instructions.

Nor does RV32I include multiply and divide; they comprise the optional RV32M extension (see Chapter 4). Unlike ARM-32 and x86-32, the full RISC-V software stack can run without them, which can shrink embedded chips. While not a hardware issue, the MIPS-32 *assembler* may replace a multiply with a sequence shifts and adds to try to improve performance, which may confuse the programmer seeing instructions executed not found in the assembly language program. RV32I also omits rotate instructions and detection of integer arithmetic overflow. Both can be calculated in a few RV32I instructions (see Section 2.6).

■ ***Elaboration: "Bit twiddling" instructions***

such as rotate are under consideration by the RISC-V Foundation as part of an optional instruction extension called RV32B (see Chapter 11).

■ ***Elaboration: `xor` enables a magic trick.***

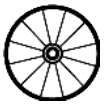
You can exchange two values without using an intermediate register! This code exchanges the values of x1 and x2. We leave the proof to the reader. Hint: exclusive or is commutative ($a \oplus b = b \oplus a$), associative ($(a \oplus b) \oplus c = a \oplus (b \oplus c)$), is its own inverse ($a \oplus a = 0$), and has an identity ($a \oplus 0 = a$).

```
xor  x1,x1,x2 # x1'  == x1^x2, x2' == x2
xor  x2,x1,x2 # x1'  == x1^x2, x2' == x1'^x2 == x1^x2^x2 == x1
xor  x1,x1,x2 # x1'' == x1'^x2' == x1^x2^x1 == x1^x1^x2 == x2, x2' == x1
```

However fascinating, RISC-V's ample register set usually lets compilers find a scratch register, so it rarely uses the XOR-swap.

2.5 RV32I Loads and Stores

As well as providing loads and stores of 32-bit words (`lw`, `sw`), Figure 2.1 shows that RV32I has loads for signed and unsigned bytes and halfwords (`lb`, `lbu`, `lh`, `lhu`) and stores for bytes and halfwords (`sb`, `sh`). Signed bytes and halfwords are sign-extended to 32 bits and written to the destination registers. This widening of narrow data allows subsequent integer computation instructions to operate correctly on all 32 bits, even if the natural data types are narrower. Unsigned bytes and halfwords, useful for text and unsigned integers, are zero-extended to 32 bits before being written to the destination register.



The only addressing mode for loads and stores is adding a sign-extended 12-bit immediate to a register, called displacement addressing mode in x86-32 [Irvine 2014].

What's Different? RV32I omitted the sophisticated addressing modes of ARM-32 and x86-32. Alas, all ARM-32 addressing modes aren't available for all data types, but RV32I addressing does not discriminate against any data type. RISC-V can imitate some x86 addressing modes. For example, setting the immediate field to 0 has the same effect as the

register-indirect addressing mode. Unlike x86-32, RISC-V has no special stack instructions. By using one of the 31 registers as the stack pointer (see Figure 2.4), the standard addressing mode gets most of the benefits of `push` and `pop` instructions without the added ISA complexity. Unlike MIPS-32, RISC-V rejected *delayed load*. Similar in spirit to delayed branches, MIPS-32 redefined the load so the data is unavailable until two instructions later, when it would show up in a five-stage pipeline. Whatever benefit it had evaporated for the longer pipelines that came later.

While ARM-32 and MIPS-32 require data to be aligned naturally to data-sized boundaries in memory, RISC-V does not. Misaligned accesses are sometimes required when porting legacy code. One option is to disallow misaligned accesses in the base ISA and then provide some separate instructions support for misaligned accesses, such as Load Word Left and Load Word Right of MIPS-32. This option would complicate register access, however, since `lwl` and `lwr` require writing pieces of registers instead of simply full registers. Requiring instead that the regular loads and stores support misaligned accesses simplified the overall design.

■ ***Elaboration: Endianness***

RISC-V chose *little-endian* byte ordering because it is dominant commercially: all x86-32 systems, and Apple iOS, Google Android OS, and Microsoft Windows for ARM are all little-endian. Since the endian order matters only when accessing the identical data both as a word and as bytes, endianness affects few programmers.

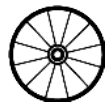
2.6 RV32I Conditional Branch

RV32I can compare two registers and branch on the result if they are equal (`beq`), not equal (`bne`), greater than or equal (`bge`), or less than (`blt`). The latter two cases are signed comparisons, but RV32I also offers unsigned versions: `bgeu` and `bltu`. The two remaining relationships (greater than and less than or equal) can be checked simply by reversing the operands, since $x < y$ means that $y > x$ and $x \geq y$ implies $y \leq x$.

`bltu` **allows signed array bounds to be checked with a single instruction**, since any negative index will compare greater than any nonnegative bound!

Since RISC-V instructions must be a multiple of two bytes long—see Chapter 7 to learn about the optional 2-byte instructions—the branch addressing mode multiplies the 12-bit immediate by 2, sign-extends it, and then adds it to the PC. PC-relative addressing helps with position independent code and thereby reduces the work of the linker and loader (Chapter 3).

What's Different? As noted above, RISC-V excluded the infamous delayed branch of MIPS-32, Oracle SPARC, and others. It also avoided the condition codes of ARM-32 and x86-32 for conditional branches. They add extra state that is implicitly set by most instructions, which needlessly complicate the dependence calculation for out-of-order execution. Finally, it omitted the loop instructions of the x86-32: `loop`, `loope`, `loopz`, `loopne`, `loopnz`.



■ ***Elaboration: Multiword addition without condition codes***

is done as follows in RV32I by using `sltu` to calculate the carry-out:

```
add  a0,a2,a4 # add lower 32 bits: a0 = a2 + a4
sltu a2,a0,a2 # a2' = 1 if (a2+a4) < a2, a2' = 0 otherwise
add  a5,a3,a5 # add upper 32 bits: a5 = a3 + a5
add  a1,a2,a5 # add carry-out from lower 32 bits
```

■ ***Elaboration: Reading the PC***

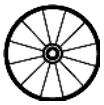
The current PC can be obtained by setting the U-immediate field of `auipc` to 0. For the x86-32, to read the PC you need to call a function (which pushes the PC to the stack); the callee then reads the pushed PC from the stack, and finally returns the PC (by popping the stack). So reading the current PC took 1 store, 2 loads, and 2 taken jumps!

■ ***Elaboration: Software checking of overflow***

Most but not all programs ignore integer arithmetic overflow, so RISC-V relies on software overflow checking. Unsigned addition requires only a single additional branch instruction after the addition: `addu t0, t1, t2; bltu t0, t1, overflow`.
For signed addition, if one operand's sign is known, overflow checking requires only a single branch after the addition: `addi t0, t1, +imm; blt t0, t1, overflow`.
This covers the common case of addition with an immediate operand. For general signed addition, three additional instructions after the addition are required, observing that the sum should be less than one of the operands if and only if the other operand is negative.

```
add t0, t1, t2
slti t3, t2, 0       # t3 = (t2<0)
slt t4, t0, t1       # t4 = (t1+t2<t1)
bne t3, t4, overflow # overflow if (t2<0) && (t1+t2>=t1)
                     #          || (t2>=0) && (t1+t2<t1)
```

2.7 RV32I Unconditional Jump



The single *jump and link* instruction (`jal`) in Figure 2.1 serves dual functions. To support procedure calls, it saves the address of the next instruction PC+4 into the destination register, normally the return address register `ra` (see Figure 2.4). To support unconditional jumps, we use the zero register (x0) instead of `ra` as the destination register, as x0 can't be changed. Like branches, `jal` multiplies its 20-bit branch address by 2, sign extends it, and then adds the result to the PC to form the jump address.

The register version of jump and link (`jalr`) is similarly multipurpose. It can call a procedure to a dynamically calculated address or simply perform a procedure return by selecting the `ra` as the source register, and the zero register (x0) again as the destination register. Switch or case statements, which calculate a jump address, can also use `jalr` with the zero register as the destination register.

Register windows accelerated function call by having many more registers than 32. A new function would get a new set or *window* of 32 registers on a call. To pass arguments, the windows overlapped, meaning some registers were in two adjacent windows.

What's Different? RV32I shunned intricate procedure call instructions, such as the `enter` and `leave` instructions of the x86-32, or *register windows* as found in the Intel Itanium, Oracle SPARC, and Cadence Tensilica.

2.8 RV32I Miscellaneous

The Control Status Register instructions (`csrrc`, `csrrs`, `csrrw`, `csrrci`, `csrrsi`, `csrrwi`) in Figure 2.1 provide easy access to registers that help measure program performance. These 64-bit counters, which can be read 32 bits at a time, measure wall clock time, clock cycles executed, and number of instructions retired.

```
void insertion_sort(long a[], size_t n)
{
  for (size_t i = 1, j; i < n; i++) {
    long x = a[i];
    for (j = i; j > 0 && a[j-1] > x; j--) {
      a[j] = a[j-1];
    }
    a[j] = x;
  }
}
```

Figure 2.5: Insertion Sort in C. While simple, Insertion Sort has many advantages over complicated sorting algorithms: it is memory efficient and fast for small data sets while being adaptive, stable, and online. GCC compilers produced the code for the following four figures. We set the optimization flags to reduce code size, as that produced the easiest to understand code.

ISA	ARM-32	ARM Thumb-2	MIPS-32	microMIPS	x86-32	RV32I	RV32I+RVC
Instructions	19	18	24	24	20	19	19
Bytes	76	46	96	56	45	76	52

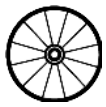
Figure 2.6: Number of instructions and code size for Insertion Sort for these ISAs. Chapter 7 describes ARM Thumb-2, microMIPS, and RV32C.

The `ecall` instruction makes requests to the supporting execution environment, such as system calls. Debuggers use the `ebreak` instruction to transfer control to a debugging environment.

The `fence` instruction sequences device I/O and memory accesses as viewed by other threads and by external devices or coprocessors. The `fence.i` instruction synchronizes the instruction and data streams. RISC-V does not guarantee that stores to instruction memory are visible to instruction fetches in the same processor until a `fence.i` instruction executes.

Chapter 10 covers the RISC-V system instructions.

What's Different? RISC-V uses memory mapped I/O instead of the `in`, `ins`, `insb`, `insw` and `out`, `outs`, `outsb`, `outsw` instructions of the x86-32. It supports strings using byte loads and stores instead of the 16 special string instructions of the x86-32 `rep`, `movs`, `coms`, `scas`, `lods`, ...



2.9 Comparing RV32I, ARM-32, MIPS-32, and x86-32 using Insertion Sort

We've introduced the RISC-V base instruction set, and commented upon its choices as compared to ARM-32, MIPS-32, and x86-32. We'll now do a head-to-head comparison. Figure 2.5 shows Insertion Sort in C, which will be our benchmark. Figure 2.6 is a table that summarizes the number of instructions and number of bytes in Insertion Sort for the ISAs.

Figures 2.8 to 2.11 show the compiled code for RV32I, ARM-32, MIPS-32, and x86-32. Despite the emphasis on simplicity, the RISC-V version uses the same or fewer instructions, and the code sizes of the architectures are quite close. In this example, the compare-and-execute branches of RISC-V save as many instructions as do the fancier address modes and the push and pop instructions of ARM-32 and x86-32 in Figures 2.9 and 2.11.

We move the code examples to after the end of the chapter text to maintain the flow of the writing in this and following chapters.

2.10 Concluding Remarks

Those who cannot remember the past are condemned to repeat it.

—George Santayana, 1905

Figure 2.7 uses the seven metrics of ISA design from Chapter 1 to organize the lessons from past ISAs listed it the previous sections, and shows the positive outcomes for RV32I. We're *not* implying that RISC-V is the first ISA to have those outcomes. Indeed, RV32I inherits the following from RISC-I, its great-great-grandparent [Patterson 2017]:

The genealogy of all RISC-V instructions is chronicled in [Chen and Patterson 2016].

- 32-bit byte-addressable address space
- All instructions are 32-bit long
- 31 registers, all 32 bits wide, with register 0 hardwired to zero
- All operations are between registers (none are register-to-memory)
- Load/store word plus signed and unsigned load/store byte and halfword
- Immediate option for all arithmetic, logical, and shift instructions
- Immediates always sign-extend
- One data addressing mode (register + immediate) and PC-relative branching
- No multiply or divide instructions
- An instruction to load a wide immediate into the upper part of register so that a 32-bit constant takes only two instructions

RISC-V benefits from starting one-quarter to one-third century later, which allowed its architects to follow Santayana's advice to borrow the good ideas but to not repeat the mistakes of the past—including those of RISC-I—in the current RISC-V ISA. Moreover, the RISC-V Foundation will grow the ISA slowly via optional extensions to prevent the rampant incrementalism that has plagued successful ISAs of the past.

The Lindy effect [Lin 2017] observes that the future life expectancy of a technology or idea is proportional to its age. It has stood the test of time, so the longer it has survived in the past, the longer it likely will survive in the future. If that hypothesis holds, RISC architecture may be a good idea for a long time.

■ ***Elaboration: Is RV32I unique?***

Early microprocessors had separate chips for floating-point arithmetic, so those instructions were optional. Moore's Law soon brought everything on chip, and modularity faded in ISAs. Subsetting the full ISA in simpler processors and trapping to software to emulate them goes back decades, with the IBM 360 model 44 and the Digital Equipment microVAX as examples. RV32I is different in that the full software stack needs only the base instructions, so an RV32I processor need not trap repeatedly for omitted instructions in RV32G. Probably the closest ISA to RISC-V in that respect is the Tensilica Xtensa, which is aimed at embedded applications. Its 80-instruction base ISA is intended to be extended by users with custom instructions that accelerate their applications. RV32I has a simpler base ISA, has a 64-bit address version, and offers extensions that target supercomputers as well as microcontrollers.

2.11 To Learn More

Lindy effect, 2017. URL `https://en.wikipedia.org/wiki/Lindy_effect`.

	Mistakes of the Past			Lessons learned
	ARM-32 (1986)	MIPS-32 (1986)	x86-32 (1978)	RV32I (2011)
Cost	Integer multiply mandatory	Integer multiply and divide mandatory	8-bit and 16-bit operations. Integer multiply and divide mandatory	No 8-bit and 16-bit operations. Integer multiply and divide optional (RV32M)
Simplicity	No zero register. Conditional instruction execution. Complex data address modes. Stack instructions (push/pop). Shift-option for arithmetic/logic instructions	Zero- and sign-extended immediates. Some arithmetic instructions can cause overflow traps	No zero register. Complex procedure call/return instructions (enter/leave). Stack instructions (push/pop). Complex data address modes. Loop instructions	Register x0 dedicated to 0. Immediates only sign-extended. One data addressing mode. No conditional execution. No complex call/return or stack instructions. No traps for arithmetic overflow. Separate shift instructions
Performance	Condition codes for branches. Source and destination registers vary in instruction format. Load multiple. Computed immediates. PC a general purpose register	Source and destination registers vary in instruction format.	Condition codes for branches. At most 2 registers per instruction	Compare and branch instructions (no condition codes). 3 registers per instruction. No load multiple. Source and destination registers fixed in instruction format. Constant immediates. PC not a general purpose register
Isolate architecture from implementation	Exposes the pipeline length when writing the PC as a general purpose register	Delayed branch. Delayed load. HI and LO registers just for multiply and divide	Registers not general purpose (AX, CX, DX, DI, SI have unique uses)	No delayed branch. No delayed load. General purpose registers
Room for growth	Limited available opcode space	Limited available opcode space		Generous available opcode space
Program size	Only 32-bit instructions (+Thumb-2 as separate ISA)	Only 32-bit instructions (+microMIPS as separate ISA)	Byte-variable instructions, but poor choices	32-bit instructions + 16-bit RV32C extension
Ease of programming / compiling / linking	Only 15 registers. Aligned data in memory. Irregular data address modes. Inconsistent performance counters	Aligned data in memory. Inconsistent performance counters	Only 8 registers. No PC-relative data addressing. Inconsistent performance counters	31 registers. Data can be unaligned. PC-relative data addressing. Symmetric data address mode. Performance counters defined in architecture

Figure 2.7: Lessons that RISC-V architects learned from past instruction set mistakes. Often the lesson was simply to avoid ISA "optimizations" of the past. The lessons and mistakes are classified by the seven ISA metrics from Chapter 1. Many features listed under cost, simplicity, and performance could be swapped with each other, as it's a matter of taste, but they are important no matter where they appear.

T. Chen and D. A. Patterson. RISC-V genealogy. Technical Report UCB/EECS-2016-6, EECS Department, University of California, Berkeley, Jan 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-6.html.

W. Hohl and C. Hinds. *ARM Assembly Language: Fundamentals and Techniques*. CRC Press, 2016.

K. R. Irvine. *Assembly language for x86 processors*. Prentice Hall, 2014.

D. Patterson. How close is RISC-V to RISC-I?, 2017.

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

Notes

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

```
# RV32I (19 instructions, 76 bytes, or 52 bytes with RVC)
# a1 is n, a3 points to a[0], a4 is i, a5 is j, a6 is x
   0: 00450693  addi  a3,a0,4    # a3 is pointer to a[i]
   4: 00100713  addi  a4,x0,1    # i = 1
Outer Loop:
   8: 00b76463  bltu  a4,a1,10   # if i < n, jump to Continue Outer loop
Exit Outer Loop:
   c: 00008067  jalr  x0,x1,0    # return from function
Continue Outer Loop:
  10: 0006a803  lw    a6,0(a3)   # x = a[i]
  14: 00068613  addi  a2,a3,0    # a2 is pointer to a[j]
  18: 00070793  addi  a5,a4,0    # j = i
Inner Loop:
  1c: ffc62883  lw    a7,-4(a2)  # a7 = a[j-1]
  20: 01185a63  bge   a6,a7,34   # if a[j-1] <= a[i], jump to Exit Inner Loop
  24: 01162023  sw    a7,0(a2)   # a[j] = a[j-1]
  28: fff78793  addi  a5,a5,-1   # j--
  2c: ffc60613  addi  a2,a2,-4   # decrement a2 to point to a[j]
  30: fe0796e3  bne   a5,x0,1c   # if j != 0, jump to Inner Loop
Exit Inner Loop:
  34: 00279793  slli  a5,a5,0x2  # multiply a5 by 4
  38: 00f507b3  add   a5,a0,a5   # a5 is now byte address oi a[j]
  3c: 0107a023  sw    a6,0(a5)   # a[j] = x
  40: 00170713  addi  a4,a4,1    # i++
  44: 00468693  addi  a3,a3,4    # increment a3 to point to a[i]
  48: fc1ff06f  jal   x0,8       # jump to Outer Loop
```

Figure 2.8: RV32I code for Insertion Sort in Figure 2.5. The address in hexadecimal is on the left, the machine language code in hexadecimal is next, and then the assembly language instruction followed by a comment. RV32I allocates two registers to point to `a[j]` and `a[j-1]`. It has plenty of registers, some of which the ABI sets aside for procedure calls. Unlike the other ISAs, it skips saving and restoring these registers to memory. While the code size is larger than x86-32, using the optional RV32C instructions (see Chapter 7) closes the size gap. Note the compare and branch instructions avoid the three compare instructions that ARM-32 and x86-32 require.

```
# ARM-32 (19 instructions, 76 bytes; or 18 insns/46 bytes with Thumb-2)
# r0 points to a[0], r1 is n, r2 is j, r3 is i, r4 is x
  0: e3a03001 mov  r3, #1               # i = 1
  4: e1530001 cmp  r3, r1               # i vs. n (unnecessary?)
  8: e1a0c000 mov  ip, r0               # ip = a[0]
  c: 212fff1e bxcs lr                   # don't let return address change ISAs
 10: e92d4030 push {r4, r5, lr}         # save r4, r5, return address
Outer Loop:
 14: e5bc4004 ldr  r4, [ip, #4]!        # x = a[i] ; increment ip
 18: e1a02003 mov  r2, r3               # j = i
 1c: e1a0e00c mov  lr, ip               # lr = a[0] (using lr as scratch reg)
Inner Loop:
 20: e51e5004 ldr  r5, [lr, #-4]        # r5 = a[j-1]
 24: e1550004 cmp  r5, r4               # compare a[j-1] vs. x
 28: da000002 ble  38                   # if a[j-1]<=a[i], jump to Exit Inner Loop
 2c: e2522001 subs r2, r2, #1           # j--
 30: e40e5004 str  r5, [lr], #-4        # a[j] = a[j-1]
 34: 1afffff9 bne  20                   # if j != 0, jump to Inner Loop
Exit Inner Loop:
 38: e2833001 add  r3, r3, #1           # i++
 3c: e1530001 cmp  r3, r1               # i vs. n
 40: e7804102 str  r4, [r0, r2, lsl #2] # a[j] = x
 44: 3afffff2 bcc  14                   # if i < n, jump to Outer Loop
 48: e8bd8030 pop  {r4, r5, pc}         # restore r4, r5, and return address
```

Figure 2.9: ARM-32 code for Insertion Sort in Figure 2.5. The address in hexadecimal is on the left, the machine language code in hexadecimal is next, and then the assembly language instruction followed by a comment. Short on registers, ARM-32 saves two of them on the stack for later reuse along with the return address. It uses an addressing mode that scales `i` and `j` to be byte addresses. Given that a branch has the potential to change ISAs between ARM-32 and Thumb-2, `bxcs` first sets the least significant bit of the return address to 0 before saving it. The condition codes save one compare instruction to check `j` after decrementing it, but there are still three compares elsewhere.

```
# MIPS-32 (24 instructions, 96 bytes, or 56 bytes with microMIPS)
# a1 is n, a3 is pointer to a[0], v0 is j, v1 is i, t0 is x
   0: 24860004 addiu a2,a0,4   # a2 is pointer to a[i]
   4: 24030001 li    v1,1      # i = 1
Outer Loop:
   8: 0065102b sltu  v0,v1,a1  # set on i < n
   c: 14400003 bnez  v0,1c     # if i<n, jump to Continue Outer Loop
  10: 00c03825 move  a3,a2     # a3 is pointer to a[j] (slot filled)
  14: 03e00008 jr    ra        # return from function
  18: 00000000 nop             # branch delay slot unfilled
Continue Outer Loop:
  1c: 8cc80000 lw    t0,0(a2)  # x = a[i]
  20: 00601025 move  v0,v1     # j = i
Inner Loop:
  24: 8ce9fffc lw    t1,-4(a3) # t1 = a[j-1]
  28: 00000000 nop             # load delay slot unfilled
  2c: 0109502a slt   t2,t0,t1  # set a[i] < a[j-1]
  30: 11400005 beqz  t2,48     # if a[j-1]<=a[i], jump to Exit Inner Loop
  34: 00000000 nop             # branch delay slot unfilled
  38: 2442ffff addiu v0,v0,-1  # j--
  3c: ace90000 sw    t1,0(a3)  # a[j] = a[j-1]
  40: 1440fff8 bnez  v0,24     # if j != 0, jump to Inner Loop
  44: 24e7fffc addiu a3,a3,-4  # decr. a2 to point to a[j] (slot filled)
Exit Inner Loop:
  48: 00021080 sll   v0,v0,0x2 #
  4c: 00821021 addu  v0,a0,v0  # v0 now byte address oi a[j]
  50: ac480000 sw    t0,0(v0)  # a[j] = x
  54: 24630001 addiu v1,v1,1   # i++
  58: 1000ffeb b     8         # jump to Outer Loop
  5c: 24c60004 addiu a2,a2,4   # incr. a2 to point to a[i] (slot filled)
```

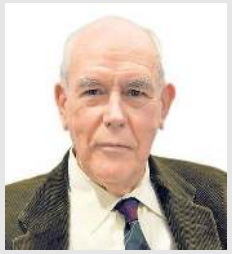
Figure 2.10: MIPS-32 code for Insertion Sort in Figure 2.5. The address in hexadecimal is on the left, the machine language code in hexadecimal is next, and then the assembly language instruction followed by a comment. The MIPS-32 code has three `nop` instructions, which adds to its length. Two are due to delayed branches and one is due to the delayed load. The compiler couldn't find useful instructions to place in those delay slots. The delayed branches also make the code harder to understand, since the instruction that follows is also executed when a branch or jump is taken. For example, the last instruction (`addiu`) at address 5c is part of the loop even though it trails the branch instruction.

```
# x86-32 (20 instructions, 45 bytes)
# eax is j, ecx is x, edx is i
# pointer to a[0] is in memory at address esp+0xc, n is in memory at esp+0x10
  0: 56             push esi                 # save esi on stack (esi needed below)
  1: 53             push ebx                 # save ebx on stack (ebx needed below)
  2: ba 01 00 00 00 mov  edx,0x1             # i = 1
  7: 8b 4c 24 0c    mov  ecx,[esp+0xc]       # ecx is pointer to a[0]
Outer Loop:
  b: 3b 54 24 10    cmp  edx,[esp+0x10]      # compare i vs. n
  f: 73 19          jae  2a <Exit Loop>      # if i >= n, jump to Exit Outer Loop
 11: 8b 1c 91       mov  ebx,[ecx+edx*4]     # x = a[i]
 14: 89 d0          mov  eax,edx             # j = i
Inner Loop:
 16: 8b 74 81 fc    mov  esi,[ecx+eax*4-0x4] # esi = a[j-1]
 1a: 39 de          cmp  esi,ebx             # compare a[j-1] vs. x
 1c: 7e 06          jle  24 <Exit Loop>      # if a[j-1]<=a[i],jump Exit Inner Loop
 1e: 89 34 81       mov  [ecx+eax*4],esi     # a[j] = a[j-1]
 21: 48             dec  eax                 # j--
 22: 75 f2          jne  16 <Inner Loop>     # if j != 0, jump to Inner Loop
Exit Inner Loop:
 24: 89 1c 81       mov  [ecx+eax*4],ebx     # a[j] = x
 27: 42             inc  edx                 # i++
 28: eb e1          jmp  b <Outer Loop>      # jump to Outer Loop
Exit Outer Loop:
 2a: 5b             pop  ebx                 # restore old value of ebx from stack
 2b: 5e             pop  esi                 # restore old value of esi from stack
 2c: c3             ret                      # return from function
```

Figure 2.11: x86-32 code for Insertion Sort in Figure 2.5. The address in hexadecimal is on the left, the machine language code in hexadecimal is next, and then the assembly language instruction followed by a comment. Lacking registers, the x86-32 saves two of them on the stack. Moreover, two of the variables allocated to registers in RV32I are instead kept in memory (`n` and the pointer to `a[0]`). It uses the Scaled Indexed addressing mode to good effect for accessing `a[i]` and `a[j]`. Seven of the 20 x32-86 instructions are one byte long, which gives the x86-32 good code size for this simple program. There are two popular versions of x86 assembly language: Intel/Microsoft and AT&T/Linux. We use the Intel syntax, in part because it matches the operand order of RISC-V, ARM-32, and MIPS-32 with the destination on the left and the source(s) on the right, while the operands are vice versa for AT&T (and the registers prepend a % before their names). This seemingly trivial matter is nearly a religious issue for some programmers. Pedagogy drives our choice, not orthodoxy.

3 RISC-V Assembly Language

Ivan Sutherland (1938-) is called the father of computer graphics for the invention of Sketchpad—the 1962 forerunner of the graphical user interface in modern computing—which led to a Turing Award.



It's very satisfying to take a problem we thought difficult and find a simple solution. The best solutions are always simple.

—Ivan Sutherland

3.1 Introduction

Figure 3.1 shows the four classic steps in translation starting from a C program and ending with a machine-language program ready to run in the computer. This chapter covers the last three steps, but we begin with the role the assembler plays in the RISC-V calling convention.

3.2 Calling convention

There are six general stages in calling a function [Patterson and Hennessy 2017]:

1. Place the arguments where the function can access them.
2. Jump to the function (using RV32I's `jal`).
3. Acquire local storage resources the function needs, saving registers as required.
4. Perform the desired task of the function.
5. Place the function result value where the calling program can access it, restore any registers, and release any local storage resources.
6. Since a function can be called from several points in a program, return control to the point of origin (using `ret`).

To obtain good performance, try to keep variables in registers rather than memory, but on the other hand, avoid going to memory frequently to save and restore these registers.

RISC-V fortunately has enough registers to offer the best of both worlds: keep operands in registers yet reduce the need to save and restore them. The insight is to have some registers that are *not* guaranteed to be preserved across a function call, called *temporary registers*, and some that are, called *saved registers*. Functions that avoid calling other functions are called *leaf* functions. When a leaf function has only a few arguments and local variables, we can keep everything in registers without "spilling" any to memory. If these conditions don't hold,