

RISC-V

Assembly Language Programming

(Draft v0.18.1-0-g23bd3ad)

John Winans
jwinans@niu.edu

October 19, 2022

6 Copyright © 2018, 2019, 2020 John Winans

7 This document is made available under a Creative Commons Attribution 4.0 International License.
8 See [Appendix D](#) for more information.

9 Download your own copy of this book from github here: <https://github.com/johnwinans/rvalp>.

10 This document may contain inaccuracies or errors. The author provides no guarantee regarding the
11 accuracy of this document's contents. If you discover that this document contains errors, please notify
12 the author.

13

14 ARM® is a registered trademark of ARM Limited in the EU and other countries.

15 IBM® is a trademarks or registered trademark of International Business Machines Corporation in the
16 United States, other countries, or both.

17 Intel® and Pentium® are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other
18 countries.

▶ Fix Me:

*Need to say something
about trademarks for things
mentioned in this text*

Contents

19

20	Preface	iv
21	1 Introduction	1
22	1.1 The Digital Computer	1
23	1.2 Instruction Set Architecture	4
24	1.3 How the CPU Executes a Program	4
25	2 Numbers and Storage Systems	6
26	2.1 Boolean Functions	6
27	2.2 Integers and Counting	9
28	2.3 Sign and Zero Extension	19
29	2.4 Shifting	20
30	2.5 Main Memory Storage	21
31	3 The Elements of a Assembly Language Program	28
32	3.1 Assembly Language Statements	28
33	3.2 Memory Layout	28
34	3.3 A Sample Program Source Listing	28
35	3.4 Running a Program With rvddt	29
36	4 Writing RISC-V Programs	32
37	4.1 Use <code>ebreak</code> to Stop rvddt Execution	32
38	4.2 Using the <code>addi</code> Instruction	33
39	4.3 <code>todo</code>	37
40	4.4 Other Instructions With Immediate Operands	37
41	4.5 Transferring Data Between Registers and Memory	37
42	4.6 RR operations	38
43	4.7 Setting registers to large values using <code>lui</code> with <code>addi</code>	38
44	4.8 Labels and Branching	38
45	4.9 Jumps	39
46	4.10 Pseudoinstructions	39

47	4.11 Relocation	41
48	4.12 Relaxation	43
49	5 RV32 Machine Instructions	44
50	5.1 Conventions and Terminology	44
51	5.2 Addressing Modes	48
52	5.3 Instruction Encoding Formats	48
53	5.4 CPU Registers	58
54	5.5 memory	59
55	A Installing a RISC-V Toolchain	60
56	A.1 The GNU Toolchain	60
57	A.2 rvddt	61
58	A.3 qemu	62
59	B Floating Point Numbers	63
60	B.1 IEEE-754 Floating Point Number Representation	63
61	C The ASCII Character Set	69
62	C.1 NAME	69
63	C.2 DESCRIPTION	69
64	C.3 NOTES	71
65	C.4 COLOPHON	71
66	D Attribution 4.0 International	72
67	Bibliography	77
68	Glossary	78
69	Index	79
70	RV32I Reference Cards	81

Preface

71

72 I set out to write this book because I couldn't find it in a single volume elsewhere.

73 The closest published work on this topic appear to be select portions of *The RISC-V Instruction Set*
74 *Manual, Volume I: User-Level ISA, Document Version 2.2*[1], The RISC-V Reader[2], and Computer
75 Organization and Design RISC-V Edition: The Hardware Software Interface[3].

76 There *are* some terse guides on the Internet that are suitable for those who already know an assembly
77 language. With all the (deserved) excitement brewing over system organization (and the need to
78 compress the time out of university courses targeting assembly language programming [4]), it is no
79 surprise that RISC-V texts for the beginning assembly programmer are not (yet) available.

80 When I started in computing, I learned how to count in binary in a high school electronics course using
81 data sheets for integrated circuits such as the 74191[5] and 74154[6] prior to knowing that assembly
82 language even existed.

83 I learned assembly language from data sheets and texts, that are still sitting on my shelves today,
84 such as:

- 85 • The MCS-85 User's Manual[7]
- 86 • The EDTASM Manual[8]
- 87 • The MC68000 User's Manual[9]
- 88 • Assembler Language With ASSIST[10]
- 89 • IBM System/370 Principles of Operation[11]
- 90 • OS/VS-DOS/VSE-VM/370 Assembler Language[12]
- 91 • ... and several others

92 All of these manuals discuss each CPU instruction in excruciating detail with both a logical and
93 narrative description. For RISC-V this is also the case for the *RISC-V Reader*[2] and the *Computer*
94 *Organization and Design RISC-V Edition*[3] books and is also present in this text (I consider that to
95 be the minimal level of responsibility.)

96 Where I hope this text will differentiate itself from the existing RISC-V titles is in its attempt to
97 address the needs of those learning assembly language for the first time. To this end I have primed this
98 project with some of the curriculum material I created when teaching assembly language programming
99 in the late '80s.

100

Chapter 1

101

Introduction

102

At its core, a digital computer has at least one **Central Processing Unit (CPU)**. A CPU executes a continuous stream of instructions called a **program**. These program instructions are expressed in what is called **machine language**. Each machine language instruction is a **binary** value. In order to provide a method to simplify the management of machine language programs a symbolic mapping is provided where a **mnemonic** can be used to specify each machine instruction and any of its parameters... rather than require that programs be expressed as a series of binary values. A set of mnemonics, parameters and rules for specifying their use for the purpose of programming a CPU is called an *Assembly Language*.

110

1.1 The Digital Computer

111

There are different types of computers. A *digital* computer is the type that most people think of when they hear the word *computer*. Other varieties of computers include *analog* and *quantum*.

113

A digital computer is one that processes data represented using numeric values (digits), most commonly expressed in binary (ones and zeros) form.

115

This text focuses on digital computing.

116

A typical digital computer is composed of storage systems (memory, disc drives, USB drives, etc.), a CPU (with one or more cores), input peripherals (a keyboard and mouse) and output peripherals (display, printer or speakers.)

119

1.1.1 Storage Systems

120

Computer storage systems are used to hold the data and instructions for the CPU.

121

Types of computer storage can be classified into two categories: *volatile* and *non-volatile*.

122 **1.1.1.1 Volatile Storage**123 Volatile storage is characterized by the fact that it will lose its contents (forget) any time that it is
124 powered off.125 One type of volatile storage is provided inside the CPU itself in small blocks called **registers**. These
126 registers are used to hold individual data values that can be manipulated by the instructions that are
127 executed by the CPU.128 Another type of volatile storage is *main memory* (sometimes called **RAM**) Main memory is connected
129 to a computer's CPU and is used to hold the data and instructions that can not fit into the CPU
130 registers.131 Typically, a CPU's registers can hold tens of data values while the main memory can contain many
132 billions of data values.133 To keep track of the data values, each register is assigned a number and the main memory is broken
134 up into small blocks called **bytes** that each assigned a number called an **address** (an *address* is often
135 referred to as a *location*).136 A CPU can process data in a register at a speed that can be an order of magnitude faster than the
137 rate that it can process (specifically, transfer data and instructions to and from) the main memory.138 Register storage costs an order of magnitude more to manufacture than main memory. While it is
139 desirable to have many registers, the economics dictate that the vast majority of volatile computer
140 storage be provided in its main memory. As a result, optimizing the copying of data between the
141 registers and main memory is a desirable trait of good programs.142 **1.1.1.2 Non-Volatile Storage**143 Non-volatile storage is characterized by the fact that it will *NOT* lose its contents when it is powered
144 off.145 Common types of non-volatile storage are disc drives, **ROM** flash cards and USB drives. Prices can
146 vary widely depending on size and transfer speeds.

147 It is typical for a computer system's non-volatile storage to operate more slowly than its main memory.

148 This text will focus on volatile storage.

149 **1.1.2 CPU**150 The **CPU** is a collection of registers and circuitry designed to manipulate the register data and to
151 exchange data and instructions with the main memory. The instructions that are read from the
152 main memory tell the CPU to perform various mathematical and logical operations on the data in its
153 registers and where to save the results of those operations.**Fix Me:**

*Add a block diagram of the
CPU components described
here.*

154 **1.1.2.1 Execution Unit**155 The part of a CPU that coordinates all aspects of the operations of each instruction is called the
156 *execution unit*. It is what performs the transfers of instructions and data between the CPU and

157 the main memory and tells the registers when they are supposed to either store or recall data being
158 transferred. The execution unit also controls the ALU (Arithmetic and Logic Unit).

159 **1.1.2.2 Arithmetic and Logic Unit**

160 When an instruction manipulates data by performing things like an *addition*, *subtraction*, *comparison*
161 or other similar operations , the ALU is what will calculate the sum, difference, and so on... under
162 the control of the execution unit.

163 **1.1.2.3 Registers**

164 In the RV32 CPU there are 31 general purpose registers that each contain 32 [bits](#) (where each bit is
165 one [binary](#) digit value of one or zero) and a number of special-purpose registers. Each of the general
166 purpose registers is given a name such as `x1`, `x2`, ... on up to `x31` (*general purpose* refers to the
167 fact that the *CPU itself* does not prescribe any particular function to any of these registers.) Two
168 important special-purpose registers are `x0` and `pc`.

169 Register `x0` will always represent the value zero or logical *false* no matter what. If any instruction
170 tries to change the value in `x0` the operation will fail. The need for *zero* is so common that, other
171 than the fact that it is hard-wired to zero, the `x0` register is made available as if it were otherwise a
172 general purpose register.¹

173 The `pc` register is called the *program counter*. The CPU uses it to remember the memory address
174 where its program instructions are located.

175 The term `XLEN` refer to the width of an integer register in bits (either 32, 64, or 128.) The number
176 of bits in each register is defined by the [Instruction Set Architecture \(ISA\)](#).

177 **1.1.2.4 Harts**

178 Analogous to a *core* in other types of CPUs, a [hart](#) (hardware [thread](#)) in a RISC-V CPU refers to the
179 collection of 32 registers, instruction execution unit and ALU.[1, p. 20]

180 When more than one hart is present in a CPU, a different stream of instructions can be executed
181 on each hart all at the same time. Programs that are written to take advantage of this are called
182 *multithreaded*.

183 This text will primarily focus on CPUs that have only one hart.

184 **1.1.3 Peripherals**

185 A *peripheral* is a device that is not a CPU or main memory. They are typically used to transfer
186 information/data into and out of the main memory.

187 This text is not concerned with the peripherals of a computer system other than in sections where
188 instructions are discussed with the purpose of addressing the needs of a peripheral device. Such
189 instructions are used to initiate, execute and/or synchronize data transfers.

¹Having a special *zero* register allows the total set of instructions that the CPU can execute to be simplified. Thus reducing its complexity, power consumption and cost.

190 1.2 Instruction Set Architecture

191 The catalog of rules that describes the details of the instructions and features that a given CPU
192 provides is called an [Instruction Set Architecture \(ISA\)](#).

193 An ISA is typically expressed in terms of the specific meaning of each binary instruction that a CPU
194 can recognize and how it will process each one.

195 The RISC-V ISA is defined as a set of modules. The purpose of dividing the ISA into modules is to
196 allow an implementer to select which features to incorporate into a CPU design.[\[1, p. 4\]](#)

197 Any given RISC-V implementation must provide one of the *base* modules and zero or more of the
198 *extension* modules.[\[1, p. 4\]](#)

199 1.2.1 RV Base Modules

200 The base modules are RV32I (32-bit general purpose), RV32E (32-bit embedded), RV64I (64-bit
201 general purpose) and RV128I (128-bit general purpose).[\[1, p. 4\]](#)

202 These base modules provide the minimal functional set of integer operations needed to execute a
203 useful application. The differing bit-widths address the needs of different main-memory sizes.

204 This text primarily focuses on the RV32I base module and how to program it.

205 1.2.2 Extension Modules

206 RISC-V extension modules may be included by an implementer interested in optimizing a design for
207 one or more purposes.[\[1, p. 4\]](#)

208 Available extension modules include M (integer math), A (atomic), F (32-bit floating point), D (64-bit
209 floating point), Q (128-bit floating point), C (compressed size instructions) and others.

210 The extension name *G* is used to represent the combined set of IMAFD extensions as it is expected
211 to be a common combination.

212 1.3 How the CPU Executes a Program

213 The process of executing a program is continuous repeats of a series of *instruction cycles* that are each
214 comprised of a *fetch*, *decode* and *execute* phase.

215 The current status of a CPU hart is entirely embodied in the data values that are stored in its registers
216 at any moment in time. Of particular interest to an executing program is the `pc` register. The `pc`
217 contains the memory address containing the instruction that the CPU is currently executing.[^2](#)

218 For this to work, the instructions to be executed must have been previously stored in adjacent main
219 memory locations and the address of the first instruction placed into the `pc` register.

²In the RISC-V ISA the `pc` register points to the *current* instruction where in most other designs, the `pc` register points to the *next* instruction.

220 **1.3.1 Instruction Fetch**221 In order to *fetch* an instruction from the main memory the CPU will update the address in the `pc`
222 register and then request that the main memory return the value of the data stored at that address.
223 ³224 **1.3.2 Instruction Decode**225 Once an instruction has been fetched, it must be inspected to determine what operation(s) are to
226 be performed. This means inspecting the portions of the instruction that dictate which registers are
227 involved and what that, if anything, ALU should do.228 **1.3.3 Instruction Execute**229 Typical instructions do things like add a number to the value currently stored in one of the registers
230 or store the contents of a register into the main memory at some given address.

231 Part of every instruction is a notion of what should be done next.

232 Most of the time an instruction will complete by indicating that the CPU should proceed to fetch and
233 execute the instruction at the next larger main memory address. In these cases the `pc` is incremented
234 to point to the memory address after the current instruction.235 Any parameters that an instruction requires must either be part of the instruction itself or read from
236 (or stored into) one or more of the general purpose registers.237 Some instructions can specify that the CPU proceed to execute an instruction at an address other
238 than the one that follows itself. This class of instructions have names like *jump* and *branch* and are
239 available in a variety of different styles.240 The RISC-V ISA uses the word *jump* to refer to an *unconditional* change in the sequential processing
241 of instructions and the word *branch* to refer to a *conditional* change.

242 Conditional branch instructions can be used to tell the CPU to do things like:

243 If the value in `x8` is currently less than the value in `x24` then proceed to the instruction at
244 the next main memory address, otherwise branch to an instruction at a different address.245 This type of instruction can therefore result in one of two different actions pending the result of the
246 comparison.⁴247 Once the instruction execution phase has completed, the next instruction cycle will be performed
248 using the new value in the `pc` register.

³RV32I instructions are more than one byte in size, but this general description is suitable for now.

⁴This is the fundamental method used by a CPU to make decisions.

Chapter 2

Numbers and Storage Systems

This chapter discusses how data are represented and stored in a computer.

In the context of computing, *boolean* refers to a condition that can be either true or false and *binary* refers to the use of a base-2 numeric system to represent numbers.

RISC-V assembly language uses binary to represent all values, be they boolean or numeric. It is the context within which they are used that determines whether they are boolean or numeric.

► Fix Me:

Add some diagrams here showing bits, bytes and the MSB, LSB, ... perhaps relocated from the RV32I chapter?

2.1 Boolean Functions

Boolean functions apply on a per-bit basis. When applied to multi-bit values, each bit position is operated upon independent of the other bits.

RISC-V assembly language uses zero to represent *false* and one to represent *true*. In general, however, it is useful to relax this and define zero **and only zero** to be *false* and anything that is not *false* is therefore *true*.¹

The reason for this relaxation is to describe the common case where the CPU processes data, multiple bits at-a-time.

These groups have names like **byte** (8 bits), **halfword** (16 bits) and **fullword** (32 bits).

2.1.1 NOT

The *NOT* operator applies to a single operand and represents the opposite of the input.

► Fix Me:

Need to define unary, binary and ternary operators without confusing binary operators with binary numbers.

If the input is 1 then the output is 0. If the input is 0 then the output is 1. In other words, the output value is *not* that of the input value.

Expressing the *not* function in the form of a truth table:

¹This is how *true* and *false* behave in C, C++, and many other languages as well as the common assembly language idioms discussed in this text.

271

A	\bar{A}
0	1
1	0

272 A truth table is drawn by indicating all of the possible input values on the left of the vertical bar
273 with each row displaying the output values that correspond to the input for that row. The column
274 headings are used to define the illustrated operation expressed using a mathematical notation. The
275 *not* operation is indicated by the presence of an *overline*.

276 In computer programming languages, things like an overline can not be efficiently expressed using a
277 standard keyboard. Therefore it is common to use a notation such as that used by the C language
278 when discussing the *NOT* operator in symbolic form. Specifically the tilde: ‘~’.

279 It is also uncommon for programming languages to express boolean operations on single-bit input(s).
280 A more generalized operation is used that applies to a set of bits all at once. For example, performing
281 a *not* operation of eight bits at once can be illustrated as:

282 $\sim 1\ 1\ 1\ 1\ 0\ 1\ 0\ 1 \quad \text{<= A}$
283 \hline
284 $0\ 0\ 0\ 0\ 1\ 0\ 1\ 0 \quad \text{<= output}$

285 In a line of code the above might read like this: `output = ~A`

286 2.1.2 AND

287 The boolean *and* function has two or more inputs and the output is a single bit. The output is 1 if
288 and only if all of the input values are 1. Otherwise it is 0.

289 This function works like it does in spoken language. For example if A is 1 *and* B is 1 then the output
290 is 1 (true). Otherwise the output is 0 (false).

291 In mathematical notion, the *and* operator is expressed the same way as is *multiplication*. That is by a
292 raised dot between, or by juxtaposition of, two variable names. It is also worth noting that, in base-2,
293 the *and* operation actually *is* multiplication!

294

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

295 This text will use the operator used in the C language when discussing the *and* operator in symbolic
296 form. Specifically the ampersand: ‘&’.

297 An eight-bit example:

298 $1\ 1\ 1\ 1\ 0\ 1\ 0\ 1 \quad \text{<= A}$
299 $\&\ 1\ 0\ 0\ 1\ 0\ 0\ 1\ 1 \quad \text{<= B}$
300 \hline
301 $1\ 0\ 0\ 1\ 0\ 0\ 0\ 1 \quad \text{<= output}$

302 In a line of code the above might read like this: `output = A & B`

303 **2.1.3 OR**

304 The boolean *or* function has two or more inputs and the output is a single bit. The output is 1 if at
 305 least one of the input values are 1.

306 This function works like it does in spoken language. For example if A is 1 *or* B is 1 then the output
 307 is 1 (true). Otherwise the output is 0 (false).

308 In mathematical notion, the *or* operator is expressed using the plus (+).

309

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

310 This text will use the operator used in the C language when discussing the *or* operator in symbolic
 311 form. Specifically the pipe: ‘|’.

312 An eight-bit example:

313 1 1 1 1 0 1 0 1 <== A
 314 | 1 0 0 1 0 0 1 1 <== B
 315 -----
 316 1 1 1 1 0 1 1 1 <== output

317 In a line of code the above might read like this: `output = A | B`

318 **2.1.4 XOR**

319 The boolean *exclusive or* function has two or more inputs and the output is a single bit. The output
 320 is 1 if only an odd number of inputs are 1. Otherwise the output will be 0.

321 Note that when *xor* is used with two inputs, the output is set to 1 (true) when the inputs have different
 322 values and 0 (false) when the inputs both have the same value.

323 In mathematical notion, the *xor* operator is expressed using the plus in a circle (⊕).

324

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

325 This text will use the operator used in the C language when discussing the *xor* operator in symbolic
 326 form. Specifically the carrot: ‘^’.

327 An eight-bit example:

Decimal			Binary								Hex	
10^2	10^1	10^0	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	16^1	16^0
100	10	1	128	64	32	16	8	4	2	1	16	1
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	1	0	1
0	0	2	0	0	0	0	0	0	1	0	0	2
0	0	3	0	0	0	0	0	0	1	1	0	3
0	0	4	0	0	0	0	0	1	0	0	0	4
0	0	5	0	0	0	0	0	1	0	1	0	5
0	0	6	0	0	0	0	0	1	1	0	0	6
0	0	7	0	0	0	0	0	1	1	1	0	7
0	0	8	0	0	0	0	1	0	0	0	0	8
0	0	9	0	0	0	0	1	0	0	1	0	9
0	1	0	0	0	0	0	1	0	1	0	0	a
0	1	1	0	0	0	0	1	0	1	1	0	b
0	1	2	0	0	0	0	1	1	0	0	0	c
0	1	3	0	0	0	0	1	1	0	1	0	d
0	1	4	0	0	0	0	1	1	1	0	0	e
0	1	5	0	0	0	0	1	1	1	1	0	f
0	1	6	0	0	0	1	0	0	0	0	1	0
0	1	7	0	0	0	1	0	0	0	1	1	1
...			
1	2	5	0	1	1	1	1	1	0	1	7	d
1	2	6	0	1	1	1	1	1	1	0	7	e
1	2	7	0	1	1	1	1	1	1	1	7	f
1	2	8	1	0	0	0	0	0	0	0	8	0

Figure 2.1: Counting in decimal, binary and hexadecimal.

```

328 1 1 1 1 0 1 0 1 <== A
329 ^ 1 0 0 1 0 0 1 1 <== B
330 -----
331 0 1 1 0 0 1 1 0 <== output

```

332 In a line of code the above might read like this: `output = A ^ B`

333 2.2 Integers and Counting

334 A binary integer is constructed with only 1s and 0s in the same manner as decimal numbers are
335 constructed with values from 0 to 9.

336 Counting in binary (base-2) uses the same basic rules as decimal (base-10). The difference is when we
337 consider that there are ten decimal digits and only two binary digits. Therefore, in base-10, we must
338 carry when adding one to nine (because there is no digit representing a ten) and, in base-2, we must
339 carry when adding one to one (because there is no digit representing a two.)

340 Figure 2.1 shows an abridged table of the decimal, binary and hexadecimal values ranging from 0_{10}
341 to 128_{10} .

342 One way to look at this table is on a per-row basis where each **place value** is represented by the

343 base raised to the power of the [place value](#) position (shown in the column headings.) For example to
 344 interpret the decimal value on the fourth row:

$$0 \times 10^2 + 0 \times 10^1 + 3 \times 10^0 = 3_{10} \quad (2.2.1)$$

345 Interpreting the binary value on the fourth row by converting it to decimal:

$$0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 3_{10} \quad (2.2.2)$$

346 Interpreting the hexadecimal value on the fourth row by converting it to decimal:

$$0 \times 16^1 + 3 \times 16^0 = 3_{10} \quad (2.2.3)$$

347 We refer to the place values with the largest exponent (the one furthest to the left for any given base)
 348 as the most significant digit and the place value with the lowest exponent as the least significant
 349 digit. For binary numbers these are the [Most Significant Bit \(MSB\)](#) and [Least Significant Bit \(LSB\)](#)
 350 respectively.²

351 Another way to look at this table is on a per-column basis. When tasked with drawing such a table by
 352 hand, it might be useful to observe that, just as in decimal, the right-most column will cycle through
 353 all of the values represented in the chosen base then cycle back to zero and repeat. (For example, in
 354 binary this pattern is 0-1-0-1-0-1-0-...) The next column in each base will cycle in the same manner
 355 except each of the values is repeated as many times as is represented by the place value (in the case
 356 of decimal, 10^1 times, binary 2^1 times, hex 16^1 times. Again, the binary numbers for this pattern are
 357 0-0-1-1-0-0-1-1-...) This continues for as many columns as are needed to represent the magnitude of
 358 the desired number.

359 Another item worth noting is that any even binary number will always have a 0 LSB and odd numbers
 360 will always have a 1 LSB.

361 As is customary in decimal, leading zeros are sometimes not shown for readability.

362 The relationship between binary and hex values is also worth taking note. Because $2^4 = 16$, there is
 363 a clean and simple grouping of 4 [bits](#) to 1 [hit](#) (aka [nybble](#)). There is no such relationship between
 364 binary and decimal.

365 Writing and reading numbers in binary that are longer than 8 bits is cumbersome and prone to error.
 366 The simple conversion between binary and hex makes hex a convenient shorthand for expressing binary
 367 values in many situations.

368 For example, consider the following value expressed in binary, hexadecimal and decimal (spaced to
 369 show the relationship between binary and hex):

370 **Binary value:** 0010 0111 1011 1010 1100 1100 1111 0101
 371 **Hex Value:** 2 7 B A C C F 5
 372 **Decimal Value:** 666553589

373 Empirically we can see that grouping the bits into sets of four allows an easy conversion to hex and

²Changing the value of the MSB will have a more *significant* impact on the numeric value than changing the value of the LSB.

374 expressing it as such is $\frac{1}{4}$ as long as in binary while at the same time allowing for easy conversion
 375 back to binary.

376 The decimal value in this example does not easily convey a sense of the binary value.

In programming languages like the C, its derivatives and RISC-V assembly, numeric values are interpreted as decimal **unless** they start with a zero (0). Numbers that start with 0 are interpreted as octal (base-8), numbers starting with 0x are interpreted as hexadecimal and numbers that start with 0b are interpreted as binary.

377

2.2.1 Converting Between Bases

379 **2.2.1.1 From Binary to Decimal**

380 It is occasionally necessary to convert between decimal, binary and/or hex.

381 To convert from binary to decimal, put the decimal value of the **place values** ... 8, 4, 2, 1 over the
 382 binary digits like this:

383 Base-2 place values: 128 64 32 16 8 4 2 1
 384 Binary: 0 0 0 1 1 0 1 1
 385 Decimal: 16 +8 +2 +1 = 27

386 Now sum the place-values that are expressed in decimal for each bit with the value of 1: $16 + 8 + 2 + 1$.
 387 The integer binary value 00011011_2 represents the decimal value 27_{10} .

388 **2.2.1.2 From Binary to Hexadecimal**

389 Conversion from binary to hex involves grouping the bits into sets of four and then performing the
 390 same summing process as shown above. If there is not a multiple of four bits then extend the binary
 391 to the left with zeros to make it so.

392 Grouping the bits into sets of four and summing:

393 Base-2 place values: 8 4 2 1 8 4 2 1 8 4 2 1 8 4 2 1
 394 Binary: 0 1 1 0 1 1 0 1 1 0 1 0 1 0 1 1 1 0
 395 Decimal: 4+2 =6 8+4+ 1=13 8+ 2 =10 8+4+2 =14

396 After the summing, convert each decimal value to hex. The decimal values from 0-9 are the same
 397 values in hex. Because we don't have any more numerals to represent the values from 10-15, we use the
 398 first 6 letters (See the right-most column of [Figure 2.1](#).) Fortunately there are only six hex mappings
 399 involving letters. Thus it is reasonable to memorize them.

400 Continuing this example:

401 Decimal: 6 13 10 14
 402 Hex: 6 D A E

403 **2.2.1.3 From Hexadecimal to Binary**404 The four-bit mapping between binary and hex makes this task as straight forward as using a look-up
405 table to translate each [hit](#) (Hex digIT) it to its unique four-bit pattern.406 Perform this task either by memorizing each of the 16 patterns or by converting each hit to decimal
407 first and then converting each four-bit binary value to decimal using the place-value summing method
408 discussed in [section 2.2.1.1](#).

409 For example:

410 Hex: 7 C
411 Decimal Sum: 4+2+1=7 8+4 =12
412 Binary: 0 1 1 1 1 1 0 0413 **2.2.1.4 From Decimal to Binary**414 To convert arbitrary decimal numbers to binary, extend the list of binary place values until it exceeds
415 the value of the decimal number being converted. Then make successive subtractions of each of the
416 place values that would yield a non-negative result.417 For example, to convert 1234_{10} to binary:

418 Base-2 place values: 2048-1024-512-256-128-64-32-16-8-4-2-1

419
420 0 2048 (too big)
421 1 1234 - 1024 = 210
422 0 512 (too big)
423 0 256 (too big)
424 1 210 - 128 = 82
425 1 82 - 64 = 18
426 0 32 (too big)
427 1 18 - 16 = 2
428 0 8 (too big)
429 0 4 (too big)
430 1 2 - 2 = 0
431 0 1 (too big)432 The answer using this notation is listed vertically in the left column with the [MSB](#) on the top and
433 the [LSB](#) on the bottom line: 010011010010₂.434 **2.2.1.5 From Decimal to Hex**435 Conversion from decimal to hex can be done by using the place values for base-16 and the same math
436 as from decimal to binary or by first converting the decimal value to binary and then from binary to
437 hex by using the methods discussed above.438 Because binary and hex are so closely related, performing a conversion by way of binary is straight
439 forward.

2.2.2 Addition of Binary Numbers

The addition of binary numbers can be performed long-hand the same way decimal addition is taught in grade school. In fact binary addition is easier since it only involves adding 0 or 1.

The first thing to note that in any number base $0 + 0 = 0$, $0 + 1 = 1$, and $1 + 0 = 1$. Since there is no “two” in binary (just like there is no “ten” decimal) adding $1 + 1$ results in a zero with a carry as in: $1 + 1 = 10_2$ and in: $1 + 1 + 1 = 11_2$. Using these five sums, any two binary integers can be added.

This truth table shows what is called a *Full Addr*. A full addr is a function that can add three input bits (the two addends and a carry value from a “prior column”) and produce the sum and carry output values.³

ci	a	b	co	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Adding two unsigned binary numbers using 16 full adders:

```

451      111111 1111 <== carries
452      0110101111001111 <== addend
453      + 0000011101100011 <== addend
454      -----
455      0111001100110010 <== sum

```

Note that the carry “into” the LSB is zero.

2.2.3 Signed Numbers

There are multiple methods used to represent signed binary integers. The method used by most modern computers is called *two’s complement*.

A two’s complement number is encoded in such a manner as to simplify the hardware used to add, subtract and compare integers.

A simple method of thinking about two’s complement numbers is to negate the place value of the **MSB**. For example, the number one is represented the same as discussed before:

```

464  Base-2 place values: -128 64 32 16 8 4 2 1
465  Binary:           0 0 0 0 0 0 0 1

```

The **MSB** of any negative number in this format will always be 1. For example the value -1_{10} is:

³Note that the sum could be expressed in Boolean Algebra as: $sum = ci \oplus a \oplus b$

467 Base-2 place values: -128 64 32 16 8 4 2 1
 468 Binary: 1 1 1 1 1 1 1 1

469 ... because: $-128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = -1$.

470 This format has the virtue of allowing the same addition logic discussed above to be used to calculate
 471 the sums of signed numbers as unsigned numbers.

472 Calculating the signed addition: $4 + 5 = 9$

```
473      1 <== carries
 474 000100 <== 4 = 0 + 0 + 0 + 4 + 0 + 0
 475 +000101 <== 5 = 0 + 0 + 0 + 4 + 0 + 1
 476 -----
 477      001001 <== 9 = 0 + 0 + 8 + 0 + 0 + 1
```

478 Calculating the signed addition: $-4 + -5 = -9$

```
479      1 11 <== carries
 480 111100 <== -4 = -32 + 16 + 8 + 4 + 0 + 0
 481 +111011 <== -5 = -32 + 16 + 8 + 0 + 2 + 1
 482 -----
 483      1 110111 <== -9 (with a truncation) = -32 + 16 + 4 + 2 + 1 = -9
```

484 Calculating the signed addition: $-1 + 1 = 0$

```
485      -128 64 32 16 8 4 2 1 <== place value
 486      1 1 1 1 1 1 1 1 <== carries
 487      1 1 1 1 1 1 1 1 <== addend (-1)
 488 + 0 0 0 0 0 0 0 1 <== addend (1)
 489 -----
 490      1 0 0 0 0 0 0 0 <== sum (0 with a truncation)
```

491 In order for this to work, the carry out of the sum of the MSBs **must** be discarded.

492 2.2.3.1 Converting between Positive and Negative

493 Changing the sign on two's complement numbers can be described as inverting all of the bits (which
 494 is also known as the *one's complement*) and then add one.

495 For example, negating the number four:

```
-128 64 32 16 8 4 2 1
 0 0 0 0 0 1 0 0 <== 4

          1 1 <== carries
 496 1 1 1 1 1 0 1 1 <== one's complement of 4
 + 0 0 0 0 0 0 0 1 <== plus 1
 -----
 1 1 1 1 1 1 0 0 <== -4
```

497 This can be verified by adding 5 to the result and observe that the sum is 1:

```

498      -128 64 32 16 8 4 2 1
499      1 1 1 1 1           <== carries
500      1 1 1 1 1 1 0 0 <== -4
501      + 0 0 0 0 0 1 0 1 <== 5
502      -----
503      1 0 0 0 0 0 0 1 <== 1 (with a truncation)

```

504 Note that the changing of the sign using this method is symmetric in that it is identical when converting
 505 from negative to positive and when converting from positive to negative: *flip the bits and add 1*.

506 For example, changing the value -4 to 4 to illustrate the reverse of the conversion above:

```

507      -128 64 32 16 8 4 2 1
508      1 1 1 1 1 1 0 0 <== -4
509
510          1 1 <== carries
511      0 0 0 0 0 0 1 1 <== one's complement of -4
512      + 0 0 0 0 0 0 0 1 <== plus 1
513      -----
514      0 0 0 0 0 1 0 0 <== 4

```

515 2.2.4 Subtraction of Binary Numbers

516 Subtraction of binary numbers is performed by first negating the subtrahend and then adding the two
 517 numbers. Due to the nature of two's complement numbers this method will work for both signed and
 518 unsigned numbers!

► Fix Me:

This section needs more examples of subtracting signed and unsigned numbers and a discussion on how signedness is not relevant until the results are interpreted. For example adding $-4 + -8 = -12$ using two 8-bit numbers is the same as adding $252 + 248 = 500$ and truncating the result to 244.

519 Observation: Since we always have a carry-in of zero into the LSB when adding, we can take advantage
 520 of that fact by (ab)using that carry input to perform that adding the extra 1 to the subtrahend as
 521 part of changing its sign in the examples below.

522 An example showing the subtraction of two *signed* binary numbers: $-4 - 8 = -12$

```

523      -128 64 32 16 8 4 2 1
524      1 1 1 1 1 1 0 0 <== -4 (minuend)
525      - 0 0 0 0 1 0 0 0 <== 8 (subtrahend)
526      -----
527
528      1 1 1 1 1 1 1 1 <== carries
529      1 1 1 1 1 1 0 0 <== -4
530      + 1 1 1 1 0 1 1 1 <== one's complement of 8
531      -----
532
533      1 1 1 1 1 0 1 0 0 <== -12

```

534 2.2.5 Truncation

535 Discarding the carry bit that can be generated from the MSB is called *truncation*.

536 So far we have been ignoring the carries that can come from the MSBs when adding and subtracting.
 537 We have also been ignoring the potential impact of a carry causing a signed number to change its sign
 538 in an unexpected way.

539 In the examples above, truncating the results either had 1) no impact on the calculated sums or 2)
 540 was absolutely necessary to correct the sum in cases such as: $-4 + 5$.

541 For example, note what happens when we try to subtract 1 from the most negative value that we can
 542 represent in a 4 bit two's complement number:

```
543      -8 4 2 1
 544      1 0 0 0 <== -8 (minuend)
 545      - 0 0 0 1 <== 1 (subtrahend)
 546      -----
 547
 548      1           1 <== carries
 549      1 0 0 0 <== -8
 550      + 1 1 1 0 <== one's complement of 1
 551      -----
 552      1 0 1 1 1 <== this SHOULD be -9 but with truncation it is 7
```

554 The problem with this example is that we can not represent -9_{10} using a 4-bit two's complement
 555 number.

556 Granted, if we would have used 5 bit numbers, then the “answer” would have fit OK. But the same
 557 problem would return when trying to calculate $-16 - 1$. So simply “making more room” does not
 558 solve this problem.

559 This is not just a problem when subtracting, nor is it just a problem with signed numbers.

560 The same situation can happen *unsigned* numbers. For example:

```
561      8 4 2 1
 562      1 1 1 0 0 <== carries
 563      1 1 1 0 <== 14 (addend)
 564      + 0 0 1 1 <== 3 (addend)
 565      -----
 566      1 0 0 0 1 <== this SHOULD be 17 but with truncation it is 1
```

567 How to handle such a truncation depends on whether the *original* values being added are signed or
 568 unsigned.

569 The RV ISA refers to the discarding the carry out of the MSB after an add (or subtract) of two
 570 *unsigned* numbers as an *unsigned overflow*⁴ and the situation where carries create an incorrect sign in
 571 the result of adding (or subtracting) two *signed* numbers as a *signed overflow*. [1, p. 13]

572 2.2.5.1 Unsigned Overflow

573 When adding *unsigned* numbers, an overflow only occurs when there is a carry out of the MSB resulting
 574 in a sum that is truncated to fit into the number of bits allocated to contain the result.

⁴Most microprocessors refer to *unsigned overflow* simply as a *carry* condition.

575 Figure 2.2 illustrates an unsigned overflow during addition:

```

1 1 1 1 0 0 0 0 0 <== carries
1 1 1 1 0 0 0 0 <== 240
+ 0 0 0 1 0 0 0 1 <== 17
-----
1 0 0 0 0 0 0 0 1 <== sum = 1

```

Figure 2.2: $240 + 17 = 1$ (overflow)

576 Some times an overflow like this is referred to as a *wrap around* because of the way that successive
 577 additions will result in a value that increases until it *wraps* back *around* to zero and then returns to
 578 increasing in value until it, again, wraps around again.

When adding, *unsigned overflow* occurs when ever there is a carry *out of* the most significant bit.

579 When subtracting *unsigned* numbers, an overflow only occurs when the subtrahend is greater than
 580 the minuend (because in those cases the different would have to be negative and there are no negative
 581 values that can be represented with an unsigned binary number.)

583 Figure 2.3 illustrates an unsigned overflow during subtraction:

```

0 0 0 0 0 0 1 1 <== 3 (minuend)
- 0 0 0 0 0 1 0 0 <== 4 (subtrahend)
-----
0 0 0 0 0 0 1 1 <== carries
0 0 0 0 0 0 1 1 <== 3
+ 1 1 1 1 1 0 1 1 <== one's complement of 4
-----
1 1 1 1 1 1 1 1 <== 255 (overflow)

```

Figure 2.3: $3 - 4 = 255$ (overflow)

When subtracting, *unsigned overflow* occurs when ever there is *not* a carry *out of* the most significant bit (IFF the carry-in on the LSB is used to add the extra 1 to the subtrahend when changing its sign.)

584

585 2.2.5.2 Signed Overflow

586 When adding *signed* numbers, an overflow only occurs when the two addends are positive and sum is
 587 negative or the addends are both negative and the sum is positive.

588 When subtracting *signed* numbers, an overflow only occurs when the minuend is positive and the
 589 subtrahend is negative and difference is negative or when the minuend is negative and the subtrahend
 590 is positive and the difference is positive.⁵

⁵I had to look it up to remember which were which too... it is: minuend - subtrahend = difference.[13]

591 Consider the results of the addition of two *signed* numbers while looking more closely at the carry
 592 values.

```

 0 1 0 0 0 0 0 0 <== carries
 0 1 0 0 0 0 0 0 <== 64
 + 0 1 0 0 0 0 0 0 <== 64
 -----
 1 0 0 0 0 0 0 0 <== sum = -128
  
```

Figure 2.4: $64 + 64 = -128$ (overflow)

593 Figure 2.4 is an example of *signed overflow*. As shown, the problem is that the sum of two positive
 594 numbers has resulted in an obviously incorrect negative result due to a carry flowing into the sign-bit
 595 in the MSB.

596 Granted, if the same values were added using values larger than 8-bits then the sum would have been
 597 correct. However, these examples assume that all the operations are performed on (and results stored
 598 into) 8-bit values. Given any finite-number of bits, there are values that could be added such that an
 599 overflow occurs.

600 Figure 2.5 shows another overflow situation that is caused by the fact that there is nowhere for the
 601 carry out of the sign-bit to go. We say that this result has been *truncated*.

```

 1 0 0 0 0 0 0 0 <== carries
 1 0 0 0 0 0 0 0 <== -128
 + 1 0 0 0 0 0 0 0 <== -128
 -----
 0 0 0 0 0 0 0 0 <== sum = 0
  
```

Figure 2.5: $-128 + -128 = 0$ (overflow)

602 Truncation is not necessarily a problem. Consider the truncations in figures 2.6 and 2.7. Figure 2.7
 603 demonstrates the importance of discarding the carry from the sum of the MSBs of signed numbers
 604 when addends do not have the same sign.

```

 1 1 1 1 1 1 1 0 <== carries
 1 1 1 1 1 1 0 1 <== -3
 + 1 1 1 1 1 0 1 1 <== -5
 -----
 1 1 1 1 1 0 0 0 <== sum = -8
  
```

Figure 2.6: $-3 + -5 = -8$

```

 1 1 1 1 1 1 0 0 <== carries
 1 1 1 1 1 1 1 0 <== -2
 + 0 0 0 0 1 0 1 0 <== 10
 -----
 0 0 0 0 1 0 0 0 <== sum = 8
  
```

Figure 2.7: $-2 + 10 = 8$

605 Just like an unsigned number can wrap around as a result of successive additions, a signed number
 606 can do the same thing. The only difference is that signed numbers won't wrap from the maximum

607 value back to zero, instead it will wrap from the most positive to the most negative value as shown
 608 in [Figure 2.8](#).

```

 0 1 1 1 1 1 1 1 0 <== carries
 0 1 1 1 1 1 1 1 <== 127
 +
 0 0 0 0 0 0 0 1 <== 1
 -----
 1 0 0 0 0 0 0 0 <== sum = -128
  
```

Figure 2.8: $127 + 1 = -128$

609 Formally, a *signed overflow* occurs when ever the carry *into* the most significant bit is not the
 same as the carry *out of* the most significant bit.

610

2.3 Sign and Zero Extension

611 Due to the nature of the two's complement encoding scheme, the following numbers all represent the
 612 same value:

```

 1111 <== -1
 11111111 <== -1
 1111111111111111 <== -1
 111111111111111111111111 <== -1
  
```

613 As do these:

```

 01100 <== 12
 0000001100 <== 12
 000000000000000000000000000000001100 <== 12
  
```

614 The lengthening of these numbers by replicating the digits on the left is what is called *sign extension*.

615 Any signed number can have any quantity of additional MSBs added to it, provided that they
 repeat the value of the sign bit.

616

617 [Figure 2.9](#) illustrates extending the negative sign bit to the left by replicating it. A negative number
 618 will have its **MSB** (bit 19 in this example) set to 1. Extending this value to the left will set all the
 619 new bits to the left of it to 1 as well.

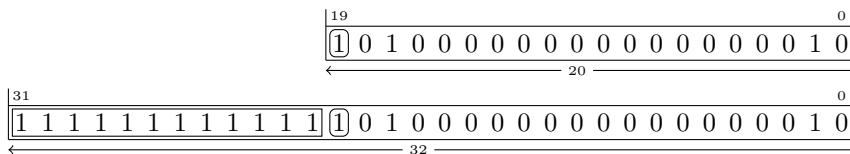


Figure 2.9: Sign-extending a negative integer from 20 bits to 32 bits.

Figure 2.10 illustrates extending the sign bit of a positive number to the left by replicating it. A positive number will have its MSB set to 0. Extending this value to the left will set all the new bits to the left of it to 0 as well.

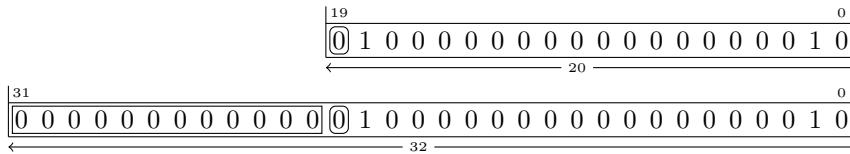


Figure 2.10: Sign-extending a positive integer from 20 bits to 32 bits.

In a similar vein, any unsigned number also may have any quantity of additional MSBs added to it provided that they are all zero. This is called *zero extension*. For example, the following all represent the same value:

Any *unsigned* number may be zero extended to any size.

Figure 2.11 illustrates zero-extending a 20-bit number to the left to form a 32-bit number.

► Fix Me:

Remove the sign-bit boxes from this figure?

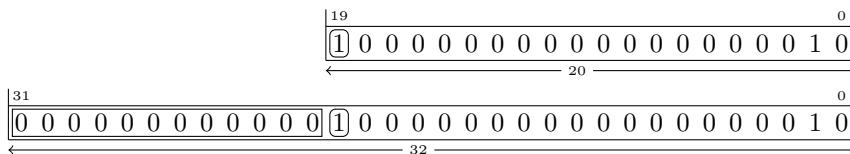


Figure 2.11: Zero-extending an unsigned integer from 20 bits to 32 bits.

2.4 Shifting

We were all taught how to multiply and divide decimal numbers by ten by moving (or *shifting*) the decimal point to the right or left respectively. Doing the same in any other base has the same effect in that it will multiply or divide the number by its base.

641 Multiplication and division are only two reasons for shifting. There can be other occasions where
642 doing so is useful.

► Fix Me:
Include decimal values in the shift diagrams.

As implemented by a CPU, shifting applies to the value in a register and the results stored back into a register of finite size. Therefore a shift result will always be truncated to fit into a register.

645 Note that when dealing with numeric values, any truncation performed during a right-shift will man-
646 ifest itself as rounding toward zero.

► Fix Me:

647 2.4.1 Logical Shifting

648 Shifting *logically* to the left or right is a matter of re-aligning the bits in a register and truncating the
649 result.

650 To shift left two positions:

651

	0
	0

652

653 To shift right one position:

654

	0
	0

655

656 Note that the vacated bit positions are always filled with zero.

657 2.4.2 Arithmetic Shifting

658 Some times it is desirable to retain the value of the sign bit when shifting. The RISC-V ISA provides
659 an arithmetic right shift instruction for this purpose (there is no arithmetic left shift for this ISA.)

660 When shifting to the right *arithmetically*, vacated bit positions are filled by replicating the
value of the sign bit.

661 An arithmetic right shift of a negative number by 4 bit positions:

662

	0
	0

663

664 2.5 Main Memory Storage

665 As mentioned in [section 1.1.1.1](#), the main memory in a RISC-V system is byte-addressable. For that
666 reason we will visualize it by displaying ranges of bytes displayed in hex and in [ASCII](#). As will become
667 obvious, the ASCII part makes it easier to find text messages.⁶

⁶Most of the memory dumps in this text are generated by `rvddt` and are shown on a per-byte basis without any attempt to reorder their values. Some other applications used to dump memory do not dump the bytes in address-order! It is important to know how your software tools operate when using them to dump the contents of memory and/or files.

► Fix Me:

Redraw these with arrows tracking the shifted bits and the truncated values

668

2.5.1 Memory Dump

669 Listing 2.1 shows a *memory dump* from the rvddt ‘d’ command requesting a dump starting at address
 670 0x00002600 for the default quantity (0x100) of bytes.

Listing 2.1: rvddt_memdump.out

rvddt memory dump

```
671 1 ddt> d 0x00002600
672 2 00002600: 93 05 00 00 13 06 00 00 93 06 00 00 13 07 00 00 *.....
673 3 00002610: 93 07 00 00 93 08 d0 05 73 00 00 00 63 54 05 02 *.....
674 4 00002620: 13 01 01 ff 23 24 81 00 13 04 05 00 23 26 11 00 *$.....
675 5 00002630: 33 04 80 40 97 00 00 00 e7 80 40 01 23 20 85 00 *3...@.....
676 6 00002640: 6f 00 00 00 6f 00 00 00 b7 87 00 00 03 a5 07 43 *o.....
677 7 00002650: 67 80 00 00 00 00 00 00 76 61 6c 3d 00 00 00 00 *g.....
678 8 00002660: 00 00 00 00 80 84 2e 41 1f 85 45 41 80 40 9a 44 *.....
679 9 00002670: 4f 11 f3 c3 6e 8a 67 41 20 1b 00 00 20 1b 00 00 *0.....
680 10 00002680: 44 1b 00 00 14 1b 00 00 14 1b 00 00 04 1c 00 00 *D.....
681 11 00002690: 44 1b 00 00 14 1b 00 00 04 1c 00 00 14 1b 00 00 *D.....
682 12 000026a0: 44 1b 00 00 10 1b 00 00 10 1b 00 00 10 1b 00 00 *D.....
683 13 000026b0: 04 1c 00 00 54 1f 00 00 54 1f 00 00 d4 1f 00 00 *.....
684 14 000026c0: 4c 1f 00 00 4c 1f 00 00 34 20 00 00 d4 1f 00 00 *L.....
685 15 000026d0: 4c 1f 00 00 34 20 00 00 4c 1f 00 00 d4 1f 00 00 *L.....
686 16 000026e0: 48 1f 00 00 48 1f 00 00 48 1f 00 00 34 20 00 00 *H.....
687 17 000026f0: 00 01 02 02 03 03 03 03 04 04 04 04 04 04 04 04 *.....
```

690 ℓ 1 The rvddt prompt showing the dump command.

691 ℓ 2 From left to right, the dump is presented as the address of the first byte (0x00002600) followed
 692 by a colon, the value of the byte at address 0x00002600 expressed in hex, the next byte (at
 693 address 0x00002601) and so on for 16 bytes. There is a double-space between the 7th and 8th
 694 bytes to help provide a visual reference for the center to make it easy to locate bytes on the right
 695 end. For example, the byte at address 0x0000260c is four bytes to the right of byte number
 696 eight (at the gap) and contains 0x13. To the right of the 16-bytes is an asterisk-enclosed set of
 697 16 columns showing the ASCII characters that each byte represents. If a byte has a value that
 698 corresponds to a printable character code, the character will be displayed. For any illegal/un-
 699 displayable byte values, a dot is shown to make it easier to count the columns.

700 ℓ 3-17 More of the same as seen on ℓ 2. The address at the left can be seen to advance by 16_{10} (or
 701 10_{16}) for each line shown.

702

2.5.2 Endianness

703 The choice of which end of a multi-byte value is to be stored at the lowest byte address is referred to as
 704 *endianness*. For example, if a CPU were to store a [halfword](#) into memory, should the byte containing
 705 the [Most Significant Bit \(MSB\)](#) (the *big* end) go first or does the byte with the [Least Significant Bit \(LSB\)](#) (the *little* end) go first?

707 On the one hand the choice is arbitrary. On the other hand, it is possible that the choice could impact
 708 the performance of the system.⁷

709 IBM mainframe CPUs and the 68000 family store their bytes in big-endian order. While the Intel
 710 Pentium and most embedded processors use little-endian order. Some CPUs are even *bi-endian* in
 711 that they have instructions that can change their order on the fly.

712 The RISC-V system uses the little-endian byte order.

⁷See [14] for some history of the big/little-endian “controversy.”

713 **2.5.2.1 Big-Endian**714 Using the contents of [Listing 2.1](#), a big-endian CPU would interpret the contents as follows:

- 715 • The 8-bit value read from address `0x00002658` would be `0x76`.
- 716 • The 8-bit value read from address `0x00002659` would be `0x61`.
- 717 • The 8-bit value read from address `0x0000265a` would be `0x6c`.
- 718 • The 8-bit value read from address `0x0000265b` would be `0x3d`.
- 719 • The 16-bit value read from address `0x00002658` would be `0x7661`.
- 720 • The 16-bit value read from address `0x0000265a` would be `0x6c3d`.
- 721 • The 32-bit value read from address `0x00002658` would be `0x76616c3d`.

722 Notice that in a big-endian system, the *place values* of the bits comprising the `0x76` (located at memory
 723 address `0x00002658`) are *different* depending on the number of bytes representing the value that is
 724 being read.

725 For example, when a 16-bit value is read from `0x00002658` then the `76` represents the binary place
 726 values: 2^{15} to 2^8 . When a 32-bit value is read then the `76` represents the binary place values: 2^{31} to
 727 2^{24} . In other words the value read from the first memory location (with the lowest address), of the
 728 plurality of addresses containing the complete value being read, is always placed on the *left end*, into
 729 the Most Significant Bits. One might dare say that the `76` is placed at the end with the *big* place
 730 values.

731 More examples:

- 732 • An 8-bit value read from address `0x00002624` would be `0x23`.
- 733 • An 8-bit value read from address `0x00002625` would be `0x24`.
- 734 • An 8-bit value read from address `0x00002626` would be `0x81`.
- 735 • An 8-bit value read from address `0x00002627` would be `0x00`.
- 736 • A 16-bit value read from address `0x00002624` would be `0x2324`.
- 737 • A 16-bit value read from address `0x00002626` would be `0x8100`.
- 738 • A 32-bit value read from address `0x00002624` would be `0x23248100`.

739 Again, notice that the byte from memory address `0x00002624`, regardless of the *number* of bytes
 740 comprising the complete value being fetched, will always appear on the *left/big* end of the final value.

On a big-endian system, the bytes in the dump are in the same order as they would be used
 by the CPU if it were to read them as a multi-byte value.

741

2.5.2.2 Little-Endian

Using the contents of [Listing 2.1](#), a little-endian CPU would interpret the contents as follows:

- An 8-bit value read from address `0x00002658` would be `0x76`.
- An 8-bit value read from address `0x00002659` would be `0x61`.
- An 8-bit value read from address `0x0000265a` would be `0x6c`.
- An 8-bit value read from address `0x0000265b` would be `0x3d`.
- A 16-bit value read from address `0x00002658` would be `0x6176`.
- A 16-bit value read from address `0x0000265a` would be `0x3d6c`.
- A 32-bit value read from address `0x00002658` would be `0x3d6c6176`.

Notice that in a little-endian system, the *place values* of the bits comprising the `0x76` (located at memory address `0x00002658`) are the *same* regardless of the the number of bytes representing the value that is being read.

Unlike the behavior of a big-endian machine, when little-endian machine reads a 16-bit value from `0x00002658` the `76` represents the binary place values from 2^7 to 2^0 . When a 32-bit value is read then the `76` (still) represents the binary place values from 2^7 to 2^0 . In other words the value read from the first memory location (with the lowest address), of the plurality of addresses containing the complete value being read, is always placed on the *right end*, into the Least Significant Bits. One might say that the `76` is placed at the end with the *little* place values.

Also notice that it is the *bytes* are what are “reversed” in a little-endian system (*not* the hex digits.)

More examples:

- The 8-bit value read from address `0x00002624` would be `0x23`.
- The 8-bit value read from address `0x00002625` would be `0x24`.
- The 8-bit value read from address `0x00002626` would be `0x81`.
- The 8-bit value read from address `0x00002627` would be `0x00`.
- The 16-bit value read from address `0x00002624` would be `0x2423`.
- The 16-bit value read from address `0x00002626` would be `0x0081`.
- The 32-bit value read from address `0x00002624` would be `0x00812423`.

As above, notice that the byte from memory address `0x00002624`, regardless of the *number* of bytes comprising the complete value being fetched, will always appear on the *right/little* end of the final value.

On a little-endian system, the bytes in the dump are in reverse order as they would be used by the CPU if it were to read them as a multi-byte value.

In the RISC-V ISA it is noted that

774 A minor point is that we have also found little-endian memory systems to be more natural
 775 for hardware designers. However, certain application areas, such as IP networking, operate
 776 on big-endian data structures, and so we leave open the possibility of non-standard big-
 777 endian or bi-endian systems.”¹ [1, p. 6]

778 2.5.3 Arrays and Character Strings

779 While Endianness defines how single values are stored in memory, the *array* defines how multiple
 780 values are stored.

781 An array is a data structure comprised of an ordered set of elements. This text will limit its definition
 782 of array to a plurality of elements that are all of the same type. Where type refers to the size (number
 783 of bytes) and representation (signed, unsigned,...) of each element.

784 In an array, the elements are stored adjacent to one another such that the address e of any element
 785 $x[n]$ is:

$$e = a + n * s \quad (2.5.1)$$

786 Where x is the name of the array, n is the element number of interest, e is the address of interest, a
 787 is the address of the first element in the array and s is the size (in bytes) of each element.

788 Given an array x containing m elements, $x[0]$ is the first element of the array and $x[m - 1]$ is the last
 789 element of the array.⁸

790 Using this definition, and the memory dump shown in Listing 2.1, and the knowledge that we are
 791 using a little-endian machine and given that $a = 0x00002656$ and $s = 2$, the values of the first 8
 792 elements of array x are:

- 793 • $x[0]$ is $0x0000$ and is stored at $0x00002656$.
- 794 • $x[1]$ is $0x6176$ and is stored at $0x00002658$.
- 795 • $x[2]$ is $0x3d6c$ and is stored at $0x0000265a$.
- 796 • $x[3]$ is $0x0000$ and is stored at $0x0000265c$.
- 797 • $x[4]$ is $0x0000$ and is stored at $0x00002660$.
- 798 • $x[5]$ is $0x0000$ and is stored at $0x00002662$.
- 799 • $x[6]$ is $0x8480$ and is stored at $0x00002664$.
- 800 • $x[7]$ is $0x412e$ and is stored at $0x00002666$.

In general, there is no fixed rule nor notion as to how many elements an array has. It is up to the programmer to ensure that the starting address and the number of elements in any given array (its size) are used properly so that data bytes outside an array are not accidentally used as elements.

801 ⁸Some computing languages (C, C++, Java, C#, Python, Perl,...) define an array such that the first element is indexed as $x[0]$. While others (FORTRAN, MATLAB) define the first element of an array to be $x[1]$.

802 There is, however, a common convention used for an array of characters that is used to hold a text
 803 message (called a *character string* or just *string*).

804 When an array is used to hold a string the element past the last character in the string is set to zero.
 805 This is because 1) zero is not a valid printable ASCII character and 2) it simplifies software in that
 806 knowing no more than the starting address of a string is all that is needed to process it. Without
 807 this zero *sentinel* value (called a *null terminator*), some knowledge of the number of characters in the
 808 string would have to otherwise be conveyed to any code needing to consume or process the string.

809 In [Listing 2.1](#), the 5-byte long array starting at address 0x00002658 contains a string whose value can
 810 be expressed as either:

811 76 61 6c 3d 00

812 or

813 "val"

814 When the double-quoted text form is used, the GNU assembler used in this text differentiates between
 815 *ascii* and *asciiz* strings such that an *ascii* string is **not** null terminated and an *asciiz* string **is** null
 816 terminated.

817 The value of providing a method to create a string that is not null terminated is that a program may
 818 define a large string by concatenating a number of *ascii* strings together and following the last with
 819 a byte of zero to null-terminate it.

820 It is a common mistake to create a string with a missing null terminator. The result of printing such
 821 a string is that the string will be printed as well as whatever random data bytes in memory follow it
 822 until a byte whose value is zero is encountered by chance.

823 2.5.4 Context is Important!

824 Data values can be interpreted differently depending on the context in which they are used. Assuming
 825 what a set of bytes is used for based on their contents can be very misleading! For example, there is
 826 a 0x76 at address 0x00002658. This is a 'v' if you use it as an ASCII (see [Appendix C](#)) character, a
 827 118₁₀ if it is an integer value and TRUE if it is a conditional.

828 2.5.5 Alignment

829 With respect to memory and storage, *alignment* refers to the *location* of a data element when the
 830 address that it is stored is a precise multiple of a power-of-2.

► Fix Me:

Include the obligatory diagram showing the overlapping data types when they are all aligned.

831 The primary alignments of concern are typically 2 (a halfword), 4 (a fullword), 8 (a double word) and
 832 16 (a quad-word) bytes.

833 For example, any data element that is aligned to 2-byte boundary must have an (hex) address that
 834 ends in any of: 0, 2, 4, 6, 8, A, C or E. Any 4-byte aligned element must be located at an address
 835 ending in 0, 4, 8 or C. An 8-byte aligned element at an address ending with 0 or 8, and 16-byte aligned
 836 elements must be located at addresses ending in zero.

837 Such alignments are important when exchanging data between the CPU and memory because the
 838 hardware implementations are optimized to transfer aligned data. Therefore, aligning data used by

839 any program will reap the benefit of running faster.⁹

840 An element of data is considered to be *aligned to its natural size* when its address is an exact multiple
841 of the number of bytes used to represent the data. Note that the ISA we are concerned with *only*
842 operates on elements that have sizes that are powers of two.

843 For example, a 32-bit integer consumes one full word. If the four bytes are stored in main memory at
844 an address than is a multiple of 4 then the integer is considered to naturally aligned.

845 The same would apply to 16-bit, 64-bit, 128-bit and other such values as they fit into 2, 8 and 16 byte
846 elements respectively.

847 Some CPUs can deliver four (or more) bytes at the same time while others might only be capable
848 of delivering one or two bytes at a time. Such differences in hardware typically impact the cost and
849 performance of a system.¹⁰

850 2.5.6 Instruction Alignment

851 The RISC-V ISA requires that all instructions be aligned to their natural boundaries.

852 Every possible instruction that an RV32I CPU can execute contains exactly 32 bits. Therefore they
853 are always stored on a full word boundary. Any *unaligned* instruction is *illegal*.¹¹

854 An attempt to fetch an instruction from an unaligned address will result in an error referred to as
855 an alignment *exception*. This and other exceptions cause the CPU to stop executing the current
856 instruction and start executing a different set of instructions that are prepared to handle the problem.
857 Often an exception is handled by completely stopping the program in a way that is commonly referred
858 to as a system or application *crash*.

⁹ Alignment of data, while important for efficient performance, is not mandatory for RISC-V systems.[1, p. 19]

¹⁰ The design and implementation choices that determine how any given system operates are part of what is called a system's *organization* and is beyond the scope of this text. See [3] for more information on computer organization.

¹¹ This rule is relaxed by the C extension to allow an instruction to start at any even address.[1, p. 5]

Chapter 3

The Elements of a Assembly Language Program

3.1 Assembly Language Statements

Introduce the assembly language grammar.

- Statement = 1 line of text containing an instruction or directive.
- Instruction = label, mnemonic, operands, comment.
- Directive = Used to control the operation of the assembler.

3.2 Memory Layout

Is this a good place to introduce the text, data, bss, heap and stack regions?

Or does that belong in a new section/chapter that discusses addressing modes?

3.3 A Sample Program Source Listing

A simple program that illustrates how this text presents program source code is seen in [Listing 3.1](#).
This program will place a zero in each of the 4 registers named x28, x29, x30 and x31.

Listing 3.1: `zero4regs.S`

Setting four registers to zero.

```

873
874 1      .text                      # put this into the text section
875 2      .align 2                  # align to 2^2
876 3      .globl _start
877 4      _start:
878 5      addi    x28, x0, 0      # set register x28 to zero
879 6      addi    x29, x0, 0      # set register x29 to zero
880 7      addi    x30, x0, 0      # set register x30 to zero
881 8      addi    x31, x0, 0      # set register x31 to zero

```

883 This program listing illustrates a number of things:

- 884 • Listings are identified by the name of the file within which they are stored. This listing is from
885 a file named: `zero4regs.S`.
- 886 • The assembly language programs discussed in this text will be saved in files that end with: `.S`
887 (Alternately you can use `.sx` on systems that don't understand the difference between upper
888 and lowercase letters.¹)
- 889 • A description of the listing's purpose appears under the name of the file. The description of
890 [Listing 3.1](#) is *Setting four registers to zero*.
- 891 • The lines of the listing are numbered on the left margin for easy reference.
- 892 • An assembly program consists of lines of plain text.
- 893 • The RISC-V ISA does not provide an operation that will simply set a register to a numeric
894 value. To accomplish our goal this program will add zero to zero and place the sum in in each
895 of the four registers.
- 896 • The lines that start with a dot '.' (on lines 1, 2 and 3) are called *assembler directives* as they
897 tell the assembler itself how we want it to translate the following *assembly language instructions*
898 into *machine language instructions*.
- 899 • Line 4 shows a *label* named `_start`. The colon at the end is the indicator to the assembler that
900 causes it to recognize the preceding characters as a label.
- 901 • Lines 5-8 are the four assembly language instructions that make up the program. Each instruc-
902 tion in this program consists of four *fields*. (Different instructions can have a different number
903 of fields.) The fields on line 5 are:

904 addi The instruction mnemonic. It indicates the operation that the CPU will perform.
905 x28 The *destination* register that will receive the sum when the *addi* instruction is finished.
906 The names of the 32 registers are $x0 - x31$.
907 x0 One of the addends of the sum operation. (The $x0$ register will always contain the value
908 zero. It can never be changed.)
909 0 The second addend is the number zero.
910 # set ... Any text anywhere in a RISC-V assembly language program that starts with the pound-
911 sign is ignored by the assembler. They are used to place a *comment* in the program to help
912 the reader better understand the motive of the programmer.

913 3.4 Running a Program With rvddt

914 To illustrate what a CPU does when it executes instructions this text will use the `rvddt` simulator to
915 display shows sequence of events and the binary values involved. This simulator supports the RV32I
916 ISA and has a configurable amount of memory.²

917 [Listing 3.2](#) shows the operation of the four *addi* instructions from [Listing 3.1](#) when it is executed in
918 trace-mode.

¹The author of this text prefers to avoid using such systems.

²The `rvddt` simulator was written to generate the listings for this text. It is similar to the fancier *spike* simulator.
Given the simplicity of the RV32I ISA, `rvddt` is less than 1700 lines of C++ and was written in one (long) afternoon.

Listing 3.2: zero4regs.out

Running a program with the rvddt simulator

```

919 1 [winans@w510 src]$ ./rvddt -f ../../examples/load4regs.bin
920 2 Loading '../examples/load4regs.bin' to 0x0
921 3 ddt> t4
922 4     x0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
923 5     x8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
924 6     x16: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
925 7     x24: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
926 8     pc: 00000000
927 9 00000000: 00000e13 addi    x28, x0, 0    # x28 = 0x00000000 = 0x00000000 + 0x00000000
928 10    x0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
929 11    x8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
930 12    x16: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
931 13    x24: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 00000000 f0f0f0f0 f0f0f0f0
932 14    pc: 00000004
933 15 00000004: 00000e93 addi    x29, x0, 0    # x29 = 0x00000000 = 0x00000000 + 0x00000000
934 16    x0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
935 17    x8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
936 18    x16: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
937 19    x24: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 00000000 00000000 f0f0f0f0 f0f0f0f0
938 20    pc: 00000008
939 21 00000008: 00000f13 addi    x30, x0, 0    # x30 = 0x00000000 = 0x00000000 + 0x00000000
940 22    x0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
941 23    x8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
942 24    x16: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
943 25    x24: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 00000000 00000000 00000000 f0f0f0f0
944 26    pc: 0000000c
945 27 0000000c: 00000f93 addi    x31, x0, 0    # x31 = 0x00000000 = 0x00000000 + 0x00000000
946 28 ddt> r
947 29     x0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
948 30     x8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
949 31     x16: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
950 32     x24: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 00000000 00000000 00000000 00000000
951 33     pc: 00000010
952 34 ddt> x
953 35 [winans@w510 src]$
```

ℓ 1 This listing includes the command-line that shows how the simulator was executed to load a file containing the machine instructions (aka machine code) from the assembler.

ℓ 2 A message from the simulator indicating that it loaded the machine code into simulated memory at address 0.

ℓ 3 This line shows the prompt from the debugger and the command `t4` that the user entered to request that the simulator trace the execution of four instructions.

ℓ 4-8 Prior to executing the first instruction, the state of the CPU registers is displayed.

ℓ 4 The values in registers 0, 1, 2, 3, 4, 5, 6 and 7 are printed from left to right in [big-endian](#), [hexadecimal](#) form. The double-space gap in the middle of the line is a reference to make it easier to visually navigate across the line without being forced to count the values from the far left when seeking the value of, say, `x5`.

ℓ 5-7 The values of registers 8-31 are printed.

ℓ 8 The *program counter* (`pc`) register is printed. It contains the address of the instruction that the CPU will execute. After each instruction, the `pc` will either advance four bytes ahead or be set to another value by a branch instruction as discussed above.

ℓ 9 A four-byte instruction is fetched from memory at the address in the `pc` register, is decoded and printed. From left to right the fields shown on this line are:

973 00000000 The memory address from which the instruction was fetched. This address is displayed in
974 [big-endian](#), [hexadecimal](#) form.

975 00000e13 The machine code of the instruction displayed in [big-endian](#), [hexadecimal](#) form.

976 addi The mnemonic for the machine instruction.

977 x28 The **rd** field of the addi instruction.

978 x0 The **rs1** field of the addi instruction that holds one of the two addends of the operation.

979 0 The **imm** field of the addi instruction that holds the second of the two addends of the
980 operation.

981 # ... A simulator-generated comment that explains what the instruction is doing. For this in-
982 struction it indicates that **x28** will have the value zero stored into it as a result of performing
983 the addition: 0 + 0.

984 ℓ 10-14 These lines are printed as the prelude while tracing the second instruction. Lines 7 and 13 show
985 that **x28** has changed from f0f0f0f0 to 00000000 as a result of executing the first instruction and
986 lines 8 and 14 show that the **pc** has advanced from zero (the location of the first instruction) to
987 four, where the second instruction will be fetched. None of the rest of the registers have changed
988 values.

989 ℓ 15 The second instruction decoded executed and described. This time register **x29** will be assigned
990 a value.

991 ℓ 16-27 The third and fourth instructions are traced.

992 ℓ 28 Tracing has completed. The simulator prints its prompt and the user enters the ‘r’ command
993 to see the register state after the fourth instruction has completed executing.

994 ℓ 29-33 Following the fourth instruction it can be observed that registers **x28**, **x29**, **x30** and **x31** have
995 been set to zero and that the **pc** has advanced from zero to four, then eight, then 12 (the hex
996 value for 12 is c) and then to 16 (which, in hex, is 10).

997 ℓ 34 The simulator exit command ‘x’ is entered by the user and the terminal displays the shell prompt.

Chapter 4

Writing RISC-V Programs

1000 This chapter introduces each of the RV32I instructions by developing programs that demonstrate their usefulness. ▶ Fix Me:
Introduce the ISA register names and aliases in here?

➡ Fix Me:

Introduce the ISA register names and aliases in here?

4.1 Use ebreak to Stop rvddt Execution

1003 It is a good idea to learn how to stop before learning how to go!

1004 The **ebreak** instruction exists for the sole purpose of transferring control back to a debugging environment.[1,
1005 p. 24]

1006 When `rvddt` executes an `ebreak` instruction, it will immediately terminate any executing `trace` or `go`
1007 command currently executing and return to the command prompt without advancing the `pc` register.

1008 The machine language encoding shows that `ebreak` has no operands.

1009 ebreak

Listing 4.2 demonstrates that since `rvddt` does not advance the `pc` when it encounters an `ebreak` instruction, subsequent `trace` and/or `go` commands will re-execute the same `ebreak` and halt the simulation again (and again). This feature is intended to help prevent overzealous users from accidentally running past the end of a code fragment.¹

Listing 4.1: ebreak/ebreak.S

A one-line `ebreak` program.

```
1015      A one-line ebreak program:  
1016 1      .text          # put this into the text section  
1017 2      .align 2        # align to a multiple of 4  
1018 3      .globl _start  
1019 4  
1020 5 _start:  
1021 6      ebreak
```

¹This was one of the first *enhancements* I needed for myself :-)

Listing 4.2: ebreak/ebreak.out

ebreak stops rvddt without advancing pc.

```

1023 1 $ rvddt -f ebreak.bin
1024 2 sp initialized to top of memory: 0x0000ffff
1025 3 Loading 'ebreak.bin' to 0x0
1026 4 This is rvddt. Enter ? for help.
1027 5 ddt> d 0 16
1028 6 00000000: 73 00 10 00 a5 *s.....*
1029 7 ddt> r
1030 8 x0 00000000 f0f0f0f0 0000ffff f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1031 9 x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1032 10 x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1033 11 x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1034 12 pc 00000000
1035 13 ddt> ti 0 1000
1036 14 00000000: ebreak
1037 15 ddt> ti
1038 16 00000000: ebreak
1039 17 ddt> g 0
1040 18 00000000: ebreak
1041 19 ddt> r
1042 20 x0 00000000 f0f0f0f0 0000ffff f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1043 21 x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1044 22 x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1045 23 x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1046 24 pc 00000000
1047 25 ddt> x

```

4.2 Using the addi Instruction

The detailed description of how the **addi** instruction is executed is that it:

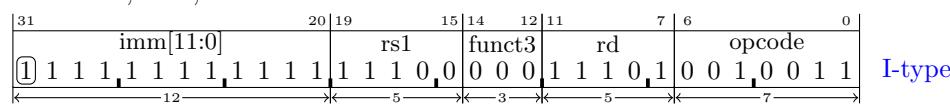
► Fix Me:

Define what constant and immediate values are somewhere.

1. Sign-extends the immediate operand.
2. Add the sign-extended immediate operand to the contents of the **rs1** register.
3. Store the sum in the **rd** register.
4. Add four to the **pc** register (point to the next instruction.)

In the following example **rs1** = **x28**, **rd** = **x29** and the immediate operand is -1.

addi x29, x28, -1



Depending on the values of the fields in this instruction a number of different operations can be performed. The most obvious is that it can add things. But it can also be used to copy registers, set a register to zero and even, when you need to, accomplish nothing.

4.2.1 No Operation

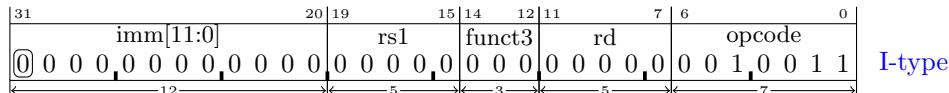
It might seem odd but it is sometimes important to be able to execute an instruction that accomplishes nothing while simply advancing the **pc** to the next instruction. One reason for this is to fill unused

1065 memory between two instructions in a program.²

1066 An instruction that accomplishes nothing is called a **nop** (sometimes systems call these **noop**). The
1067 name means *no operation*. The intent of a **nop** is to execute without having any side effects other
1068 than to advance the **pc** register.

1069 The **addi** instruction can serve as a **nop** by coding it like this:

1070 **addi x0, x0, 0**



1072 The result will be to add zero to zero and discard the result (because you can never store a value into
1073 the **x0** register.)

1074 The RISC-V assembler provides a pseudoinstruction specifically for this purpose that you can use
1075 to improve the readability of your code. Note that the **addi** and **nop** instructions in [Listing 4.3](#) are
1076 assembled into the exact same binary machine instructions as can be seen by comparing it to **objdump**
1077 [Listing 4.4](#), and **rvddt** [Listing 4.5](#) output.

Listing 4.3: nop/nop.S

Demonstrate that **addi** can be used as a **nop**.

```
1078
1079 1 .text          # put this into the text section
1080 2 .align 2        # align to a multiple of 4
1081 3 .globl _start
1082 4
1083 5 _start:
1084 6     addi    x0, x0, 0    # these two instructions assemble into the same thing!
1085 7     nop
1086 8
1087 9     ebreak
```

Listing 4.4: nop/nop.lst

Using **addi** to perform a **nop**

```
1088
1089 1 nop:      file format elf32-littleriscv
1090 2 Disassembly of section .text:
1091 3 00000000 <_start>:
1092 4     0: 00000013          nop
1093 5     4: 00000013          nop
1094 6     8: 00100073          ebreak
```

Listing 4.5: nop/nop.out

Using **addi** to perform a **nop**

```
1095
1096 1 $ rvddt -f nop.bin
1097 2 sp initialized to top of memory: 0x0000ffff
1098 3 Loading 'nop.bin' to 0x0
1099 4 This is rvddt. Enter ? for help.
1100 5 ddt> d 0 16
1101 6 00000000: 13 00 00 00 13 00 00 00 73 00 10 00 a5 a5 a5 a5 *.....s....*
1102 7 ddt> r
1103 8     x0 00000000 f0f0f0f0 0000ffff f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1104 9     x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1105 10    x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1106 11    x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
```

²This can happen during the evolution of one portion of code that reduces in size but has to continue to fit into a system without altering any other code... or sometimes you just need to waste a small amount of time in a device driver.

```

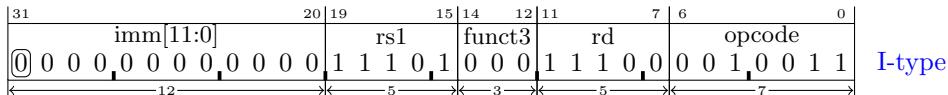
1109 12      pc 00000000
1110 13 ddt> ti 0 1000
1111 14 00000000: 00000013 addi    x0, x0, 0      # x0 = 0x00000000 = 0x00000000 + 0x00000000
1112 15 00000004: 00000013 addi    x0, x0, 0      # x0 = 0x00000000 = 0x00000000 + 0x00000000
1113 16 00000008: ebreak
1114 17 ddt> r
1115 18      x0 00000000 f0f0f0f0 0000ffff f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1116 19      x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1117 20      x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1118 21      x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
1119 22      pc 00000008
1120 23 ddt> x

```

4.2.2 Copying the Contents of One Register to Another

By adding zero to one register and storing the sum in another register the `addi` instruction can be used to copy the value stored in one register to another register. The following instruction will copy the contents of `t4` into `t3`.

addi t3, t4, 0



This is a commonly required operation. To make your intent clear you may use the `mv` pseudoinstruction for this purpose.

[Listing 4.6](#) shows the source of a program that is dumped in [Listing 4.7](#) illustrating that the assembler has generated the same machine instruction (0x000e8e13 at addresses 0x0 and 0x4) for both of the instructions.

Listing 4.6: mv/mv.S
Comparing addi to mv

```

1133 1 .text          # put this into the text section
1134 2 .align 2       # align to a multiple of 4
1135 3 .globl _start
1136 4
1137 5 _start:
1138 6     addi    t3, t4, 0    # t3 = t4
1139 7     mv      t3, t4    # t3 = t4
1140 8
1141 9     ebreak
1142

```

Listing 4.7: mv/mv.lst
An objdump of an addi and mv Instruction.

```

1144 1 mv:      file format elf32-littleriscv
1145 2 Disassembly of section .text:
1146 3 00000000 <_start>:
1147 4     0: 000e8e13          mv  t3,t4
1148 5     4: 000e8e13          mv  t3,t4
1149 6     8: 00100073          ebreak
1150

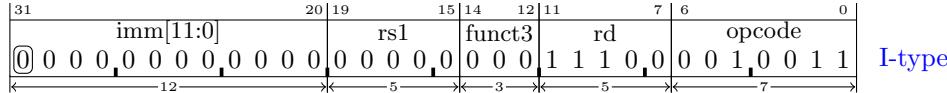
```

4.2.3 Setting a Register to Zero

Recall that `x0` always contains the value zero. Any register can be set to zero by copying the contents of `x0` using `mv` (aka `addi`).³

For example, to set `t3` to zero:

`addi t3, x0, 0`



Listing 4.8: `mvzero/mv.S`

Using `mv` (aka `addi`) to zero-out a register.

```

1158 1 .text          # put this into the text section
1159 2 .align 2      # align to a multiple of 4
1160 3 .globl _start
1161 4
1162 5 _start:
1163 6     mv      t3, x0      # t3 = 0
1164 7
1165 8     ebreak

```

Listing 4.9 traces the execution of the program in Listing 4.8 showing how `t3` is changed from `0xf0f0f0f0` (seen on $\ell 16$) to `0x00000000` (seen on $\ell 26$.)

Listing 4.9: `mvzero/mv.out`

Setting `t3` to zero.

```

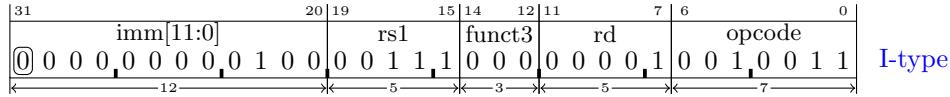
1170 1 $ rvddt -f mv.bin
1171 2 sp initialized to top of memory: 0x0000ffff
1172 3 Loading 'mv.bin' to 0x0
1173 4 This is rvddt. Enter ? for help.
1174 5 ddt> a
1175 6 ddt> d 0 16
1176 7 00000000: 13 0e 00 00 73 00 10 00 a5 a5 a5 a5 a5 a5 a5 a5 *....s.....
1177 8 ddt> t 0 1000
1178 9 zero x0 00000000 ra x1 f0f0f0f0 sp x2 0000ffff gp x3 f0f0f0f0
1179 10 tp x4 f0f0f0f0 to x5 f0f0f0f0 t1 x6 f0f0f0f0 t2 x7 f0f0f0f0
1180 11 s0 x8 f0f0f0f0 s1 x9 f0f0f0f0 a0 x10 f0f0f0f0 a1 x11 f0f0f0f0
1181 12 a2 x12 f0f0f0f0 a3 x13 f0f0f0f0 a4 x14 f0f0f0f0 a5 x15 f0f0f0f0
1182 13 a6 x16 f0f0f0f0 a7 x17 f0f0f0f0 s2 x18 f0f0f0f0 s3 x19 f0f0f0f0
1183 14 s4 x20 f0f0f0f0 s5 x21 f0f0f0f0 s6 x22 f0f0f0f0 s7 x23 f0f0f0f0
1184 15 s8 x24 f0f0f0f0 s9 x25 f0f0f0f0 s10 x26 f0f0f0f0 s11 x27 f0f0f0f0
1185 16 t3 x28 f0f0f0f0 t4 x29 f0f0f0f0 t5 x30 f0f0f0f0 t6 x31 f0f0f0f0
1186 17 pc 00000000
1187 18 00000000: 00000e13 addi t3, zero, 0 # t3 = 0x00000000 = 0x00000000 + 0x00000000
1188 19 zero x0 00000000 ra x1 f0f0f0f0 sp x2 0000ffff gp x3 f0f0f0f0
1189 20 tp x4 f0f0f0f0 to x5 f0f0f0f0 t1 x6 f0f0f0f0 t2 x7 f0f0f0f0
1190 21 s0 x8 f0f0f0f0 s1 x9 f0f0f0f0 a0 x10 f0f0f0f0 a1 x11 f0f0f0f0
1191 22 a2 x12 f0f0f0f0 a3 x13 f0f0f0f0 a4 x14 f0f0f0f0 a5 x15 f0f0f0f0
1192 23 a6 x16 f0f0f0f0 a7 x17 f0f0f0f0 s2 x18 f0f0f0f0 s3 x19 f0f0f0f0
1193 24 s4 x20 f0f0f0f0 s5 x21 f0f0f0f0 s6 x22 f0f0f0f0 s7 x23 f0f0f0f0
1194 25 s8 x24 f0f0f0f0 s9 x25 f0f0f0f0 s10 x26 f0f0f0f0 s11 x27 f0f0f0f0
1195 26 t3 x28 00000000 t4 x29 f0f0f0f0 t5 x30 f0f0f0f0 t6 x31 f0f0f0f0
1196 27 pc 00000004
1197 28 00000004: ebreak
1198 29 ddt> x

```

³There are other pseudoinstructions (such as `li`) that can also turn into an `addi` instruction. Objdump might display '`addi t3,x0,0`' as '`mv t3,x0`' or '`li t3,0`'.

1201 **4.2.4 Adding a 12-bit Signed Value**

1202 addi x1, x7, 4



1203

```

1204      addi    t0, zero, 4      # t0 = 4
1205      addi    t1, t1, 100     # t1 = 104
1206
1207      addi    t0, zero, 0x123    # t0 = 0x123
1208      addi    t0, t0, 0xffff    # t0 = 0x122 (subtract 1)
1209
1210      addi    t0, zero, 0xffff    # t0 = 0xffffffff (-1) (diagram out the chaining carry)
1211                      # refer back to the overflow/truncation discussion in binary chapter
1212
1213      addi x0, x0, 0 # no operation (pseudo: nop)
1214      addi rd, rs, 0 # copy reg rs to rd (pseudo: mv rd, rs)

```

1215 **4.3 todo**

1216 Ideas for the order of introducing instructions.

1217 **4.4 Other Instructions With Immediate Operands**

```

1218      andi
1219      ori
1220      xori
1221
1222      slti
1223      sltiu
1224      srai
1225      slli
1226      srli

```

1227 **4.5 Transferring Data Between Registers and Memory**

1228 RV is a load-store architecture. This means that the only way that the CPU can interact with the
 1229 memory is via the *load* and *store* instructions. All other data manipulation must be performed on
 1230 register values.

1231 Copying values from memory to a register (first examples using regs set with addi):

```

1232      lb
1233      lh
1234      lw
1235      lbu
1236      lhu

```

1237 Copying values from a register to memory:

```
1238    sb
1239    sh
1240    sw
```

1241 **4.6 RR operations**

```
1242    add
1243    sub
1244    and
1245    or
1246    sra
1247    srl
1248    sll
1249    xor
1250    sltu
1251    slt
```

1252 **4.7 Setting registers to large values using lui with addi**

```
1253    addi      // useful for values from -2048 to 2047
1254    lui       // useful for loading any multiple of 0x1000
1255
1256    Setting a register to any other value must be done using a combo of insns:
1257
1258    auipc     // Load an address relative the the current PC (see la pseudo)
1259    addi
1260
1261    lui       // Load constant into into bits 31:12 (see li pseudo)
1262    addi      // add a constant to fill in bits 11:0
1263        if bit 11 is set then need to +1 the lui value to compensate
```

1264 **4.8 Labels and Branching**

1265 Start to introduce addressing here?

```
1266    beq
1267    bne
1268    blt
1269    bge
1270    bltu
1271    bgeu
1272
1273    bgt rs, rt, offset      # pseudo for: blt rt, rs, offset      (reverse the operands)
1274    ble rs, rt, offset      # pseudo for: bge rt, rs, offset      (reverse the operands)
1275    bgtu rs, rt, offset     # pseudo for: bltu rt, rs, offset     (reverse the operands)
1276    bleu rs, rt, offset     # pseudo for: bgeu rt, rs, offset     (reverse the operands)
1277
```

```

1278      beqz rs, offset      # pseudo for: beq rs, x0, offset
1279      bnez rs, offset      # pseudo for: bne rs, x0, offset
1280      blez rs, offset      # pseudo for: bge x0, rs, offset
1281      bgez rs, offset      # pseudo for: bge rs, x0, offset
1282      bltz rs, offset      # pseudo for: blt rs, x0, offset
1283      bgtz rs, offset      # pseudo for: blt x0, rs, offset

```

4.9 Jumps

1285 Introduce and present subroutines but not nesting until introduce stack operations.

```

1286      jal
1287      jalr

```

4.10 Pseudoinstructions

```

1289      li    rd,constant
1290          lui      rd,(constant + 0x00000800) >> 12
1291          addi     rd,rd,(constant & 0x00000fff)
1292
1293      la    rd,label
1294          auipc   rd,((label-.) + 0x00000800) >> 12
1295          addi     rd,rd,((label-(-4)) & 0x00000fff)
1296
1297      l{b|h|w} rd,label
1298          auipc   rd,((label-.) + 0x00000800) >> 12
1299          l{b|h|w} rd,((label-(-4)) & 0x00000fff)(rd)
1300
1301      s{b|h|w} rd,label,rt      # rt used as a temp reg for the operation (default=x6)
1302          auipc   rt,((label-.) + 0x00000800) >> 12
1303          s{b|h|w} rd,((label-(-4)) & 0x00000fff)(rt)
1304
1305      call label      auipc   x1,((label-.) + 0x00000800) >> 12
1306          jalr     x1,((label-(-4)) & 0x00000fff)(x1)
1307
1308      tail label,rt      # rt used as a temp reg for the operation (default=x6)
1309          auipc   rt,((label-.) + 0x00000800) >> 12
1310          jalr     x0,((label-(-4)) & 0x00000fff)(rt)
1311
1312      mv    rd,rs      addi     rd,rs,0
1313
1314      j     label      jal      x0,label
1315      jal   label      jal      x1,label
1316      jr   rs        jalr     x0,0(rs)
1317      jalr rs        jalr     x1,0(rs)
1318      ret            jalr     x0,0(x1)

```

4.10.1 The li Pseudoinstruction

1320 Note that the `li` pseudoinstruction includes an (effectively) conditional addition of 1 to the immediate
1321 operand in the `lui` instruction. This is because the immediate operand in the `addi` instruction is sign-

1322 extended before it is added to `rd`. If the immediate operand to the `addi` has its most-significant-bit
 1323 set to 1 then it will have the effect of subtracting 1 from the operand in the `lui` instruction.

1324 Consider the case of putting the value `0x12345800` into register `x5`:

1325 `li x5,0x12345800`

1326 A naive (incorrect) solution might be:

1327 `lui x5,0x12345 // x5 = 0x12345000`
 1328 `addi x5,x5,0x800 // x5 = 0x12345000 + sx(0x800) = 0x12345000 + 0xfffff800 = 0x12344800`

1329 The result of the above code is that an incorrect value has been placed into `x5`.

1330 To remedy this problem, the value used in the `lui` instruction can be altered (by adding 1 to its
 1331 operand) to compensate for the sign-extention in the `addi` instruction:

1332 `lui x5,0x12346 // x5 = 0x12346000 (note: this is 0x12345800 + 0x0800)`
 1333 `addi x5,x5,0x800 // x5 = 0x12346000 + sx(0x800) = 0x12346000 + 0xfffff800 = 0x12345800`

1334 Keep in mind that the `li` pseudoinstruction must *only* increment the operand of the `lui` instruction
 1335 when it is known that the operand of the subsequent `addi` instruction will be a negative number.

1336 By adding `0x00000800` to the immediate operand of the `lui` instruction in this example, a carry-
 1337 bit into bit-12 will be set to 1 iff the value in bits 11-0 will be treated as a negative value in the
 1338 subsequent `addi` instruction. In other words, when bit-11 is set to 1 in the immediate operand of the
 1339 `li` pseudoinstruction, the immediate operand of the `lui` instruction will be incremented by 1.

► Fix Me:
 Add a ribbon diagram of
 this?

1340 Consider the case where we wish to put the value `0x12345700` into register `x5`:

1341 `lui x5,0x12345 // x5 = 0x12345000 (note that 0x12345700 + 0x0800 = 0x12345f00)`
 1342 `addi x5,x5,0x700 // x5 = 0x12345000 + sx(0x700) = 0x12345000 + 0x00000700 = 0x12345700`

1343 The sign-extension in this example performed by the `addi` instruction will convert the `0x700` to
 1344 `0x00000700` before the addition.

1345 Observe that `0x12345700+0x0800 = 0x12345f00` and therefore, after shifting to the right, the least
 1346 significant `0xf00` is truncated, leaving `0x12345` as the immediate operand of the `lui` instruction. The
 1347 addition of `0x0800` in this example has no effect on the immediate operand of the `lui` instruction
 1348 because bit-11 in the original value `0x12345700` is zero.

1349 A general algorithm for implementing the `li rd,constant` pseudoinstruction is:

1350 `lui rd,(constant + 0x00000800) >> 12`
 1351 `addi rd,rd,(constant & 0x00000fff) // the 12-bit immediate is sign extended`

1352 Note that on RV64 and RV128 systems, the `lui` places the immediate operand into bits 31-12 and
 1353 then sign-extends the result to `XLEN` bits.

► Fix Me:
 Find a proper citation for
 this.

1354 **4.10.2 The la Pseudoinstruction**

1355 The **la** (and others that use **auipc** such as the **l{b|h|w}**, **s{b|h|w}**, **call**, and **tail**) pseudoinstructions
 1356 also compensate for a sign-ended negative number when adding a 12-bit immediate operand.
 1357 The only difference is that these use a pc-relative addressing mode.

1358 For example, consider the task of putting an address represented by the label **var1** into register **x10**:

```
1359 00010040 la x10, var1
1360 00010048 ... # note that the la pseudoinstruction expands into 8 bytes
1361 ...
1362
1363 var1:
1364 00010900 .word 999 # a 32-bit integer constant stored in memory at address var1
```

1365 The **la** instruction in this example will expand into:

```
1366 00010040 auipc x10,((var1-.) + 0x00000800) >> 12
1367 00010044 addi x10,x10,((var1-.-4)) & 0x00000fff
```

1368 Note that **auipc** will shift the immediate operand to the left 12 bits and then add that to the **pc**
 1369 register (see [Figure 5.3.1](#).)

1370 The assembler will calculate the value of **(var1-.)** by subtracting the address represented by the label
 1371 **var1** from the address of the current instruction (which is expressed as **'.'**) resulting in the number
 1372 of bytes from the current instruction to the target label... which is **0x000008c0**.

1373 Therefore the expanded pseudoinstruction example will become:

```
1374 00010040 auipc x10,((0x00010900 - 0x00010040) + 0x00000800) >> 12
1375 00010044 addi x10,x10,((0x00010900 - (0x00010044 - 4)) & 0x00000fff) # note the extra -4 here!
```

1376 After performing the subtractions, it will reduce to this:

```
1377 00010040 auipc x10,(0x000008c0 + 0x00000800) >> 12
1378 00010044 addi x10,x10,(0x000008c0 & 0x00000fff)
```

1379 Continuing to reduce the math operations we get:

```
1380 00010040 auipc x10,0x00001 # 0x000008c0 + 0x00000800 = 0x000010c0
1381 00010044 addi x10,x10,0x8c0
```

1382 Note that the **la** pseudoinstruction exhibits the same sort of technique as the **li** in that if/when the
 1383 immediate operand of the **addi** instruction has its most significant bit set then the operand in the
 1384 **auipc** has to be incremented by 1 to compensate.

1385 **4.11 Relocation**

1386 Because expressions that refer to constants and address labels are common in assembly language
 1387 programs, a shorthand notation is available for calculating the pairs of values that are used in the

1388 implementation of things like the `li` and `la` pseudoinstructions (that have to be written to compensate
 1389 for the sign-extension that will take place in the immediate operand that appears in instructions like
 1390 `addi` and `jalr`.)

1391 4.11.1 Absolute Addresses

1392 To refer to an absolute value, the following operators can be used:

```
1393     %hi(constant)    // becomes: (constant + 0x00000800) >> 12
1394     %lo(constant)    // becomes: (constant & 0x00000fff)
```

1395 Thus, the `li` pseudoinstruction can, therefore, be expressed like this:

```
1396     li    rd,constant    lui    rd,%hi(constant)
1397           addi    rd,rd,%lo(constant)
```

1398 4.11.2 PC-Relative Addresses

1399 The following can be used for PC-relative addresses:

```
1400     %pcrel_hi(symbol) // becomes: ((symbol-.) + 0x0800) >> 12
1401     %pcrel_lo(label)  // becomes: ((symbol-label) & 0x00000fff)
```

1402 Note the subtlety involved with the `label` on `%pcrel_lo`. It is needed to determine the address of the
 1403 instruction that contains the corresponding `%pcrel_hi`. (The label `label` MUST be on a line that used
 1404 a `%pcrel_hi()` or get an error from the assembler.)

1405 Thus, the `la rd,label` pseudoinstruction can be expressed like this:

```
1406     xxx: auipc rd,%pcrel_hi(label)
1407           addi  rd,rd,%pcrel_lo(xxx) // the xxx tells pcrel_lo where to find the matching pcrel_hi
```

1408 Examples of using the `auipc` & `addi` together with `%pcrel_hi()` and `%pcrel_lo()`:

```
1409     xxx: auipc  t1,%pcrel_hi(yyy)    // ((yyy-.) + 0x0800) >> 12
1410           addi    t1,t1,%pcrel_lo(xxx) // ((yyy-xxx) & 0x00000fff)
1411 ...
1412     yyy:                                // the address: yyy is saved into t1 above
1413 ...
```

1414 Referencing the same `%pcrel_hi` in multiple subsequent uses of `%pcrel_lo` is legal:

```
1415     label: auipc  t1,%pcrel_hi(symbol)
1416           addi    t2,t1,%pcrel_lo(label) // t2 = symbol
1417           addi    t3,t1,%pcrel_lo(label) // t3 = symbol
1418           lw     t4,%pcrel_lo(label)(t1) // t4 = fetch value from memory at 'symbol'
1419           addi    t4,t4,123           // t4 = t4 + 123
1420           sw     t4,%pcrel_lo(label)(t1) // store t4 back into memory at 'symbol'
```

1421

4.12 Relaxation

1422 In the simplest of terms, *Relaxation* refers to the ability of the linker (not the compiler!) to determine
1423 if/when the instructions that were generated with the `xxx_hi` and `xxx_lo` operators are unneeded
1424 (and thus waste execution time and memory) and can therefore be removed.

1425 However, doing so is not trivial as it will result in moving things around in memory, possibly changing
1426 the values of address labels in the already-assembled program! Therefore, while the motivation for
1427 relaxation is obvious, the process of implementing it is non-trivial.

1428 See: <https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md>

Chapter 5

RV32 Machine Instructions

5.1 Conventions and Terminology

When discussing instructions, the following abbreviations/notations are used:

5.1.1 XLEN

XLEN represents the bit-length of an **x** register in the machine architecture. Possible values are 32, 64 and 128.

5.1.2 sx(val)

Sign extend *val* to the left.

This is used to convert a signed integer value expressed using some number of bits to a larger number of bits by adding more bits to the left. In doing so, the sign will be preserved. In this case *val* represents the least [MSBs](#) of the value.

For more on sign-extension see [section 2.3](#).

5.1.3 zx(val)

Zero extend *val* to the left.

This is used to convert an unsigned integer value expressed using some number of bits to a larger number of bits by adding more bits to the left. In doing so, the new bits added will all be set to zero. As is the case with **sx(val)**, *val* represents the [LSBs](#) of the final value.

For more on zero-extension see [Figure 2.3](#).

5.1.4 $\text{zr}(\text{val})$

Zero extend val to the right.

Some times a binary value is encoded such that a set of bits represented by val are used to represent the MSBs of some longer (more bits) value. In this case it is necessary to append zeros to the right to convert val to the longer value.

Figure 5.1 illustrates converting a 20-bit val to a 32-bit fullword.

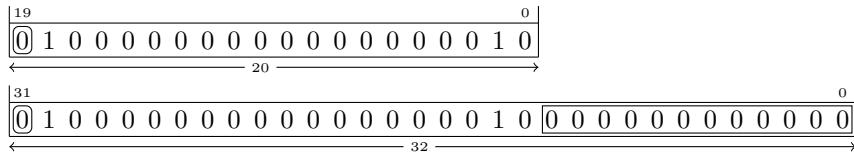


Figure 5.1: Zero-extending an integer to the right from 20 bits to 32 bits.

5.1.5 Sign Extended Left and Zero Extend Right

Some instructions such as the J-type (see section 5.3.2) include immediate operands that are extended in both directions.

Figure 5.2 and Figure 5.3 illustrates zero-extending a 20-bit negative number one bit to the right and sign-extending it 11 bits to the left:

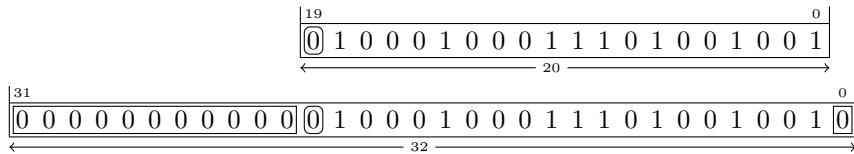


Figure 5.2: Sign-extending a positive 20-bit number 11 bits to the left and one bit to the right.

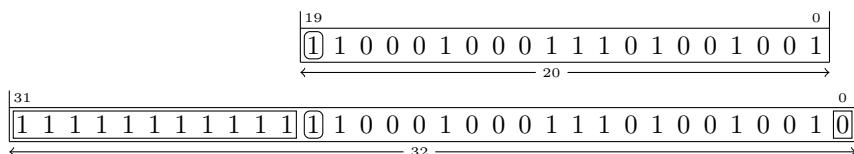


Figure 5.3: Sign-extending a negative 20-bit number 11 bits to the left and one bit to the right.

5.1.6 $\text{m8}(\text{addr})$

The contents of an 8-bit value in memory at address addr .

Given the contents of the memory dump shown in Figure 5.4, $\text{m8}(0x42)$ refers to the memory location at address 42_{16} that currently contains the 8-bit value fc_{16} .

The $\text{m}_n(\text{addr})$ notation can be used to refer to memory that is being read or written depending on the context.

1465 When memory is being written, the following notation is used to indicate that the least significant 8
1466 bits of *source* will be written into memory at the address *addr*:

1467 $m8(addr) \leftarrow source$

1468 When memory is being read, the following notation is used to indicate that the 8 bit value at the
1469 address *addr* will be read and stored into *dest*:

1470 $dest \leftarrow m8(addr)$

1471 Note that *source* and *dest* are typically registers.

00000030	2f	20	72	65	61	64	20	61	20	62	69	6e	61	72	79	20
00000040	66	69	fc	65	20	66	69	6c	6c	65	64	20	77	69	74	68
00000050	20	72	76	33	32	49	20	69	6e	73	74	72	75	63	74	69
00000060	6f	6e	73	20	61	6e	64	20	66	65	65	64	20	74	68	65

Figure 5.4: Sample memory contents.

1472 5.1.7 $m16(addr)$

1473 The contents of an 16-bit little-endian value in memory at address *addr*.

1474 Given the contents of the memory dump shown in [Figure 5.4](#), $m16(0x42)$ refers to the memory location
1475 at address 42_{16} that currently contains $65fc_{16}$. See also [section 5.1.6](#).

1476 5.1.8 $m32(addr)$

1477 The contents of an 32-bit little-endian value in memory at address *addr*.

1478 Given the contents of the memory dump shown in [Figure 5.4](#), $m32(0x42)$ refers to the memory location
1479 at address 42_{16} that currently contains $662065fc_{16}$. See also [section 5.1.6](#).

1480 5.1.9 $m64(addr)$

1481 The contents of an 64-bit little-endian value in memory at address *addr*.

1482 Given the contents of the memory dump shown in [Figure 5.4](#), $m64(0x42)$ refers to the memory location
1483 at address 42_{16} that currently contains $656c6c69662065fc_{16}$. See also [section 5.1.6](#).

1484 5.1.10 $m128(addr)$

1485 The contents of an 128-bit little-endian value in memory at address *addr*.

1486 Given the contents of the memory dump shown in [Figure 5.4](#), $m128(0x42)$ refers to the memory location
1487 at address 42_{16} that currently contains $7220687469772064656c6c69662065fc_{16}$. See also [sec-
1488 tion 5.1.6](#).

1489 **5.1.11 .+offset**

1490 The address of the current instruction plus a numeric offset.

1491 **5.1.12 .-offset**

1492 The address of the current instruction minus a numeric offset.

1493 **5.1.13 pcrel_13**1494 An address that is within $[-4096..4094]$ $[-0x1000..0x0ffe]$ of the current instruction location. These
1495 addresses are typically expressed in assembly source code by using labels. See [section 5.3.6](#) for exam-
1496 ples.1497 **5.1.14 pcrel_21**1498 An address that is within $[-1048576..1048574]$ $[-0x100000..0x0ffffe]$ of the current instruction loca-
1499 tion. These addresses are typically expressed in assembly source code by using labels. See [section 5.3.2](#)
1500 for an example.1501 **5.1.15 pc**

1502 The current value of the program counter.

1503 **5.1.16 rd**

1504 An x-register used to store the result of instruction.

1505 **5.1.17 rs1**

1506 An x-register value used as a source operand for an instruction.

1507 **5.1.18 rs2**

1508 An x-register value used as a source operand for an instruction.

1509 **5.1.19 imm**1510 An immediate numeric operand. The word *immediate* refers to the fact that the operand is stored
1511 within an instruction.

1512 **5.1.20 rsN[h:l]**1513 The value of bits from h through l of x-register rsN. For example: rs1[15:0] refers to the contents of
1514 the 16 [LSBs](#) of rs1.1515 **5.2 Addressing Modes**

1516 immediate, register, base-displacement, pc-relative

► Fix Me:
[Write this section.](#)

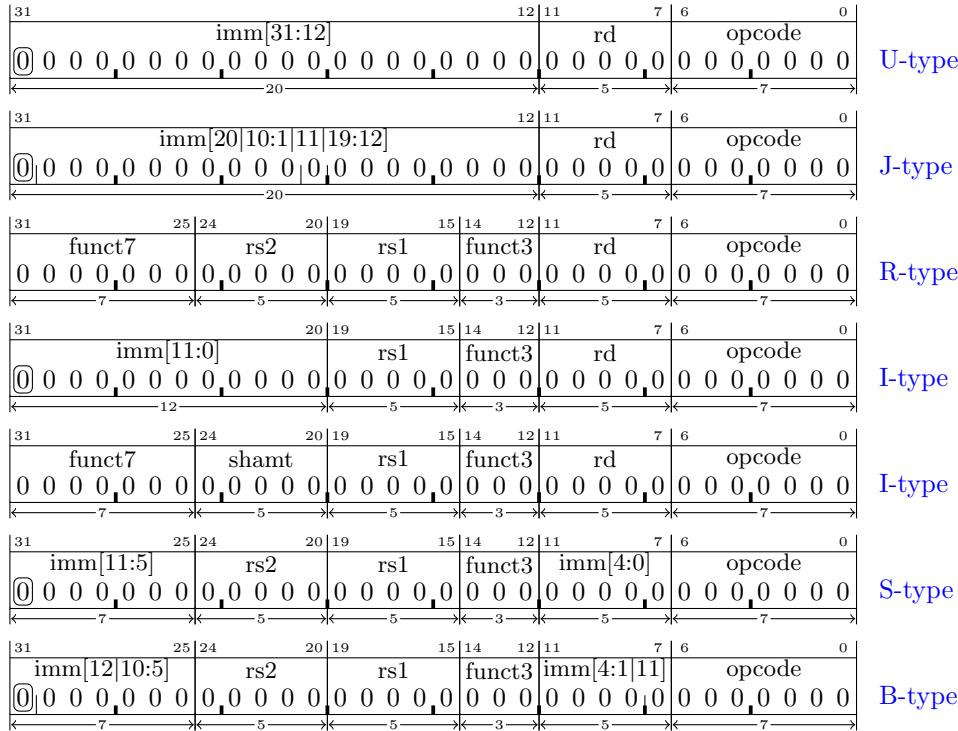
1517 **5.3 Instruction Encoding Formats**1518 This document concerns itself with the RISC-V instruction formats shown in [Figure 5.5](#).

Figure 5.5: RISC-V instruction formats.

1519 The method/format of the instructions has been designed with an eye on the ease of future manufacture
1520 of the machine that will execute them. It is easier to build a machine if it does not have to accommodate
1521 many different ways to perform the same task. The result is that a machine can be built with fewer
1522 gates, consumes less power, and can run faster than if it were built when a priority is on how a user
1523 might prefer to decode the same instructions from a hex dump.1524 Observe that all instructions have their opcode in bits 0-6 and when they include an rd register it will
1525 be specified in bits 7-11, an rs1 register in bits 15-19, an rs2 register in bits 20-24, and so on. This
1526 has a seemingly strange impact on the placement of any immediate operands.

1527 When immediate operands are present in an instruction, they are placed in the remaining unused bits.
 1528 However, they are organized such that the sign bit is *always* in bit 31 and the remaining bits placed
 1529 so as to minimize the number of places any given bit is located in different instructions.

1530 For example, consider immediate operand bits 12-19. In the U-type format they are in bit positions
 1531 12-19. In the J-type format they are also in positions 12-19. In the J-type format immediate operand
 1532 bits 1-10 are in the same instruction bit positions as they are in the I-type format and immediate
 1533 operand bits 5-10 are in the same positions as they are in the B-type and S-type formats.

1534 While this is inconvenient for anyone looking at a memory hexdump, it does make sense when consider-
 1535 ing the impact of this choice on the number of gates needed to implement circuitry to extract the
 1536 immediate operands.

1537 5.3.1 U Type

1538 The U-Type format is used for instructions that use a 20-bit immediate operand and an **rd** destination
 1539 register.

1540 The **rd** field contains an **x** register number to be set to a value that depends on the instruction.

1541 If **XLEN**=32 then the **imm** value will be extracted from the instruction and converted as shown in
 1542 Figure 5.6 to form the **imm_u** value.

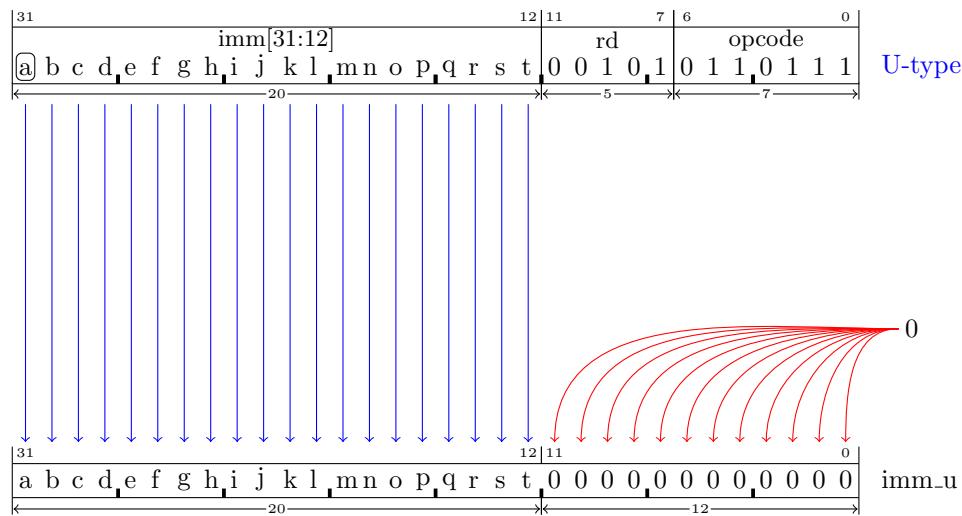


Figure 5.6: Decoding a U-type instruction.

1543 Notice that the 20-bits of the **imm** field are mapped in the same order and in the same relative position
 1544 that they appear in the instruction when they are used to create the value of the immediate operand.
 1545 Leaving the **imm** bits on the left, in the “upper bits” of the **imm_u** value suggests a rationale for the
 1546 name of this format.

1547 • **lui rd,imm**

1548 Set register **rd** to the **imm_u** value as shown in Figure 5.6.

1549 For example: **lui x23,0x12345** will result in setting register **x23** to the value **0x12345000**.

1550 • **auipc rd,imm**

1551 Add the address of the instruction to the `imm_u` value as shown [Figure 5.6](#) and store the result
 1552 in register `rd`.

1553 For example, if the instruction `auipc x22,0x10001` is executed from memory address `0x800012f4`
 1554 then register `x22` will be set to `0x900022f4`.

1555 If `XLEN=64` then the `imm_u` value in this example will be converted to the same two's complement
 1556 integer value by extending the sign-bit further to the left.

1557 5.3.2 J Type

1558 The J-type instruction format is used to encode the `jal` instruction with an immediate value that
 1559 determines the jump target address. It is similar to the U-type, but the bits in the immediate operand
 1560 are arranged in a different order.

1561 Note that the `imm_j` value is an even 21-bit value in the range of $[-1048576..1048574]$ $[-0x100000..0x0ffff]$
 1562 representing a pc-relative offset to the target address.

1563 If `XLEN=32` then the `imm` value will be extracted from the instruction and converted as shown in
 1564 [Figure 5.7](#) to form the `imm_j` value.

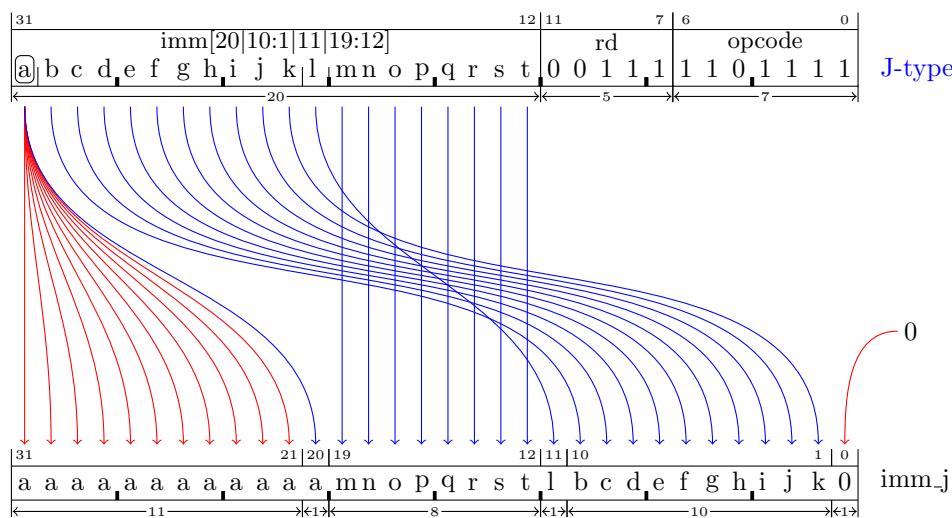


Figure 5.7: Decoding a J-type instruction.

1565 The J-type format is used by the Jump And Link instruction that calculates the target address by
 1566 adding `imm_j` to the current program counter. Since no instruction can be placed at an odd address the
 1567 20-bit imm value is zero-extended to the right to represent a 21-bit signed offset capable of expressing
 1568 a wider range of target addresses than the 20-bit imm value alone.

1569 • jal rd,pcrel_21

1570 Set register `rd` to the address of the next instruction that would otherwise be executed (the
 1571 address of the `jal` instruction + 4) and then jump to the address given by the sum of the `pc`
 1572 register and the `imm_j` value as decoded from the instruction shown in [Figure 5.7](#).

1573 Note that `pcrel_21` is expressed in the instruction as a target address or label that is converted
 1574 to a 21-bit value representing a pc-relative offset to the target address. For example, consider
 1575 the `jal` instructions in the following code:

```

1576      00000010: 000002ef jal      x5,0x10      # jump to self (address 0x10)
1577      00000014: 008002ef jal      x5,0x1c      # jump to address 0x1c
1578      00000018: 00100073 ebreak
1579      0000001c: 00100073 ebreak

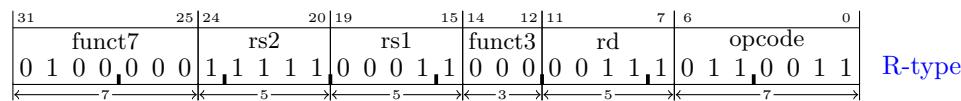
```

The instruction at address 0x10 has a target address of 0x10 and the `imm_j` is zero because offset from the “current instruction” to the target is zero.

The instruction at address 0x14 has a target address of 0x1c and the `imm_j` is 0x08 because $0x1c - 0x14 = 0x08$.

See also [section 5.3.6](#).

5.3.3 R Type



1587 The R-type instructions are used for operations that set a destination register `rd` to the result of an
1588 arithmetic, logical or shift operation applied to source registers `rs1` and `rs2`.

1589 Note that instruction bit 30 (part of the the `funct7` field) is used to select between the `add` and `sub`
1590 instructions as well as to select between `srl` and `sra`.

- `add rd,rs1,rs2`

1592 Set register `rd` to `rs1 + rs2`.

1593 Note that the value of `funct7` must be zero for this instruction. (The value of `funct7` is how
1594 the `add` instruction is differentiated from the `sub` instruction.)

- `and rd,rs1,rs2`

1596 Set register `rd` to the bitwise `and` of `rs1` and `rs2`.

1597 For example, if `x17 = 0x55551111` and `x18 = 0xff00ff00` then the instruction `and x12,x17,x18`
1598 will set `x12` to the value `0x55001100`.

- `or rd,rs1,rs2`

1600 Set register `rd` to the bitwise `or` of `rs1` and `rs2`.

1601 For example, if `x17 = 0x55551111` and `x18 = 0xff00ff00` then the instruction `or x12,x17,x18`
1602 will set `x12` to the value `0xff55ff11`.

- `sll rd,rs1,rs2`

1604 Shift `rs1` left by the number of bits specified in the least significant 5 bits of `rs2` and store the
1605 result in `rd`.¹

1606 For example, if `x17 = 0x12345678` and `x18 = 0x08` then the instruction `sll x12,x17,x18` will
1607 set `x12` to the value `0x34567800`.

- `slt rd,rs1,rs2`

1609 If the signed integer value in `rs1` is less than the signed integer value in `rs2` then set `rd` to 1.
1610 Otherwise, set `rd` to 0.

¹ When XLEN is 64 or 128, the shift distance will be given by the least-significant 6 or 7 bits of `rs2` respectively.
For more information on how shifting works, see [section 2.4](#).

1611 For example, if $x17 = 0x12345678$ and $x18 = 0x0000ffff$ then the instruction `slt x12,x17,x18`
1612 will set $x12$ to the value $0x00000000$.

1613 If $x17 = 0x82345678$ and $x18 = 0x0000ffff$ then the instruction `slt x12,x17,x18` will set
1614 $x12$ to the value $0x00000001$.

1615 • **`sltu rd,rs1,rs2`**

1616 If the unsigned integer value in `rs1` is less than the unsigned integer value in `rs2` then set `rd` to
1617 1. Otherwise, set `rd` to 0.

1618 For example, if $x17 = 0x12345678$ and $x18 = 0x0000ffff$ then the instruction `sltu x12,x17,x18`
1619 will set $x12$ to the value $0x00000000$.

1620 If $x17 = 0x12345678$ and $x18 = 0x8000ffff$ then the instruction `sltu x12,x17,x18` will set
1621 $x12$ to the value $0x00000001$.

1622 • **`sra rd,rs1,rs2`**

1623 Arithmetic-shift `rs1` right by the number of bits given in the least-significant 5 bits of the `rs2`
1624 register and store the result in `rd`.¹

1625 For example, if $x17 = 0x87654321$ and $x18 = 0x08$ then the instruction `sra x12,x17,x18` will
1626 set $x12$ to the value $0xff876543$.

1627 If $x17 = 0x76543210$ and $x18 = 0x08$ then the instruction `sra x12,x17,x18` will set $x12$ to the
1628 value $0x00765432$.

1629 Note that the value of `funct7` must be zero for this instruction. (The value of `funct7` is how
1630 the `sra` instruction is differentiated from the `srl` instruction.)

1631 • **`srl rd,rs1,rs2`**

1632 Logic-shift `rs1` right by the number of bits given in the least-significant 5 bits of the `rs2` register
1633 and store the result in `rd`.¹

1634 For example, if $x17 = 0x87654321$ and $x18 = 0x08$ then the instruction `srl x12,x17,x18` will
1635 set $x12$ to the value $0x00876543$.

1636 If $x17 = 0x76543210$ and $x18 = 0x08$ then the instruction `srl x12,x17,x18` will set $x12$ to the
1637 value $0x00765432$.

1638 Note that the value of `funct7` must be `0b0100000` for this instruction. (The value of `funct7` is how
1639 the `srl` instruction is differentiated from the `sra` instruction.)

1640 • **`sub rd,rs1,rs2`**

1641 Set register `rd` to `rs1 - rs2`.

1642 Note that the value of `funct7` must be `0b0100000` for this instruction. (The value of `funct7` is
1643 how the `sub` instruction is differentiated from the `add` instruction.)

1644 • **`xor rd,rs1,rs2`**

1645 Set register `rd` to the bitwise `xor` of `rs1` and `rs2`.

1646 For example, if $x17 = 0x55551111$ and $x18 = 0xff00ff00$ then the instruction `xor x12,x17,x18`
1647 will set $x12$ to the value $0xaa55ee11$.

1648 **5.3.4 I Type**

1649 The I-type instruction format is used to encode instructions with a signed 12-bit immediate operand
1650 with a range of $[-2048..2047]$, an `rd` register, and an `rs1` register.

1651 If `XLEN=32` then the 12-bit `imm` value example will be extracted from the instruction and converted as
1652 shown in [Figure 5.8](#) to form the `imm_i` value.

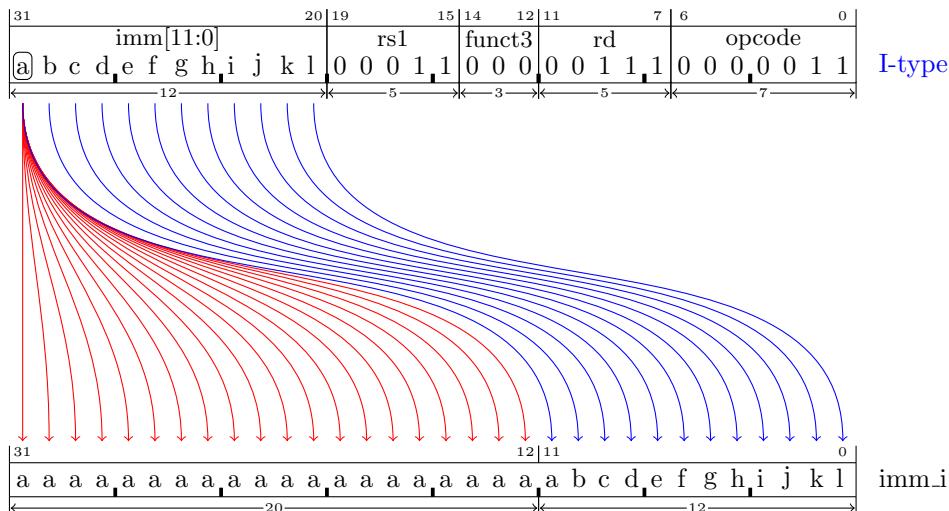


Figure 5.8: Decoding an I-type Instruction.

1653 A special case of the I-type is used for shift-immediate instructions where the imm field is used to
 1654 represent the number of bit positions to shift as shown in Figure 5.9. In this variation, the least
 1655 significant five bits of the imm field are extracted to form the `shamt_i` value.²

1656 Note also that bit 30 (the imm instruction field bit labeled 'b') is used to select between arithmetic
 1657 and logical shifting.

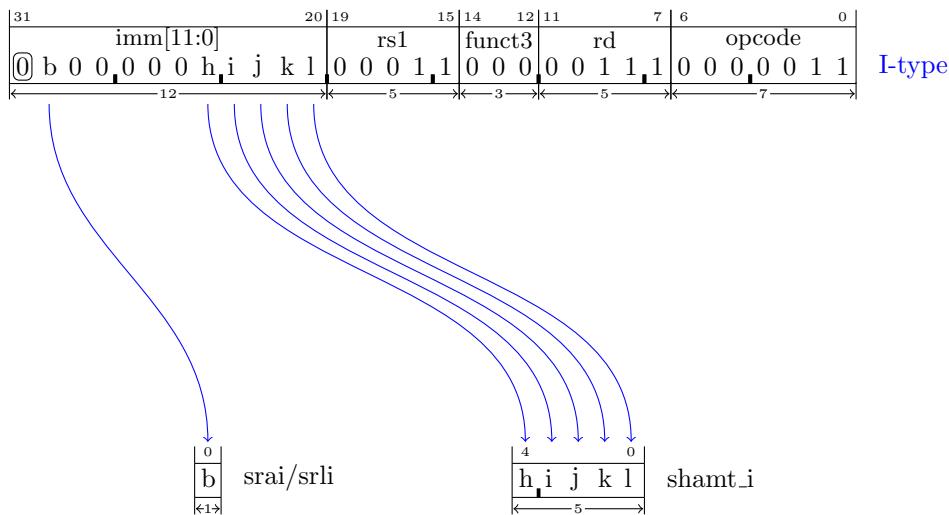


Figure 5.9: Decoding an I-type Shift Instruction.

1658 • `addi rd,rs1,imm`
 1659 Set register `rd` to `rs1 + imm_i`.
 1660 • `andi rd,rs1,imm`
 1661 Set register `rd` to the bitwise and of `rs1` and `imm_i`.

²When XLEN is 64 or 128, the `shamt_i` field will consist of 6 or 7 bits respectively.

```
00002640: 6f 00 00 00 6f 00 00 00 b7 87 00 00 03 a5 07 43 *o....o.....C*
00002650: 67 80 00 00 00 00 00 00 76 61 6c 3d 00 00 00 00 *g.....val=....*
00002660: 00 00 00 00 80 84 2e 41 1f 85 45 41 80 40 9a 44 *.....A..EA.@.D*
00002670: 4f 11 f3 c3 6e 8a 67 41 20 1b 00 00 20 1b 00 00 *0...n.gA ... ...
00002680: 44 1b 00 00 14 1b 00 00 14 1b 00 00 04 1c 00 00 *D.....*
```

Figure 5.10: An Example Memory Dump.

1662 For example, if $x17 = 0x55551111$ then the instruction `andi x12,x17,0x0ff` will set $x12$ to
1663 the value `0x00000011`.

1664 Recall that `imm` is sign-extended. Therefore if $x17 = 0x55551111$ then the instruction `andi x12,x17,0x800`
1665 will set $x12$ to the value `0x55551000`.

1666 • **jalr rd,imm(rs1)**

1667 Set register `rd` to the address of the next instruction that would otherwise be executed (the
1668 address of the `jalr` instruction + 4) and then jump to an address given by the sum of the `rs1`
1669 register and the `imm_i` value as decoded from the instruction shown in [Figure 5.8](#).

1670 Note that the `pc` register can never refer to an odd address. This instruction will explicitly set
1671 the `LSB` to zero regardless of the value of the calculated target address.

1672 • **lb rd,imm(rs1)**

1673 Set register `rd` to the value of the sign-extended byte fetched from the memory address given
1674 by the sum of `rs1` and `imm_i`.

1675 For example, given the memory contents shown in [Figure 5.10](#), if register $x13 = 0x00002650$
1676 then the instruction `lb x12,1(x13)` will set $x12$ to the value `0xfffff80`.

1677 • **lbu rd,imm(rs1)**

1678 Set register `rd` to the value of the zero-extended byte fetched from the memory address given
1679 by the sum of `rs1` and `imm_i`.

1680 For example, given the memory contents shown in [Figure 5.10](#), if register $x13 = 0x00002650$
1681 then the instruction `lbu x12,1(x13)` will set $x12$ to the value `0x00000080`.

1682 • **lh rd,imm(rs1)**

1683 Set register `rd` to the value of the sign-extended 16-bit little-endian half-word value fetched from
1684 the memory address given by the sum of `rs1` and `imm_i`.

1685 For example, given the memory contents shown in [Figure 5.10](#), if register $x13 = 0x00002650$
1686 then the instruction `lh x12,-2(x13)` will set $x12$ to the value `0x00004307`.

1687 If register $x13 = 0x00002650$ then the instruction `lh x12,-8(x13)` will set $x12$ to the value
1688 `0xfffff87b7`.

1689 • **lhu rd,imm(rs1)**

1690 Set register `rd` to the value of the zero-extended 16-bit little-endian half-word value fetched from
1691 the memory address given by the sum of `rs1` and `imm_i`.

1692 For example, given the memory contents shown in [Figure 5.10](#), if register $x13 = 0x00002650$
1693 then the instruction `lhu x12,-2(x13)` will set $x12$ to the value `0x00004307`.

1694 If register $x13 = 0x00002650$ then the instruction `lhu x12,-8(x13)` will set $x12$ to the value
1695 `0x000087b7`.

1696 • **lw rd,imm(rs1)**

1697 Set register `rd` to the value of the sign-extended 32-bit little-endian word value fetched from the
1698 memory address given by the sum of `rs1` and `imm_i`.

1699 For example, given the memory contents shown in [Figure 5.10](#), if register `x13 = 0x00002650`
 1700 then the instruction `lw x12,-4(x13)` will set `x12` to the value `4307a503`.

1701 • **ori rd,rs1,imm**

1702 Set register `rd` to the bitwise `or` of `rs1` and `imm_i`.

1703 For example, if `x17 = 0x55551111` then the instruction `ori x12,x17,0x0ff` will set `x12` to the
 1704 value `0x555511ff`.

1705 Recall that `imm` is sign-extended. Therefore if `x17 = 0x55551111` then the instruction `ori x12,x17,0x800`
 1706 will set `x12` to the value `0xfffff911`.

1707 • **slli rd,rs1,imm**

1708 Shift `rs1` left by the number of bits specified in `shamt_i` (as shown in [Figure 5.9](#)) and store the
 1709 result in `rd`.³

1710 For example, if `x17 = 0x12345678` then the instruction `slli x12,x17,4` will set `x12` to the
 1711 value `0x23456780`.

1712 • **slti rd,rs1,imm**

1713 If the signed integer value in `rs1` is less than the signed integer value in `imm_i` then set `rd` to 1.
 1714 Otherwise, set `rd` to 0.

1715 • **sltiu rd,rs1,imm**

1716 If the unsigned integer value in `rs1` is less than the unsigned integer value in `imm_i` then set `rd`
 1717 to 1. Otherwise, set `rd` to 0.

1718 Note that `imm_i` is always created by sign-extending the `imm` value as shown in [Figure 5.8](#) even
 1719 though it is then later used as an unsigned integer for the purposes of comparing its magnitude
 1720 to the unsigned value in `rs1`. Therefore, this instruction provides a method to compare `rs1` to
 1721 a value in the ranges of `[0..0x7ff]` and `[0xfffff800..0xffffffff]`.

1722 • **srai rd,rs1,imm**

1723 Arithmetic-shift `rs1` right by the number of bits specified in `shamt_i` (as shown in [Figure 5.9](#))
 1724 and store the result in `rd`.³

1725 For example, if `x17 = 0x87654321` then the instruction `srai x12,x17,4` will set `x12` to the
 1726 value `0xf8765432`.

1727 Note that the value of bit 30 must be 1 for this instruction. (The value of bit 30 is how the `srai`
 1728 instruction is differentiated from the `srl` instruction.)

1729 • **srl rd,rs1,imm**

1730 Logic-shift `rs1` right by the number of bits specified in `shamt_i` (as shown in [Figure 5.9](#)) and
 1731 store the result in `rd`.³

1732 For example, if `x17 = 0x87654321` then the instruction `srl x12,x17,4` will set `x12` to the
 1733 value `0x08765432`.

1734 Note that the value of bit 30 must be 0 for this instruction. (The value of bit 30 is how the `srl`
 1735 instruction is differentiated from the `srai` instruction.)

1736 • **xori rd,rs1,imm**

1737 Set register `rd` to the bitwise `xor` of `rs1` and `imm_i`.

1738 For example, if `x17 = 0x55551111` then the instruction `xori x12,x17,0x0ff` will set `x12` to the
 1739 value `0x555511ee`.

1740 Recall that `imm` is sign-extended. Therefore if `x17 = 0x55551111` then `xori x12,x17,0x800`
 1741 will set `x12` to the value `0xaaaae911`.

³ When `XLEN` is 64 or 128, the shift distance will be given by the least-significant 6 or 7 bits of the `imm` field respectively. For more information on how shifting works, see [section 2.4](#).

5.3.5 S Type

The S-type instruction format is used to encode instructions with a signed 12-bit immediate operand with a range of $[-2048..2047]$, an **rs1** register, and an **rs2** register.

If **XLEN**=32 then the 12-bit *imm* value example will be extracted from the instruction and converted as shown [Figure 5.11](#) to form the **imm_s** value.

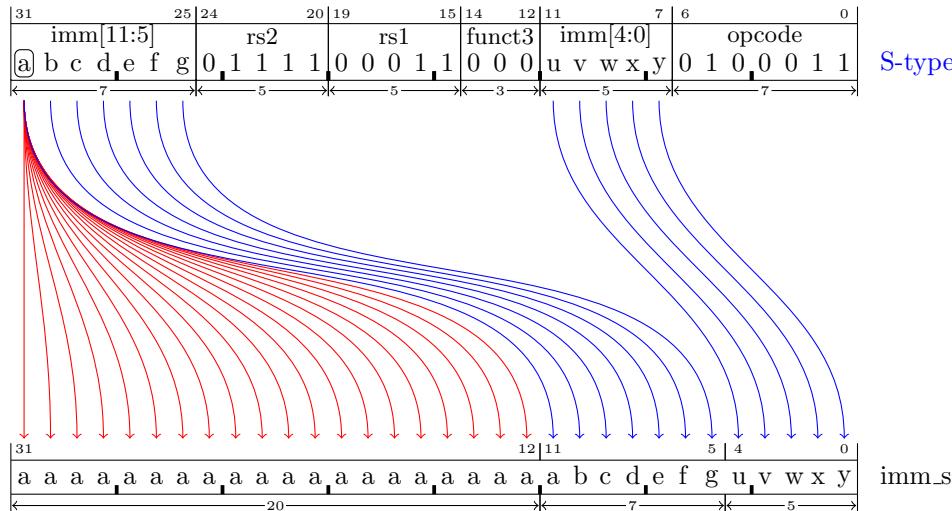


Figure 5.11: Decoding an S-type Instruction.

- **sb rs2,imm(rs1)**

Set the byte of memory at the address given by the sum of **rs1** and **imm_s** to the 8 **LSBs** of **rs2**.

For example, given the memory contents shown in [Figure 5.10](#), if registers **x13** = 0x00002650 and **x12** = 0x12345678 then the instruction **sb x12,1(x13)** will change the memory byte at address 0x00002651 from 0x80 to 0x78 resulting in:

```

00002640: 6f 00 00 00 6f 00 00 00 b7 87 00 00 03 a5 07 43 *o....o.....C*
00002650: 67 78 00 00 00 00 00 00 76 61 6c 3d 00 00 00 00 *gx.....val=....*
00002660: 00 00 00 00 80 84 2e 41 1f 85 45 41 80 40 9a 44 *.....A..EA..D*
00002670: 4f 11 f3 c3 6e 8a 67 41 20 1b 00 00 20 1b 00 00 *0...n.gA ... ...*
00002680: 44 1b 00 00 14 1b 00 00 14 1b 00 00 04 1c 00 00 *D.....*
```

- **sh rs2,imm(rs1)**

Set the 16-bit half-word of memory at the address given by the sum of **rs1** and **imm_s** to the 16 **LSBs** of **rs2**.

For example, given the memory contents shown in [Figure 5.10](#), if registers **x13** = 0x00002650 and **x12** = 0x12345678 then the instruction **sh x12,2(x13)** will change the memory half-word at address 0x00002652 from 0x0000 to 0x5678 resulting in:

```

00002640: 6f 00 00 00 6f 00 00 00 b7 87 00 00 03 a5 07 43 *o....o.....C*
00002650: 67 80 78 56 00 00 00 00 76 61 6c 3d 00 00 00 00 *g.xV.....val=....*
00002660: 00 00 00 00 80 84 2e 41 1f 85 45 41 80 40 9a 44 *.....A..EA..D*
00002670: 4f 11 f3 c3 6e 8a 67 41 20 1b 00 00 20 1b 00 00 *0...n.gA ... ...*
00002680: 44 1b 00 00 14 1b 00 00 14 1b 00 00 04 1c 00 00 *D.....*
```

1768 • **SW rs2,imm(rs1)**

1769 Store the 32-bit value in **rs2** into the memory at the address given by the sum of **rs1** and **imm_s**.

1770 For example, given the memory contents shown in [Figure 5.10](#), if registers **x13** = 0x00002650
1771 and **x12** = 0x12345678 then the instruction **sw x12,0(x13)** will change the memory word at
1772 address 0x00002650 from 0x00008067 to 0x12345678 resulting in:

```
1773 00002640: 6f 00 00 00 6f 00 00 00 b7 87 00 00 03 a5 07 43 *o...o.....C*
1774 00002650: 78 56 34 12 00 00 00 00 76 61 6c 3d 00 00 00 00 *xV4.....val=....*
1775 00002660: 00 00 00 00 80 84 2e 41 1f 85 45 41 80 40 9a 44 *.....A..EA.C.D*
1776 00002670: 4f 11 f3 c3 6e 8a 67 41 20 1b 00 00 20 1b 00 00 *0...n.gA ... . .*
1777 00002680: 44 1b 00 00 14 1b 00 00 14 1b 00 00 04 1c 00 00 *D.....*
```

1778 **5.3.6 B Type**

1779 The B-type instruction format is used for branch instructions that require an even immediate value
1780 that is used to determine the branch target address as an offset from the current instruction's address.

1781 If **XLEN**=32 then the 12-bit *imm* value example will be extracted from the instruction and converted as
1782 shown in [Figure 5.12](#) to form the **imm_b** value.

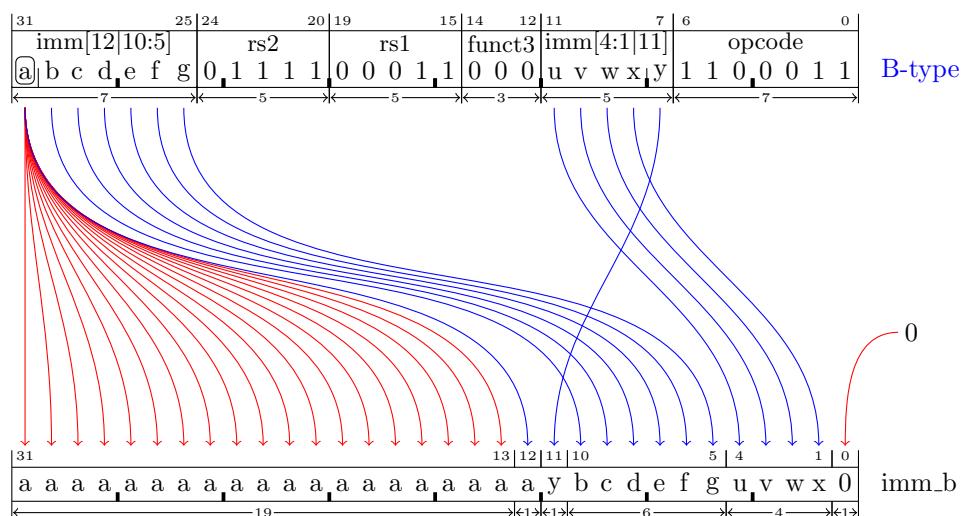


Figure 5.12: Decoding a B-type Instruction.

1783 Note that **imm_b** is expressed in the instruction as a target address that is converted to an even 13-bit
1784 value in the range of [-4096..4094] [-0x1000..0x0ffe] representing a pc-relative offset to the target
1785 address. For example, consider the branch instructions in the following code:

```
1786 00000000: 00520063 beq    x4,x5,0x0    # branches to self (address 0x0)
1787 00000004: 00520463 beq    x4,x5,0xc    # branches to address 0xc
1788 00000008: fe520ce3 beq    x4,x5,0x0    # branches to address 0x0
1789 0000000c: 00100073 ebreak
```

1790 The instruction at address 0x0 has a target address of zero and **imm_b** is zero because the offset from
1791 the “current instruction” to the target is zero.⁴

⁴This is in contrast to many other instruction sets with pc-relative addressing modes that express a branch target offset from the “next instruction.”

1792 The instruction at address 0x4 has a target address of 0xc and it has an `imm_b` of 0x08 because
 1793 $0x4 + 0x08 = 0x0c$.

1794 The instruction at address 0x8 has a target address of zero and `imm_b` is 0xffffffff8 (-8) because
 1795 $0x8 + 0xffffffff8 = 0x0$.

1796 • **beq rs1,rs2,pcrel_13**

1797 If `rs1` is equal to `rs2` then add `imm_b` to the `pc` register.

1798 • **bge rs1,rs2,pcrel_13**

1799 If the signed value in `rs1` is greater than or equal to the signed value in `rs2` then add `imm_b` to
 1800 the `pc` register.

1801 • **bgeu rs1,rs2,pcrel_13**

1802 If the unsigned value in `rs1` is greater than or equal to the unsigned value in `rs2` then add `imm_b`
 1803 to the `pc` register.

1804 • **blt rs1,rs2,pcrel_13**

1805 If the signed value in `rs1` is less than the signed value in `rs2` then add `imm_b` to the `pc` register.

1806 • **bltu rs1,rs2,pcrel_13**

1807 If the unsigned value in `rs1` is less than the unsigned value in `rs2` then add `imm_b` to the `pc`
 1808 register.

1809 • **bne rs1,rs2,pcrel_13**

1810 If `rs1` is not equal to `rs2` then add `imm_b` to the `pc` register.

1811

5.4 CPU Registers

1812 The registers are names `x0` through `x31` and have aliases suited to their conventional use. The following
 1813 table describes each register.

1814 Note that the calling convention specifies that only some of the registers are to be saved by
 1815 functions if they alter their contents. The idea being that accessing memory is time-consuming and
 1816 that by classifying some registers as “temporary” (not saved by any function that alter its contents)
 1817 it is possible to carefully implement a function with less need to store register values on the stack in
 1818 order to use them to perform the operations of the function.

► Fix Me:

*Need to add a section that
discusses the calling
conventions*

1819 The lack of grouping the temporary and saved registers is due to the fact that the E extension only
 1820 has the first 16 registers and some of the instructions in the C extension can only refer to the first 16
 1821 registers.

Reg	ABI/Alias	Description	Saved
x0	zero	Hard-wired zero	
x1	ra	Return address	
x2	sp	Stack pointer	yes
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary/alternate link register	
x6-7	t1-2	Temporaries	
x8	s0/fp	Saved register/frame pointer	yes
x9	s1	Saved register	yes
x10-11	a0-1	Function arguments/return value	
x12-17	a2-7	Function arguments	
x18-27	s2-11	Saved registers	yes
x28-31	t3-6	Temporaries	

1822

5.5 memory

1823

Note that RISC-V is a little-endian machine.

1824

All instructions must be naturally aligned to their 4-byte boundaries. [1, p. 5]

1825

If a RISC-V processor implements the C (compressed) extension then instructions may be aligned to 2-byte boundaries. [1, p. 68]

1826

Data alignment is not necessary but unaligned data can be inefficient. Accessing unaligned data using any of the load or store instructions can also prevent a memory access from operating atomically. [1, p.19]

1831

Appendix A

1832

Installing a RISC-V Toolchain

1833

All of the software presented in this text was assembled/compiled using the GNU toolchain and executed using the rvddt simulator on a Linux (Ubuntu 20.04 LTS) operating system.

1835

The installation instructions provided here were last tested on March 5, 2021.

1836

It is expected that these tools will evolve over time. See the respective documentation web sites for the latest news and options for installing them.

1838

A.1 The GNU Toolchain

1839

In order to install custom code in a location that will not cause interference with other applications (and allow for easy hacking and cleanup), these will install the toolchain under a private directory: `~/projects/riscv/install`. At any time you can remove everything and start over by executing the following command:

1843

```
1 rm -rf ~/projects/riscv/install
```

► Fix Me:

It would be good to find some Mac and Windows users to write and test proper variations on this section to address those systems. Pull requests, welcome!

Be *very* careful how you type the above `rm` command. If typed incorrectly, it could irreversibly remove many of your files!

1846

Before building the toolchain, a number of utilities must be present on your system. The following will install those that are needed:

1849

```
1 sudo apt install autoconf automake autotools-dev curl python3 python-dev libmpc-dev \
2 libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf \
3 libtool patchutils bc zlib1g-dev libexpat-dev
```

1850

Note that the above `apt` command is the only operation that should be performed as root. All other commands should be executed as a regular user. This will eliminate the possibility of clobbering system files that should not be touched when tinkering with the toolchain applications.

1853

To download, compile and install the toolchain:

► Fix Me:

Discuss the choice of ilp32 as well as what the other variations would do.

```

1  mkdir -p ~/projects/riscv
2  cd ~/projects/riscv
3  git clone https://github.com/riscv/riscv-gnu-toolchain
4  cd riscv-gnu-toolchain
1854 5  INS_DIR=~/projects/riscv/install/rv32i
6  ./configure --prefix=$INS_DIR \
7      --with-multilib-generator="rv32i-ilp32--;rv32imafd-ilp32--;rv32ima-ilp32--"
8  make

```

1855 After building the toolchain, make it available by putting it into your PATH by adding the following
1856 to the end of your `.bashrc` file:

```

1857
1858 1  export PATH=$PATH:$INS_DIR

```

1860 For this PATH change to take place, start a new terminal or paste the same `export` command into
1861 your existing terminal.

1862 A.2 rvddt

1863 Download and install the rvddt simulator by executing the following commands. Building the rvddt
1864 example programs will verify that the GNU toolchain has been built and installed properly.

```

1  cd ~/projects/riscv
2  git clone https://github.com/johnwinans/rvddt.git
3  cd rvddt/src
1865 4  make world
5  cd ../../examples
6  make world

```

1866 After building rvddt, make it available by putting it into your PATH by adding the following to the
1867 end of your `.bashrc` file:

```

1868
1869 1  export PATH=$PATH:~/projects/riscv/rvddt/src

```

1871 For this PATH change to take place, start a new terminal or paste the same `export` command into
1872 your existing terminal.

1873 Test the rvddt build by executing one of the examples:

```

1  winans@ux410:~/projects/riscv/rvddt/examples$ rvddt -f counter/counter.bin
2  sp initialized to top of memory: 0x00000fff0
3  Loading 'counter/counter.bin' to 0x0
4  This is rvddt. Enter ? for help.
5  ddt> ti 0 1000
6  00000000: 00300293 addi    x5, x0, 3      # x5 = 0x00000003 = 0x00000000 + 0x00000003
7  00000004: 00000313 addi    x6, x0, 0      # x6 = 0x00000000 = 0x00000000 + 0x00000000
8  00000008: 00130313 addi    x6, x6, 1      # x6 = 0x00000001 = 0x00000000 + 0x00000001
1874 9  0000000c: fe534ee3 blt     x6, x5, -4    # pc = (0x1 < 0x3) ? 0x8 : 0x10
10 00000008: 00130313 addi    x6, x6, 1      # x6 = 0x00000002 = 0x00000001 + 0x00000001
11 0000000c: fe534ee3 blt     x6, x5, -4    # pc = (0x2 < 0x3) ? 0x8 : 0x10
12 00000008: 00130313 addi    x6, x6, 1      # x6 = 0x00000003 = 0x00000002 + 0x00000001
13 0000000c: fe534ee3 blt     x6, x5, -4    # pc = (0x3 < 0x3) ? 0x8 : 0x10
14 00000010: ebreak
15 ddt> x
16 winans@ux410:~/projects/riscv/rvddt/examples$
```

1875 A.3 qemu

1876 You can download and install the RV32 qemu simulator by executing the following commands.

1877 At the time of this writing (2021-06) I use release v5.0.0. Release v5.2.0 has issues that confuse GDB
1878 when printing the registers and v6.0.0 has different CPU types that I have had trouble with when
1879 executing privileged instructions.

```
1 INS_DIR=~/projects/riscv/install/rv32i
2 cd ~/projects/riscv
3 git clone git@github.com:qemu/qemu.git
4 cd qemu
1880 5 git checkout v5.0.0
6 ./configure --target-list=riscv32-softmmu --prefix=${INS_DIR}
7 make -j4
8 make install
```

1881

Appendix B

1882

Floating Point Numbers

1883

B.1 IEEE-754 Floating Point Number Representation

1884

This section provides an overview of the IEEE-754 32-bit binary floating point format.[\[15\]](#)

1885

- Recall that the place values for integer binary numbers are:

1886

... 128 64 32 16 8 4 2 1

1887

- We can extend this to the right in binary similar to the way we do for decimal numbers:

1888

... 128 64 32 16 8 4 2 1 . 1/2 1/4 1/8 1/16 1/32 1/64 1/128 ...

1889

The ‘.’ in a binary number is a binary point, not a decimal point.

1890

- We use scientific notation as in 2.7×10^{-47} to express either small fractions or large numbers when we are not concerned every last digit needed to represent the entire, exact, value of a number.

1893

- The format of a number in scientific notation is $mantissa \times base^{exponent}$

1894

- In binary we have $mantissa \times 2^{exponent}$

1895

- IEEE-754 format requires binary numbers to be *normalized* to $1.significand \times 2^{exponent}$ where the *significand* is the portion of the *mantissa* that is to the right of the binary-point.

1897

- The unnormalized binary value of -2.625 is -10.101

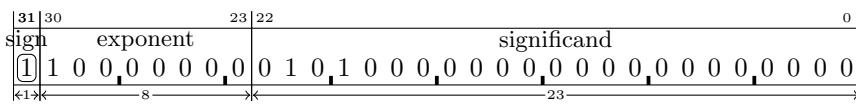
1898

- The normalized value of -2.625 is -1.0101×2^1

1899

- We need not store the ‘1.’ part because *all* normalized floating point numbers will start that way. Thus we can save memory when storing normalized values by inserting a ‘1.’ to the left of *significand*.

1902



1903

- $-((1 + \frac{1}{4} + \frac{1}{16}) \times 2^{128-127}) = -((1 + \frac{1}{4} + \frac{1}{16}) \times 2^1) = -(2 + \frac{1}{2} + \frac{1}{8}) = -(2 + .5 + .125) = -2.625$

1904 • IEEE-754 formats:

	IEEE-754 32-bit	IEEE-754 64-bit
sign	1 bit	1 bit
exponent	8 bits (excess-127)	11 bits (excess-1023)
mantissa	23 bits	52 bits
max exponent	127	1023
min exponent	-126	-1022

1906 • When the exponent is all ones, the significand is all zeros, and the sign is zero, the number
1907 represents positive infinity.

1908 • When the exponent is all ones, the significand is all zeros, and the sign is one, the number
1909 represents negative infinity.

1910 • Observe that the binary representation of a pair of IEEE-754 numbers (when one or both are
1911 positive) can be compared for magnitude by treating them as if they are two's complement
1912 signed integers. This is because an IEEE number is stored in *signed magnitude* format and
1913 therefore positive floating point values will grow upward and downward in the same fashion as
1914 for unsigned integers and that since negative floating point values will have its MSB set, they
1915 will ‘appear’ to be less than a positive floating point value.

1916 When comparing two negative IEEE float values by treating them both as two's complement
1917 signed integers, the order will be reversed because IEEE float values with larger (that is, in-
1918 creasingly negative) magnitudes will appear to decrease in value when interpreted as signed
1919 integers.

1920 This works this way because excess notation is used in the format of the exponent and why the
1921 significand's sign bit is located on the left of the exponent.¹

1922 • Note that zero is a special case number. Recall that a normalized number has an implied 1-bit
1923 to the left of the significand... which means that there is no way to represent zero! Zero is
1924 represented by an exponent of all-zeros and a significand of all-zeros. This definition allows for
1925 a positive and a negative zero if we observe that the sign can be either 1 or 0.

1926 • On the number-line, numbers between zero and the smallest fraction in either direction are in
1927 the *underflow* areas.

1928 • On the number line, numbers greater than the mantissa of all-ones and the largest exponent
1929 allowed are in the *overflow* areas.

1930 • Note that numbers have a higher resolution on the number line when the exponent is smaller.

1931 • The largest and smallest possible exponent values are reserved to represent things requiring
1932 special cases. For example, the infinities, values representing “not a number” (such as the result
1933 of dividing by zero), and for a way to represent values that are not normalized. For more
1934 information on special cases see [15].

► Fix Me:
Need to add the standard
lecture number-line diagram
showing where the
over/under-flow areas are
and why.

1935 B.1.1 Floating Point Number Accuracy

1936 Due to the finite number of bits used to store the value of a floating point number, it is not possible to
1937 represent every one of the infinite values on the real number line. The following C programs illustrate
1938 this point.

¹I know this is true and was done on purpose because Bill Cody, chairman of IEEE committee P754 that designed the IEEE-754 standard, told me so personally circa 1991.

1939

B.1.1.1 Powers Of Two

1940

Just like the integer numbers, the powers of two that have bits to represent them can be represented perfectly... as can their sums (provided that the significand requires no more than 23 bits.)

1941

Listing B.1: `powersoftwo.c`

Precise Powers of Two

1942

1943

1944

1945

1946

1947

1948

1949

1950

1951

1952

1953

1954

1955

1956

1957

1958

1959

1960

1961

1962

1963

1964

1965

1966

1967

1968

1969

1970

1971

1972

1973

1974

1975

1976

1977

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 union floatbin
6 {
7     unsigned int    i;
8     float          f;
9 };
10 int main()
11 {
12     union floatbin x;
13     union floatbin y;
14     int             i;
15     x.f = 1.0;
16     while (x.f > 1.0/1024.0)
17     {
18         y.f = -x.f;
19         printf("%25.10f = %08x    %25.10f = %08x\n", x.f, x.i, y.f, y.i);
20         x.f = x.f/2.0;
21     }
22 }
```

Listing B.2: `powersoftwo.out`Output from `powersoftwo.c`

1966

1967

1968

1969

1970

1971

1972

1973

1974

1975

1976

1977

1.0000000000 = 3f800000	-1.0000000000 = bf800000
0.5000000000 = 3f000000	-0.5000000000 = bf000000
0.2500000000 = 3e800000	-0.2500000000 = be800000
0.1250000000 = 3e000000	-0.1250000000 = be000000
0.0625000000 = 3d800000	-0.0625000000 = bd800000
0.0312500000 = 3d000000	-0.0312500000 = bd000000
0.0156250000 = 3c800000	-0.0156250000 = bc800000
0.0078125000 = 3c000000	-0.0078125000 = bc000000
0.0039062500 = 3b800000	-0.0039062500 = bb800000
0.0019531250 = 3b000000	-0.0019531250 = bb000000

1978

B.1.1.2 Clean Decimal Numbers

1979

When dealing with decimal values, you will find that they don't map simply into binary floating point values.

1980

1981

1982

1983

1984

1985

1986

1987

Note how the decimal numbers are not accurately represented as they get larger. The decimal number on line 10 of [Listing B.4](#) can be perfectly represented in IEEE format. However, a problem arises in the 11th loop iteration. It is due to the fact that the binary number can not be represented accurately in IEEE format. Its least significant bits were truncated in a best-effort attempt at rounding the value off in order to fit the value into the bits provided. This is an example of *low order truncation*. Once this happens, the value of `x.f` is no longer as precise as it could be given more bits in which to save its value.

Listing B.3: `cleandecimal.c`

Print Clean Decimal Numbers

```

1988 1 #include <stdio.h>
1989 2 #include <stdlib.h>
1990 3 #include <unistd.h>
1991 4
1992 5 union floatbin
1993 6 {
1994 7     unsigned int    i;
1995 8     float          f;
1996 9 };
1997 10 int main()
1998 11 {
1999 12     union floatbin x, y;
2000 13     int             i;
2001 14
2002 15     x.f = 10;
2003 16     while (x.f <= 1000000000000.0)
2004 17     {
2005 18         y.f = -x.f;
2006 19         printf("%25.10f = %08x      %25.10f = %08x\n", x.f, x.i, y.f, y.i);
2007 20         x.f = x.f*10.0;
2008 21     }
2009 22 }

```

Listing B.4: cleandecimal.out

Output from cleandecimal.c

```

2012 1
2013 1 10.0000000000 = 41200000      -10.0000000000 = c1200000
2014 2 100.0000000000 = 42c80000     -100.0000000000 = c2c80000
2015 3 1000.0000000000 = 447a0000    -1000.0000000000 = c47a0000
2016 4 10000.0000000000 = 461c4000   -10000.0000000000 = c61c4000
2017 5 100000.0000000000 = 47c35000  -100000.0000000000 = c7c35000
2018 6 1000000.0000000000 = 49742400 -1000000.0000000000 = c9742400
2019 7 10000000.0000000000 = 4b189680 -10000000.0000000000 = cb189680
2020 8 100000000.0000000000 = 4cbebc20 -100000000.0000000000 = ccbebc20
2021 9 1000000000.0000000000 = 4e6e6b28 -1000000000.0000000000 = ce6e6b28
2022 10 10000000000.0000000000 = 501502f9 -10000000000.0000000000 = d01502f9
2023 11 99999997952.0000000000 = 51ba43b7 -99999997952.0000000000 = d1ba43b7
2024 12 99999995904.0000000000 = 5368d4a5 -99999995904.0000000000 = d368d4a5
2025 13 999999827968.0000000000 = 551184e7 -999999827968.0000000000 = d51184e7

```

B.1.1.3 Accumulation of Error

2028 These rounding errors can be exaggerated when the number we multiply the `x.f` value by is, itself,
 2029 something that can not be accurately represented in IEEE form.²

2030 For example, if we multiply our `x.f` value by $\frac{1}{10}$ each time, we can never be accurate and we start
 2031 accumulating errors immediately.

Fix Me:

In a lecture one would show that one tenth is a repeating non-terminating binary number that gets truncated. This discussion should be reproduced here in text form.

Listing B.5: erroraccumulation.c

Accumulation of Error

```

2032 1
2033 1 #include <stdio.h>
2034 2 #include <stdlib.h>
2035 3 #include <unistd.h>
2036 4
2037 5 union floatbin
2038 6 {
2039 7     unsigned int    i;
2040 8     float          f;

```

²Applications requiring accurate decimal values, such as financial accounting systems, can use a packed-decimal numeric format to avoid unexpected oddities caused by the use of binary numbers.

```
2041 9 };
```

```
2042 10 int main()
```

```
2043 11 {
```

```
2044 12     union floatbin x, y;
```

```
2045 13     int i;
```

```
2046 14
```

```
2047 15     x.f = .1;
```

```
2048 16     while (x.f <= 2.0)
```

```
2049 17 {
```

```
2050 18     y.f = -x.f;
```

```
2051 19     printf("%25.10f = %08x      %25.10f = %08x\n", x.f, x.i, y.f, y.i);
```

```
2052 20     x.f += .1;
```

```
2053 21 }
```

```
2054 22 }
```

Listing B.6: `erroraccumulation.out`

Output from `erroraccumulation.c`

2056				
2057	1	0.1000000015	= 3dcccccd	-0.1000000015 = bdcccccd
2058	2	0.2000000030	= 3e4ccccd	-0.2000000030 = be4ccccd
2059	3	0.3000000119	= 3e99999a	-0.3000000119 = be99999a
2060	4	0.4000000060	= 3eccccc d	-0.4000000060 = becccccd
2061	5	0.5000000000	= 3f000000	-0.5000000000 = bf000000
2062	6	0.6000000238	= 3f19999a	-0.6000000238 = bf19999a
2063	7	0.7000000477	= 3f333334	-0.7000000477 = bf333334
2064	8	0.8000000715	= 3f4cccc e	-0.8000000715 = bf4cccc e
2065	9	0.9000000954	= 3f666668	-0.9000000954 = bf666668
2066	10	1.0000001192	= 3f800001	-1.0000001192 = bf800001
2067	11	1.1000001431	= 3f8cccc e	-1.1000001431 = bf8cccc e
2068	12	1.2000001669	= 3f99999b	-1.2000001669 = bf99999b
2069	13	1.3000001907	= 3fa66668	-1.3000001907 = bfa66668
2070	14	1.4000002146	= 3fb33335	-1.4000002146 = bf b33335
2071	15	1.5000002384	= 3fc00002	-1.5000002384 = bf c00002
2072	16	1.6000002623	= 3fc ccccf	-1.6000002623 = bf ccccf
2073	17	1.7000002861	= 3fd9999c	-1.7000002861 = bfd9999c
2074	18	1.8000003099	= 3fe66669	-1.8000003099 = bfe66669
2075	19	1.9000003338	= 3ff33336	-1.9000003338 = bff33336

B.1.2 Reducing Error Accumulation

In order to use floating point numbers in a program without causing excessive rounding problems an algorithm can be redesigned such that the accumulation is eliminated. This example is similar to the previous one, but this time we recalculate the desired value from a known-accurate integer value. Some rounding errors remain present, but they can not accumulate.

Listing B.7: `errorcompensation.c`

Accumulation of Error

```
2082  
2083 1 #include <stdio.h>  
2084 2 #include <stdlib.h>  
2085 3 #include <unistd.h>  
2086 4  
2087 5 union floatbin  
2088 6 {  
2089 7     unsigned int      i;  
2090 8     float          f;  
2091 9 };  
2092 10 int main()  
2093 11 {  
2094 12     union floatbin x, y;  
2095 13     int           i;  
2096 14  
2097 15     i = 1;
```

```

2098 16     while (i <= 20)
2099 17     {
2100 18         x.f = i/10.0;
2101 19         y.f = -x.f;
2102 20         printf("%25.10f = %08x      %25.10f = %08x\n", x.f, x.i, y.f, y.i);
2103 21         i++;
2104 22     }
2105 23     return(0);
2106 24 }
```

Listing B.8: `errorcompensation.out`Output from `erroraccumulation.c`

```

2108 1 0.1000000015 = 3dcccccd          -0.1000000015 = bdcccccd
2109 2 0.2000000030 = 3e4ccccd          -0.2000000030 = be4ccccd
2110 3 0.3000000119 = 3e99999a          -0.3000000119 = be99999a
2111 4 0.4000000060 = 3eccccc          -0.4000000060 = becccccd
2112 5 0.5000000000 = 3f000000          -0.5000000000 = bf000000
2113 6 0.6000000238 = 3f19999a          -0.6000000238 = bf19999a
2114 7 0.6999999881 = 3f333333          -0.6999999881 = bf333333
2115 8 0.8000000119 = 3f4ccccd          -0.8000000119 = bf4ccccd
2116 9 0.8999999762 = 3f666666          -0.8999999762 = bf666666
2117 10 1.0000000000 = 3f800000          -1.0000000000 = bf800000
2118 11 1.1000000238 = 3f8ccccd          -1.1000000238 = bf8ccccd
2119 12 1.2000000477 = 3f99999a          -1.2000000477 = bf99999a
2120 13 1.2999999523 = 3fa66666          -1.2999999523 = bfa66666
2121 14 1.3999999762 = 3fb33333          -1.3999999762 = bfb33333
2122 15 1.5000000000 = 3fc00000          -1.5000000000 = bfc00000
2123 16 1.6000000238 = 3fccccc          -1.6000000238 = bfcccccd
2124 17 1.7000000477 = 3fd9999a          -1.7000000477 = bfd9999a
2125 18 1.7999999523 = 3fe66666          -1.7999999523 = bfe66666
2126 19 1.8999999762 = 3ff33333          -1.8999999762 = bff33333
2127 20 2.0000000000 = 40000000          -2.0000000000 = c0000000
```

2130

Appendix C

2131

The ASCII Character Set

2132

A slightly abridged version of the Linux “ASCII” man(1) page.

2133

C.1 NAME

2134

ascii - ASCII character set encoded in octal, decimal, and hexadecimal

2135

C.2 DESCRIPTION

2136

ASCII is the American Standard Code for Information Interchange. It is a 7-bit code. Many 8-bit codes (e.g., ISO 8859-1) contain ASCII as their lower half. The international counterpart of ASCII is known as ISO 646-IRV.

2139

The following table contains the 128 ASCII characters.

2140

C program '\X' escapes are noted.

2141

	Oct	Dec	Hex	Char		Oct	Dec	Hex	Char
<hr/>									
2143	000	0	00	NUL '\0' (null character)		100	64	40	@
2144	001	1	01	SOH (start of heading)		101	65	41	A
2145	002	2	02	STX (start of text)		102	66	42	B
2146	003	3	03	ETX (end of text)		103	67	43	C
2147	004	4	04	EOT (end of transmission)		104	68	44	D
2148	005	5	05	ENQ (enquiry)		105	69	45	E
2149	006	6	06	ACK (acknowledge)		106	70	46	F
2150	007	7	07	BEL '\a' (bell)		107	71	47	G
2151	010	8	08	BS '\b' (backspace)		110	72	48	H
2152	011	9	09	HT '\t' (horizontal tab)		111	73	49	I
2153	012	10	0A	LF '\n' (new line)		112	74	4A	J
2154	013	11	0B	VT '\v' (vertical tab)		113	75	4B	K
2155	014	12	0C	FF '\f' (form feed)		114	76	4C	L
2156	015	13	0D	CR '\r' (carriage ret)		115	77	4D	M

2157	016	14	0E	S0 (shift out)	116	78	4E	N
2158	017	15	0F	SI (shift in)	117	79	4F	O
2159	020	16	10	DLE (data link escape)	120	80	50	P
2160	021	17	11	DC1 (device control 1)	121	81	51	Q
2161	022	18	12	DC2 (device control 2)	122	82	52	R
2162	023	19	13	DC3 (device control 3)	123	83	53	S
2163	024	20	14	DC4 (device control 4)	124	84	54	T
2164	025	21	15	NAK (negative ack.)	125	85	55	U
2165	026	22	16	SYN (synchronous idle)	126	86	56	V
2166	027	23	17	ETB (end of trans. blk)	127	87	57	W
2167	030	24	18	CAN (cancel)	130	88	58	X
2168	031	25	19	EM (end of medium)	131	89	59	Y
2169	032	26	1A	SUB (substitute)	132	90	5A	Z
2170	033	27	1B	ESC (escape)	133	91	5B	[
2171	034	28	1C	FS (file separator)	134	92	5C	'\\'
2172	035	29	1D	GS (group separator)	135	93	5D]
2173	036	30	1E	RS (record separator)	136	94	5E	^
2174	037	31	1F	US (unit separator)	137	95	5F	-
2175	040	32	20	SPACE	140	96	60	'
2176	041	33	21	!	141	97	61	a
2177	042	34	22	"	142	98	62	b
2178	043	35	23	#	143	99	63	c
2179	044	36	24	\$	144	100	64	d
2180	045	37	25	%	145	101	65	e
2181	046	38	26	&	146	102	66	f
2182	047	39	27	,	147	103	67	g
2183	050	40	28	(150	104	68	h
2184	051	41	29)	151	105	69	i
2185	052	42	2A	*	152	106	6A	j
2186	053	43	2B	+	153	107	6B	k
2187	054	44	2C	,	154	108	6C	l
2188	055	45	2D	-	155	109	6D	m
2189	056	46	2E	.	156	110	6E	n
2190	057	47	2F	/	157	111	6F	o
2191	060	48	30	0	160	112	70	p
2192	061	49	31	1	161	113	71	q
2193	062	50	32	2	162	114	72	r
2194	063	51	33	3	163	115	73	s
2195	064	52	34	4	164	116	74	t
2196	065	53	35	5	165	117	75	u
2197	066	54	36	6	166	118	76	v
2198	067	55	37	7	167	119	77	w
2199	070	56	38	8	170	120	78	x
2200	071	57	39	9	171	121	79	y
2201	072	58	3A	:	172	122	7A	z
2202	073	59	3B	;	173	123	7B	{
2203	074	60	3C	<	174	124	7C	
2204	075	61	3D	=	175	125	7D	}
2205	076	62	3E	>	176	126	7E	~
2206	077	63	3F	?	177	127	7F	DEL

2207 **C.2.1 Tables**

2208 For convenience, below are more compact tables in hex and decimal.

2209	2 3 4 5 6 7	30 40 50 60 70 80 90 100 110 120
2210	<hr/>	
2211	0: 0 @ P ' p	0: (2 < F P Z d n x
2212	1: ! 1 A Q a q	1:) 3 = G Q [e o y
2213	2: " 2 B R b r	2: * 4 > H R \ f p z
2214	3: # 3 C S c s	3: ! + 5 ? I S] g q {
2215	4: \$ 4 D T d t	4: " , 6 @ J T ^ h r
2216	5: % 5 E U e u	5: # - 7 A K U _ i s }
2217	6: & 6 F V f v	6: \$. 8 B L V ' j t ~
2218	7: ' 7 G W g w	7: % / 9 C M W a k u DEL
2219	8: (8 H X h x	8: & 0 : D N X b l v
2220	9:) 9 I Y i y	9: ' 1 ; E O Y c m w
2221	A: * : J Z j z	
2222	B: + ; K [k {	
2223	C: , < L \ l	
2224	D: - = M] m }	
2225	E: . > N ^ n ~	
2226	F: / ? O _ o DEL	

2227 **C.3 NOTES**2228 **C.3.1 History**

2229 An ascii manual page appeared in Version 7 of AT&T UNIX.

2230 On older terminals, the underscore code is displayed as a left arrow, called backarrow, the caret is
2231 displayed as an up-arrow and the vertical bar has a hole in the middle.2232 Uppercase and lowercase characters differ by just one bit and the ASCII character 2 differs from the
2233 double quote by just one bit, too. That made it much easier to encode characters mechanically or
2234 with a non-microcontroller-based electronic keyboard and that pairing was found on old teletypes.2235 The ASCII standard was published by the United States of America Standards Institute (USASI) in
2236 1968.2237 **C.4 COLOPHON**2238 This page is part of release 4.04 of the Linux man-pages project. A description of the project,
2239 information about reporting bugs, and the latest version of this page, can be found at <http://www.kernel.org/doc/man-pages/>.

2241

Appendix D

2242

Attribution 4.0 International

2243

Creative Commons Corporation ("Creative Commons") is not a law firm and does not provide legal services or legal advice. Distribution of Creative Commons public licenses does not create a lawyer-client or other relationship. Creative Commons makes its licenses and related information available on an "as-is" basis. Creative Commons gives no warranties regarding its licenses, any material licensed under their terms and conditions, or any related information. Creative Commons disclaims all liability for damages resulting from their use to the fullest extent possible.

2248

Using Creative Commons Public Licenses

2249

Creative Commons public licenses provide a standard set of terms and conditions that creators and other rights holders may use to share original works of authorship and other material subject to copyright and certain other rights specified in the public license below. The following considerations are for informational purposes only, are not exhaustive, and do not form part of our licenses.

2253

Considerations for licensors: Our public licenses are intended for use by those authorized to give the public permission to use material in ways otherwise restricted by copyright and certain other rights. Our licenses are irrevocable. Licensors should read and understand the terms and conditions of the license they choose before applying it. Licensors should also secure all rights necessary before applying our licenses so that the public can reuse the material as expected. Licensors should clearly mark any material not subject to the license. This includes other CC-licensed material, or material used under an exception or limitation to copyright. More considerations for licensors: http://wiki.creativecommons.org/Considerations_for_licensors

2259

Considerations for the public: By using one of our public licenses, a licensor grants the public permission to use the licensed material under specified terms and conditions. If the licensor's permission is not necessary for any reason—for example, because of any applicable exception or limitation to copyright—then that use is not regulated by the license. Our licenses grant only permissions under copyright and certain other rights that a licensor has authority to grant. Use of the licensed material may still be restricted for other reasons, including because others have copyright or other rights in the material. A licensor may make special requests, such as asking that all changes be marked or described. Although not required by our licenses, you are encouraged to respect those requests where reasonable. More considerations for the public: http://wiki.creativecommons.org/Considerations_for_licensees

2268

Creative Commons Attribution 4.0 International Public License

2269

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

2274

Section 1. Definitions

2275

- a. Adapted Material means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.
- b. Adapter's License means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted

2281 Material in accordance with the terms and conditions of this Public License.

2282 c. Copyright and Similar Rights means copyright and/or similar rights closely related to copyright including, without
2283 limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the
2284 rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not
2285 Copyright and Similar Rights.

2286 d. Effective Technological Measures means those measures that, in the absence of proper authority, may not be circumvented
2287 under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or
2288 similar international agreements.

2289 e. Exceptions and Limitations means fair use, fair dealing, and/or any other exception or limitation to Copyright and
2290 Similar Rights that applies to Your use of the Licensed Material.

2291 f. Licensed Material means the artistic or literary work, database, or other material to which the Licensor applied this
2292 Public License.

2293 g. Licensed Rights means the rights granted to You subject to the terms and conditions of this Public License, which are
2294 limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has
2295 authority to license.

2296 h. Licensor means the individual(s) or entity(ies) granting rights under this Public License.

2297 i. Share means to provide material to the public by any means or process that requires permission under the Licensed
2298 Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or im-
2299 portation, and to make material available to the public including in ways that members of the public may access the
2300 material from a place and at a time individually chosen by them.

2301 j. Sui Generis Database Rights means rights other than copyright resulting from Directive 96/9/EC of the European
2302 Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as
2303 well as other essentially equivalent rights anywhere in the world.

2304 k. You means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding
2305 meaning.

2306 Section 2. Scope

2307 a. License grant.

2308 1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-
2309 free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material
2310 to:
2311 a. reproduce and Share the Licensed Material, in whole or in part; and
2312 b. produce, reproduce, and Share Adapted Material.

2313 2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use,
2314 this Public License does not apply, and You do not need to comply with its terms and conditions.

2315 3. Term. The term of this Public License is specified in Section 6(a).

2316 4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights
2317 in all media and formats whether now known or hereafter created, and to make technical modifications necessary
2318 to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making
2319 technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to
2320 circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications
2321 authorized by this Section 2(a) (4) never produces Adapted Material.

2322 5. Downstream recipients.
2323 a. Offer from the Licensor – Licensed Material. Every recipient of the Licensed Material automatically receives
2324 an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public
2325 License.
2326 b. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on,
2327 or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the
2328 Licensed Rights by any recipient of the Licensed Material.

2329 6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply
2330 that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted
2331 official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

2332 b. Other rights.

2333 1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy,
2334 and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to
2335 assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed
2336 Rights, but not otherwise.

2337 2. Patent and trademark rights are not licensed under this Public License.

2338 3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed
2339 Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory
2340 licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

2341 Section 3. License Conditions

2342 Your exercise of the Licensed Rights is expressly made subject to the following conditions.

2343 a. Attribution.

2344 1. If You Share the Licensed Material (including in modified form), You must:

2345 a. retain the following if it is supplied by the Licensor with the Licensed Material:

2346 i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution,
2347 in any reasonable manner requested by the Licensor (including by pseudonym if designated);

2348 ii. a copyright notice;

2349 iii. a notice that refers to this Public License;

2350 iv. a notice that refers to the disclaimer of warranties;

2351 v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

2352 b. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

2353 c. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or
2354 hyperlink to, this Public License.

2355 2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and
2356 context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions
2357 by providing a URI or hyperlink to a resource that includes the required information.

2358 3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent
2359 reasonably practicable.

2360 4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the
2361 Adapted Material from complying with this Public License.

2362 Section 4. Sui Generis Database Rights

2363 Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

2364 a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial
2365 portion of the contents of the database;

2366 b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database
2367 Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted
2368 Material; and

2369 c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the
2370 database.

2371 For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where
2372 the Licensed Rights include other Copyright and Similar Rights.

2373 Section 5. Disclaimer of Warranties and Limitation of Liability

2374 a. UNLESS OTHERWISE SEPARATELY UNDERTAKEN BY THE LICENSOR, TO THE EXTENT POSSIBLE, THE
2375 LICENSOR OFFERS THE LICENSED MATERIAL AS-IS AND AS-AVAILABLE, AND MAKES NO REPRESENTA-
2376 TIONS OR WARRANTIES OF ANY KIND CONCERNING THE LICENSED MATERIAL, WHETHER EXPRESS,
2377 IMPLIED, STATUTORY, OR OTHER. THIS INCLUDES, WITHOUT LIMITATION, WARRANTIES OF TITLE,
2378 MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, ABSENCE OF LA-
2379 TENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OR ABSENCE OF ERRORS, WHETHER OR
2380 NOT KNOWN OR DISCOVERABLE. WHERE DISCLAIMERS OF WARRANTIES ARE NOT ALLOWED IN FULL
2381 OR IN PART, THIS DISCLAIMER MAY NOT APPLY TO YOU.

2382 b. TO THE EXTENT POSSIBLE, IN NO EVENT WILL THE LICENSOR BE LIABLE TO YOU ON ANY LEGAL
2383 THEORY (INCLUDING, WITHOUT LIMITATION, NEGLIGENCE) OR OTHERWISE FOR ANY DIRECT, SPE-
2384 CIAL, INDIRECT, INCIDENTAL, CONSEQUENTIAL, PUNITIVE, EXEMPLARY, OR OTHER LOSSES, COSTS,
2385 EXPENSES, OR DAMAGES ARISING OUT OF THIS PUBLIC LICENSE OR USE OF THE LICENSED MATERIAL,
2386 EVEN IF THE LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSSES, COSTS, EXPENSES,
2387 OR DAMAGES. WHERE A LIMITATION OF LIABILITY IS NOT ALLOWED IN FULL OR IN PART, THIS LIMI-
2388 TATION MAY NOT APPLY TO YOU.

2389 c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the
2390 extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

2391 Section 6. Term and Termination

2392 a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to
2393 comply with this Public License, then Your rights under this Public License terminate automatically.

2394 b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

2395 1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the
2396 violation; or

2397 2. upon express reinstatement by the Licenser.

2398 For the avoidance of doubt, this Section 6(b) does not affect any right the Licenser may have to seek remedies for Your
2399 violations of this Public License.

2400 c. For the avoidance of doubt, the Licenser may also offer the Licensed Material under separate terms or conditions or
2401 stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

2402 d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

2403 Section 7. Other Terms and Conditions

2404 a. The Licenser shall not be bound by any additional or different terms or conditions communicated by You unless expressly
2405 agreed.

2406 b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from
2407 and independent of the terms and conditions of this Public License.

2408 Section 8. Interpretation

2409 a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or
2410 impose conditions on any use of the Licensed Material that could lawfully be made without permission under this
2411 Public License.

2412 b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed
2413 to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from
2414 this Public License without affecting the enforceability of the remaining terms and conditions.

2415 c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed
2416 to by the Licenser.

2417 d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and
2418 immunities that apply to the Licenser or You, including from the legal processes of any jurisdiction or authority.

2420 Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of
2421 its public licenses to material it publishes and in those instances will be considered the Licenser. The text of the Creative
2422 Commons public licenses is dedicated to the public domain under the CC0 Public Domain Dedication. Except for the limited
2423 purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the
2424 Creative Commons policies published at <http://creativecommons.org/policies>, Creative Commons does not authorize the use
2425 of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent
2426 including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other
2427 arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph
2428 does not form part of the public licenses.

2429 Creative Commons may be contacted at <http://creativecommons.org>.

Bibliography

2430

2431 [1] RISC-V Foundation, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document*
2432 *Version 2.2*, 5 2017. Editors Andrew Waterman and Krste Asanović. [iv](#), [3](#), [4](#), [16](#), [25](#), [27](#), [32](#), [59](#),
2433 [82](#)

2434 [2] D. Patterson and A. Waterman, *The RISC-V Reader: An Open Architecture Atlas*. Strawberry
2435 Canyon, 11 2017. ISBN: 978-0999249116. [iv](#)

2436 [3] D. Patterson and J. Hennessy, *Computer Organization and Design RISC-V Edition: The Hard-*
2437 *ware Software Interface*. Morgan Kaufmann, 4 2017. ISBN: 978-0128122754. [iv](#), [27](#)

2438 [4] W. F. Decker, “A modern approach to teaching computer organization and assembly language
2439 programming,” *SIGCSE Bull.*, vol. 17, pp. 38–44, 12 1985. [iv](#)

2440 [5] Texas Instruments, *SN54190, SN54191, SN54LS190, SN54LS191, SN74190, SN74191,*
2441 *SN74LS190, SN74LS191 Synchronous Up/Down Counters With Down/Up Mode Control*, 3 1988.
2442 [iv](#)

2443 [6] Texas Instruments, *SN54154, SN74154 4-line to 16-line Decoders/Demultiplexers*, 12 1972. [iv](#)

2444 [7] Intel, *MCS-85 User’s Manual*, 9 1978. [iv](#)

2445 [8] Radio Shack, *TRS-80 Editor/Assembler Operation and Reference Manual*, 1978. [iv](#)

2446 [9] Motorola, *MC68000 16-bit Microprocessor User’s Manual*, 2nd ed., 1 1980. MC68000UM(AD2).
2447 [iv](#)

2448 [10] R. A. Overbeek and W. E. Singletary, *Assembler Language With ASSIST*. Science Research
2449 Associates, Inc., 2nd ed., 1983. [iv](#)

2450 [11] IBM, *IBM System/370 Principles of Operation*, 7th ed., 3 1980. [iv](#)

2451 [12] IBM, *OS/VS-DOS/VSE-VM/370 Assembler Language*, 6th ed., 3 1979. [iv](#)

2452 [13] “Definition of subtrahend.” www.mathsisfun.com/definitions/subtrahend.html. Accessed: 2018-
2453 06-02. [17](#)

2454 [14] D. Cohen, “[IEN 137, On Holy Wars and a Plea for Peace](#),” Apr. 1980. This note discusses the
2455 Big-Endian/Little-Endian byte/bit-order controversy, but did not settle it. A decade later, David
2456 V. James in “[Multiplexed Buses: The Endian Wars Continue](#)”, *IEEE Micro*, **10**(3), 9–21 (1990)
2457 continued the discussion. [22](#)

2458 [15] “[Ieee standard for floating-point arithmetic](#),” *IEEE Std 754-2019 (Revision of IEEE 754-2008)*,
2459 pp. 1–84, 2019. [63](#), [64](#)

2460 [16] RISC-V Foundation, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture,*
2461 *Document Version 1.10*, 5 2017. Editors Andrew Waterman and Krste Asanović.

2462 [17] P. Dabbelt, S. O'Rear, K. Cheng, A. Waterman, M. Clark, A. Bradbury, D. Horner, M. Nordlund,
2463 and K. Merker, *RISC-V ELF psABI specification*, 2017.

2464 [18] R. M. Stallman and the GCC Developer Community, *Using the GNU Compiler Collection (For*
2465 *GCC version 7.3.0)*. Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA
2466 02110-1301 USA: GNU Press, 2017.

2467 [19] National Semiconductor Corporation, *Series 32000 Databook*, 1986.

2468

2469 Glossary

2470 **address** A numeric value used to uniquely identify each [byte](#) of main memory. [2](#), [77](#)

2471 **alignment** Refers to a range of numeric values that begin at a multiple of some number. Primarily
2472 used when referring to a memory address. For example an alignment of two refers to one or
2473 more addresses starting at even address and continuing onto subsequent adjacent, increasing
2474 memory addresses. [26](#), [77](#)

2475 **ASCII** American Standard Code for Information Interchange. See [Appendix C](#). [21](#), [77](#)

2476 **big-endian** A number format where the most significant values are printed to the left of the lesser
2477 significant values. This is the method that everyone uses to write decimal numbers every day.
2478 [23](#), [30](#), [31](#), [77](#), [79](#)

2479 **binary** Something that has two parts or states. In computing these two states are represented by
2480 the numbers one and zero or by the conditions true and false and can be stored in one **bit**. [1](#), [3](#),
2481 [77](#), [78](#), [79](#)

2482 **bit** One binary digit. [3](#), [6](#), [10](#), [77](#), [78](#), [79](#)

2483 **byte** A [binary](#) value represented by 8 [bits](#). [2](#), [6](#), [77](#), [78](#), [79](#)

2484 **CPU** Central Processing Unit. [1](#), [2](#), [77](#)

2485 **doubleword** A [binary](#) value represented by 64 [bits](#). [77](#)

2486 **exception** An error encountered by the CPU while executing an instruction that can not be com-
2487 pleted. [27](#), [77](#)

2488 **fullword** A [binary](#) value represented by 32 [bits](#). [6](#), [77](#)

2489 **halfword** A [binary](#) value represented by 16 [bits](#). [6](#), [22](#), [77](#)

2490 **hart** Hardware Thread. [3](#), [77](#)

2491 **hexadecimal** A base-16 numbering system whose digits are 0123456789abcdef. The hex digits ([hits](#))
2492 are not case-sensitive. [30](#), [31](#), [77](#), [78](#)

2493 **high order bits** Some number of [MSBs](#). [77](#)

2494 **hit** One [hexadecimal](#) digit. [10](#), [12](#), [77](#), [78](#), [79](#)

2495 **ISA** Instruction Set Architecture. [3](#), [4](#), [77](#)

2496 **LaTeX** Is a mark up language specially suited for scientific documents. [77](#)

2497 **little-endian** A number format where the least significant values are printed to the left of the more
2498 significant values. This is the opposite ordering that everyone learns in grade school when
2499 learning how to count. For example, the [big-endian](#) number written as “1234” would be written
2500 in little endian form as “4321”. [24](#), [77](#)

2501 **low order bits** Some number of [LSBs](#). [77](#)

2502 **LSB** Least Significant Bit. [10](#), [12](#), [22](#), [44](#), [48](#), [54](#), [56](#), [77](#), [79](#)

2503 **machine language** The instructions that are executed by a CPU that are expressed in the form of
2504 [binary](#) values. [1](#), [77](#)

2505 **mnemonic** A method used to remember something. In the case of assembly language, each machine
2506 instruction is given a name so the programmer need not memorize the binary values of each
2507 machine instruction. [1](#), [77](#)

2508 **MSB** Most Significant Bit. [10](#), [12](#), [13](#), [19](#), [20](#), [22](#), [44](#), [45](#), [77](#), [78](#)

2509 **nybble** Half of a [byte](#) is a *nybble* (sometimes spelled nibble.) Another word for [hit](#). [10](#), [77](#)

2510 **overflow** The situation where the result of an addition or subtraction operation is approaching pos-
2511 itive or negative infinity and exceeds the number of bits allotted to contain the result. This is
2512 typically caused by high-order truncation. [64](#), [77](#)

2513 **place value** the numerical value that a digit has as a result of its *position* within a number. For
2514 example, the digit 2 in the decimal number 123 is in the ten’s place and its place value is 20. [9](#),
2515 [10](#), [11](#), [23](#), [24](#), [77](#)

2516 **program** A ordered list of one or more instructions. [1](#), [77](#)

2517 **quadword** A [binary](#) value represented by 128 [bits](#). [77](#)

2518 **RAM** Random Access Memory. [2](#), [77](#)

2519 **register** A unit of storage inside a CPU with the capacity of [XLEN](#) [bits](#). [2](#), [77](#), [79](#)

2520 **ROM** Read Only Memory. [2](#), [77](#)

2521 **RV32** Short for RISC-V 32. The number 32 refers to the [XLEN](#). [77](#)

2522 **RV64** Short for RISC-V 64. The number 64 refers to the [XLEN](#). [77](#)

2523 **rvddt** A RV32I simulator and debugging tool inspired by the simplicity of the Dynamic Debugging
2524 Tool (ddt) that was part of the CP/M operating system. [21](#), [29](#), [77](#)

2525 **thread** An stream of instructions. When plural, it is used to refer to the ability of a CPU to execute
2526 multiple instruction streams at the same time. [3](#), [77](#)

2527 **underflow** The situation where the result of an addition or subtraction operation is approaching
2528 zero and exceeds the number of bits allotted to contain the result. This is typically caused by
2529 low-order truncation. [64](#), [77](#)

2530 **XLEN** The number of bits a RISC-V x integer [register](#) (such as x0). For RV32 XLEN=32, RV64
2531 XLEN=64 and so on. [49](#), [50](#), [52](#), [56](#), [57](#), [77](#), [79](#)

Index

2532	A	2571	signed, 17
2533	ALU, 3	2572	unsigned, 16
2534	ASCII, 26, 69		
2535	ASCIIZ, 26	2573	
2536	B	2574	register, 2, 3
2537	big-endian, 23	2575	RV32, 44
2538	C	2576	RV32A, 4
2539	carry, 15	2577	RV32C, 4
2540	CPU, 2	2578	RV32D, 4
2541	F	2579	RV32F, 4
2542	Full Adder, 13	2580	RV32G, 4
2543	H	2581	RV32I, 4
2544	hart, 3	2582	RV32M, 4
2545	I	2583	RV32Q, 4
2546	imm_b, 57	2584	rvddt, 29
2547	imm_i, 53	2585	
2548	imm_j, 50	2586	S
2549	imm_s, 56	2587	shamt_i, 53
2550	imm_u, 49	2588	sign extension, 19
2551	Instruction	2589	
2552	addi, 33		T
2553	ebreak, 32		truncation, 15, 18
2554	mv, 35		
2555	nop, 33		
2556	instruction cycle, 4		
2557	instruction decode, 5		
2558	instruction execute, 5		
2559	instruction fetch, 5		
2560	ISA, 4		
2561	L		
2562	Least significant bit, 10		
2563	little-endian, 24		
2564	LSB, <i>see</i> Least significant bit		
2565	M		
2566	Most significant bit, 10		
2567	MSB, <i>see</i> Most significant bit		
2568	O		
2569	objdump, 34		
2570	overflow, 15		

RV32I Reference Cards

Usage Template	Type	Description	Detailed Description
add rd, rs1, rs2	R	Add	$rd \leftarrow rs1 + rs2, pc \leftarrow pc+4$
addi rd, rs1, imm	I	Add Immediate	$rd \leftarrow rs1 + imm_i, pc \leftarrow pc+4$
and rd, rs1, rs2	R	And	$rd \leftarrow rs1 \wedge rs2, pc \leftarrow pc+4$
andi rd, rs1, imm	I	And Immediate	$rd \leftarrow rs1 \wedge imm_i, pc \leftarrow pc+4$
auipc rd, imm	U	Add Upper Immediate to PC	$rd \leftarrow pc + imm_u, pc \leftarrow pc+4$
beq rs1, rs2, pcrel_13	B	Branch Equal	$pc \leftarrow pc + ((rs1 == rs2) ? imm_b : 4)$
bge rs1, rs2, pcrel_13	B	Branch Greater or Equal	$pc \leftarrow pc + ((rs1 >= rs2) ? imm_b : 4)$
bgeu rs1, rs2, pcrel_13	B	Branch Greater or Equal Unsigned	$pc \leftarrow pc + ((rs1 >= rs2) ? imm_b : 4)$
blt rs1, rs2, pcrel_13	B	Branch Less Than	$pc \leftarrow pc + ((rs1 < rs2) ? imm_b : 4)$
bltu rs1, rs2, pcrel_13	B	Branch Less Than Unsigned	$pc \leftarrow pc + ((rs1 < rs2) ? imm_b : 4)$
bne rs1, rs2, pcrel_13	B	Branch Not Equal	$pc \leftarrow pc + ((rs1 != rs2) ? imm_b : 4)$
csrrw rd, csr, rs1	I	Atomic Read/Write	$rd \leftarrow csr, csr \leftarrow rs1, pc \leftarrow pc+4$
csrrs rd, csr, rs1	I	Atomic Read and Set	$rd \leftarrow csr, csr \leftarrow csr \vee rs1, pc \leftarrow pc+4$
csrrc rd, csr, rs1	I	Atomic Read and Clear	$rd \leftarrow csr, csr \leftarrow csr \wedge \sim rs1, pc \leftarrow pc+4$
csrrwi rd, csr, zimm	I	Atomic Read/Write Immediate	$rd \leftarrow csr, csr \leftarrow zimm, pc \leftarrow pc+4$
csrrsi rd, csr, zimm	I	Atomic Read and Set Immediate	$rd \leftarrow csr, csr \leftarrow csr \vee zimm, pc \leftarrow pc+4$
csrrci rd, csr, zimm	I	Atomic Read and Clear Immediate	$rd \leftarrow csr, csr \leftarrow csr \wedge \sim zimm, pc \leftarrow pc+4$
ecall	I	Environment Call	Transfer Control to Debugger
ebreak	I	Environment Break	Transfer Control to Operating System
jal rd, pcrel_21	J	Jump And Link	$rd \leftarrow pc+4, pc \leftarrow pc+imm_j$
jalr rd, imm(rs1)	I	Jump And Link Register	$rd \leftarrow pc+4, pc \leftarrow (rs1 + imm_i) \& \sim 1$
lb rd, imm(rs1)	I	Load Byte	$rd \leftarrow sx(m8(rs1+imm_i)), pc \leftarrow pc+4$
lbu rd, imm(rs1)	I	Load Byte Unsigned	$rd \leftarrow zx(m8(rs1+imm_i)), pc \leftarrow pc+4$
lh rd, imm(rs1)	I	Load Halfword	$rd \leftarrow sx(m16(rs1+imm_i)), pc \leftarrow pc+4$
lhu rd, imm(rs1)	I	Load Halfword Unsigned	$rd \leftarrow zx(m16(rs1+imm_i)), pc \leftarrow pc+4$
lui rd, imm	U	Load Upper Immediate	$rd \leftarrow imm_u, pc \leftarrow pc+4$
lw rd, imm(rs1)	I	Load Word	$rd \leftarrow sx(m32(rs1+imm_i)), pc \leftarrow pc+4$
or rd, rs1, rs2	R	Or	$rd \leftarrow rs1 \vee rs2, pc \leftarrow pc+4$
ori rd, rs1, imm	I	Or Immediate	$rd \leftarrow rs1 \vee imm_i, pc \leftarrow pc+4$
sb rs2, imm(rs1)	S	Store Byte	$m8(rs1+imm_s) \leftarrow rs2[7:0], pc \leftarrow pc+4$
sh rs2, imm(rs1)	S	Store Halfword	$m16(rs1+imm_s) \leftarrow rs2[15:0], pc \leftarrow pc+4$
sll rd, rs1, rs2	R	Shift Left Logical	$rd \leftarrow rs1 \ll (rs2 \% XLEN), pc \leftarrow pc+4$
slli rd, rs1, shamt	I	Shift Left Logical Immediate	$rd \leftarrow rs1 \ll shamt_i, pc \leftarrow pc+4$
slt rd, rs1, rs2	R	Set Less Than	$rd \leftarrow (rs1 < rs2) ? 1 : 0, pc \leftarrow pc+4$
slti rd, rs1, imm	I	Set Less Than Immediate	$rd \leftarrow (rs1 < imm_i) ? 1 : 0, pc \leftarrow pc+4$
sltiu rd, rs1, imm	I	Set Less Than Immediate Unsigned	$rd \leftarrow (rs1 < imm_i) ? 1 : 0, pc \leftarrow pc+4$
sltu rd, rs1, rs2	R	Set Less Than Unsigned	$rd \leftarrow (rs1 < rs2) ? 1 : 0, pc \leftarrow pc+4$
sra rd, rs1, rs2	R	Shift Right Arithmetic	$rd \leftarrow rs1 \gg (rs2 \% XLEN), pc \leftarrow pc+4$
srai rd, rs1, shamt	I	Shift Right Arithmetic Immediate	$rd \leftarrow rs1 \gg shamt_i, pc \leftarrow pc+4$
srl rd, rs1, rs2	R	Shift Right Logical	$rd \leftarrow rs1 \gg (rs2 \% XLEN), pc \leftarrow pc+4$
srli rd, rs1, shamt	I	Shift Right Logical Immediate	$rd \leftarrow rs1 \gg shamt_i, pc \leftarrow pc+4$
sub rd, rs1, rs2	R	Subtract	$rd \leftarrow rs1 - rs2, pc \leftarrow pc+4$
sw rs2, imm(rs1)	S	Store Word	$m32(rs1+imm_s) \leftarrow rs2[31:0], pc \leftarrow pc+4$
xor rd, rs1, rs2	R	Exclusive Or	$rd \leftarrow rs1 \oplus rs2, pc \leftarrow pc+4$
xori rd, rs1, imm	I	Exclusive Or Immediate	$rd \leftarrow rs1 \oplus imm_i, pc \leftarrow pc+4$

RV32I Base Instruction Set Encoding [1, p. 104]

31	25 24	20 19	15 14	12 11	7	6	0	
	imm[31:12]			rd	0 1 1 0 1 1 1			U-type lui rd,imm
	imm[31:12]			rd	0 0 1 0 1 1 1			U-type auipc rd,imm
	imm[20:10:1 11 19:12]			rd	1 1 0 1 1 1 1			J-type jal rd,pcrel_21
	imm[11:0]	rs1	0 0 0	rd	1 1 0 0 1 1 1			I-type jalr rd,imm(rs1)
imm[12:10:5]	rs2	rs1	0 0 0	imm[4:1 11]	1 1 0 0 0 1 1			B-type beq rs1,rs2,pcrel_13
imm[12:10:5]	rs2	rs1	0 0 1	imm[4:1 11]	1 1 0 0 0 1 1			B-type bne rs1,rs2,pcrel_13
imm[12:10:5]	rs2	rs1	1 0 0	imm[4:1 11]	1 1 0 0 0 1 1			B-type blt rs1,rs2,pcrel_13
imm[12:10:5]	rs2	rs1	1 0 1	imm[4:1 11]	1 1 0 0 0 1 1			B-type bge rs1,rs2,pcrel_13
imm[12:10:5]	rs2	rs1	1 1 0	imm[4:1 11]	1 1 0 0 0 1 1			B-type bltu rs1,rs2,pcrel_13
imm[12:10:5]	rs2	rs1	1 1 1	imm[4:1 11]	1 1 0 0 0 1 1			B-type bgeu rs1,rs2,pcrel_13
imm[11:0]		rs1	0 0 0	rd	0 0 0 0 0 1 1			I-type lb rd,imm(rs1)
imm[11:0]		rs1	0 0 1	rd	0 0 0 0 0 1 1			I-type lh rd,imm(rs1)
imm[11:0]		rs1	0 1 0	rd	0 0 0 0 0 1 1			I-type lw rd,imm(rs1)
imm[11:0]		rs1	1 0 0	rd	0 0 0 0 0 1 1			I-type lbu rd,imm(rs1)
imm[11:0]		rs1	1 0 1	rd	0 0 0 0 0 1 1			I-type lhu rd,imm(rs1)
imm[11:5]	rs2	rs1	0 0 0	imm[4:0]	0 1 0 0 0 1 1			S-type sb rs2,imm(rs1)
imm[11:5]	rs2	rs1	0 0 1	imm[4:0]	0 1 0 0 0 1 1			S-type sh rs2,imm(rs1)
imm[11:5]	rs2	rs1	0 1 0	imm[4:0]	0 1 0 0 0 1 1			S-type sw rs2,imm(rs1)
imm[11:0]		rs1	0 0 0	rd	0 0 1 0 0 1 1			I-type addi rd,rs1,imm
imm[11:0]		rs1	0 1 0	rd	0 0 1 0 0 1 1			I-type slti rd,rs1,imm
imm[11:0]		rs1	0 1 1	rd	0 0 1 0 0 1 1			I-type sltiu rd,rs1,imm
imm[11:0]		rs1	1 0 0	rd	0 0 1 0 0 1 1			I-type xor rd,rs1,imm
imm[11:0]		rs1	1 1 0	rd	0 0 1 0 0 1 1			I-type ori rd,rs1,imm
imm[11:0]		rs1	1 1 1	rd	0 0 1 0 0 1 1			I-type andi rd,rs1,imm
0 0 0 0 0 0 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1			I-type slli rd,rs1,shamt
0 0 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1			I-type srli rd,rs1,shamt
0 1 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1			I-type srai rd,rs1,shamt
0 0 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1			R-type add rd,rs1,rs2
0 1 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1			R-type sub rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1			R-type sll rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 1 0	rd	0 1 1 0 0 1 1			R-type slt rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 1 1	rd	0 1 1 0 0 1 1			R-type sltu rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1			R-type xor rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1			R-type srl rd,rs1,rs2
0 1 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1			R-type sra rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1			R-type or rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1			R-type and rd,rs1,rs2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0 0 0	1 1 1 0 0 1 1			I-type ecall
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	0 0 0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0 0 0	1 1 1 0 0 1 1			I-type ebreak
csr[11:0]		rs1	0 0 1	rd	1 1 1 0 0 1 1			I-type csrrw rd,csr,rs1
csr[11:0]		rs1	0 1 0	rd	1 1 1 0 0 1 1			I-type csrrs rd,csr,rs1
csr[11:0]		rs1	0 1 1	rd	1 1 1 0 0 1 1			I-type csrrc rd,csr,rs1
csr[11:0]		zimm[4:0]	1 0 1	rd	1 1 1 0 0 1 1			I-type csrrwi rd,csr,zimm
csr[11:0]		zimm[4:0]	1 1 0	rd	1 1 1 0 0 1 1			I-type csrrsi rd,csr,zimm
csr[11:0]		zimm[4:0]	1 1 1	rd	1 1 1 0 0 1 1			I-type csrrci rd,csr,zimm

Instruction	Description	Operation	Type	funct7	25[24]	20[19]	15[14]	12[11]	funct3	7[6]	6[5]	opcode	
lui	rd,imm	Load Upper Immediate	U		imm[31:12]				rd	0 1 1 0 1 1 1	0	0	
auipc	rd,imm	Add Upper Immediate to PC	U		imm[31:12]				rd	0 0 1 0 1 1 1	0	0	
jal	rd,imm(rs1)	Jump And Link Register	U		imm[20:11 19:12]				rd	1 1 0 1 1 1 1	0	0	
rs1,rs2,pcrel21	rs1,rs2,pcrel13	Branch Equal	J		imm[11:0]				rs1	0 0 0	rd	1 1 0 0 1 1 1	
beq	rs1,rs2,pcrel13	Branch Not Equal	I	imm[12 10:5]				rs2	0 0 0	imm[4:11]	1 1 0 0 0 1 1	0	
bne	rs1,rs2,pcrel13	Branch Less Than	B	imm[12 10:5]				rs1	0 0 1	imm[4:11]	1 1 0 0 0 1 1	0	
blt	rs1,rs2,pcrel13	Branch Greater or Equal	B	imm[12 10:5]				rs2	1 0 0	imm[4:11]	1 1 0 0 0 1 1	0	
bge	rs1,rs2,pcrel13	Branch Less Than Unsigned	B	imm[12 10:5]				rs2	1 0 1	imm[4:11]	1 1 0 0 0 1 1	0	
bltu	rs1,rs2,pcrel13	Branch Greater or Equal Unsigned	B	imm[12 10:5]				rs1	1 1 0	imm[4:11]	1 1 0 0 0 1 1	0	
bgeu	rs1,rs2,pcrel13	Branch Less Than Unsigned	B	imm[12 10:5]				rs2	1 1 1	imm[4:11]	1 1 0 0 0 1 1	0	
lb	rd,imm(rs1)	Load Byte	I	imm[11:0]				rs1	0 0 0	rd	0 0 0 0 0 1 1	0	
lh	rd,imm(rs1)	Load Halfword	I	imm[11:0]				rs1	0 0 1	rd	0 0 0 0 0 1 1	0	
lw	rd,imm(rs1)	Load Word	I	imm[11:0]				rs2	0 1 0	rd	0 0 0 0 0 1 1	0	
lbu	rd,imm(rs1)	Load Byte Unsigned	I	imm[11:0]				rs1	1 0 0	rd	0 0 0 0 0 1 1	0	
lhu	rd,imm(rs1)	Load Halfword Unsigned	I	imm[11:0]				rs1	1 0 1	rd	0 0 0 0 0 1 1	0	
sb	rs2,imm(rs1)	Store Byte	S	imm[11:5]				rs2	0 0 0	imm[4:0]	0 1 0 0 0 1 1	0	
sh	rs2,imm(rs1)	Store Halfword	S	imm[11:5]				rs1	0 0 1	imm[4:0]	0 1 0 0 0 1 1	0	
sw	rs2,imm(rs1)	Store Word	S	imm[11:5]				rs2	0 1 0	imm[4:0]	0 1 0 0 0 1 1	0	
addi	rd,rs1,imm	Add Immediate	I	imm[11:0]				rs1	0 0 0	rd	0 0 1 0 0 1 1	0	
slti	rd,rs1,imm	Set Less Than Immediate	I	imm[11:0]				rs2	0 1 0	rd	0 0 1 0 0 1 1	0	
sltiu	rd,rs1,imm	Set Less Than Immediate Unsigned	I	imm[11:0]				rs1	0 1 1	rd	0 0 1 0 0 1 1	0	
xori	rd,rs1,imm	Exclusive Or Immediate	I	imm[11:0]				rs1	1 0 0	rd	0 0 1 0 0 1 1	0	
ori	rd,rs1,imm	Or Immediate	I	imm[11:0]				rs2	1 1 0	rd	0 0 1 0 0 1 1	0	
andi	rd,rs1,imm	And Immediate	I	imm[11:0]				rs1	1 1 1	rd	0 0 1 0 0 1 1	0	
slli	rd,rs1,shamt	Shift Left Logical Immediate	I	0 0 0 0 0 0 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1	0	0		
srli	rd,rs1,shamt	Shift Right Logical Immediate	I	0 0 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	0	0		
srai	rd,rs1,shamt	Shift Right Arithmetic Immediate	I	0 1 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	0	0		
add	rd,rs1,rs2	Add	R	0 0 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1	0	0		
sub	rd,rs1,rs2	Subtract	R	0 1 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1	0	0		
sll	rd,rs1,rs2	Shift Left Logical	R	0 0 0 0 0 0 0	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1	0	0		
slt	rd,rs1,rs2	Set Less Than	R	0 0 0 0 0 0 0	rs2	rs1	0 1 0	rd	0 1 1 0 0 1 1	0	0		
sltu	rd,rs1,rs2	Set Less Than Unsigned	R	0 0 0 0 0 0 0	rs2	rs1	0 1 1	rd	0 1 1 0 0 1 1	0	0		
xor	rd,rs1,rs2	Exclusive Or	R	0 0 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1	0	0		
srl	rd,rs1,rs2	Shift Right Logical	R	0 0 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	0	0		
sra	rd,rs1,rs2	Shift Right Arithmetic	R	0 1 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	0	0		
or	rd,rs1,rs2	Or	R	0 0 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1	0	0		
and	rd,rs1,rs2	And	R	0 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1	0	0		
ecall		Trap to Debugger	I	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 1 1 0 0 1 1	0	0	
ebreak		Trap to Operating System	I	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 1 1 0 0 1 1	0	0	
csrrw	rd,csr,rs1	Atomic Read/Write	I	csr[11:0]				rs1	0 0 1	rd	1 1 1 0 0 1 1	0	0
csrrs	rd,csr,rs1	Atomic Read and Set	I	csr[11:0]				rs1	0 1 0	rd	1 1 1 0 0 1 1	0	0
csrrc	rd,csr,rs1	Atomic Read and Clear	I	csr[11:0]				rs1	0 1 1	rd	1 1 1 0 0 1 1	0	0
csrrwi	rd,csr,zimm	Atomic Read/Write Immediate	I	zimm[4:0]	1 0 1	rd	1 1 1 0 0 1 1	0	0	rd	1 1 1 0 0 1 1	0	0
csrrsi	rd,csr,zimm	Atomic Read and Set Immediate	I	zimm[4:0]	1 1 0	rd	1 1 1 0 0 1 1	0	0	rd	1 1 1 0 0 1 1	0	0
csrrci	rd,csr,zimm	Atomic Read and Clear Immediate	I	zimm[4:0]	1 1 1	rd	1 1 1 0 0 1 1	0	0	rd	1 1 1 0 0 1 1	0	0

