RISC-V Assembly Language Programming

(Draft v0.1-19-g15586ff)

John Winans jwinans@niu.edu

 $\mathrm{May}\ 6,\ 2018$

Copyright © 2018 John Winans

This document is made available under a Creative Commons Attribution 4.0 International License. See Appendix A for more information.

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

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

 ARM^\circledR is a registered trademark of ARM Limited in the EU and other countries.

▶ Fix Me:

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

Contents

Pı	efac	e		vi
1	Intr	oducti	ion	1
	1.1	The D	rigital Computer	1
		1.1.1	Storage Systems	1
			1.1.1.1 Volatile Storage	1
			1.1.1.2 Non-Volatile Storage	2
		1.1.2	CPU	2
		1.1.3	Peripherals	2
	1.2	Instru	ction Set Architecture	2
		1.2.1	RV Base Modules	3
		1.2.2	Extension Modules	3
	1.3	An Ex	cample Computer	3
	1.4	Execu	ting a Program	3
		1.4.1	Instruction Fetch	4
		1.4.2	Instruction Execute	4
		1.4.3	A Sample Program Source Listing	5
		1.4.4	Running a Program With rvddt	6
2	Nur	nber S	Systems	9
	2.1	Intege	rs	9
		2.1.1	Converting Between Bases	11
			2.1.1.1 From Binary to Decimal	11
			2.1.1.2 From Binary to Hexadecimal	11
			2.1.1.3 From Hexadecimal to Binary	11
			2.1.1.4 From Decimal to Binary	12
			2.1.1.5 From Decimal to Hex	12
		2.1.2	Addition of Binary Numbers	12
		2.1.3	Signed Numbers	13
			2.1.3.1 Converting between Positive and Negative	13

		2.1.4	Subtraction of Binary Numbers	14
		2.1.5	Truncation and Overflow	15
		2.1.6	Logical/Boolean Functions	15
			2.1.6.1 NOT	15
			2.1.6.2 AND	15
			2.1.6.3 OR	16
			2.1.6.4 XOR	16
	2.2	IEEE-	754 Floating Point Number Representation	16
		2.2.1	Floating Point Number Accuracy	18
			2.2.1.1 Powers Of Two	18
			2.2.1.2 Clean Decimal Numbers	18
			2.2.1.3 Accumulation of Error	19
		2.2.2	Reducing Error Accumulation	20
3	The	DISC	S-V GNU Toolchain	22
J	1116	itisc	- V GIVO Toolcham	2 2
4	RV	32 Mac	chine Instructions	24
	4.1	Introd	uction	24
	4.2	Conve	ntions and Terminology	
		4.2.1	XLEN	24
		4.2.2	sx(val)	24
		4.2.3	zx(val)	25
		4.2.4	$\operatorname{zr}(\operatorname{val})$	25
		4.2.5	Sign Entended Left and Zero Extend Right	25
		4.2.6	$m8(addr) \ \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots$	26
		4.2.7	$m16(addr)\dots$	
		4.2.8	$m32(addr)\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$	27
		4.2.9	$m64(addr)\dots$	27
		4.2.10	$m128(\mathrm{addr})\ \dots \dots$	27
		4.2.11	.+offset	27
		4.2.12	offset	27
		4.2.13	$pc \ \dots $	27
		4.2.14	$\mathbf{rd} \ \ldots \ldots$	28
		4.2.15	rs1	28
		4.2.16	$rs2 \ldots \ldots$	28
		4.2.17	imm	28
		4.2.18	$rsN[h:l] \dots \dots \dots \dots \dots \dots \dots \dots \dots $	28
	4.3	Addre	ssing Modes	28
	4.4	Instru	ction Encoding Formats	28

	4.4.1	U Type		29
	4.4.2	J Type	. ;	30
	4.4.3	R Type	. ;	31
	4.4.4	I Type	. ;	31
	4.4.5	S Type	. ;	31
	4.4.6	B Type	. ;	31
	4.4.7	CPU Registers	. ;	32
4.5	memor	ry	. ;	32
4.6	RV32I	Base Instruction Set	. ;	33
	4.6.1	LUI rd, imm	. ;	33
	4.6.2	AUIPC rd, imm	. ;	34
	4.6.3	JAL rd, imm	. ;	34
	4.6.4	JALR rd, rs1, imm	. ;	35
	4.6.5	BEQ rs1, rs2, imm	. ;	36
	4.6.6	BNE rs1, rs2, imm	. ;	37
	4.6.7	BLT rs1, rs2, imm	. ;	37
	4.6.8	BGE rs1, rs2, imm	. ;	38
	4.6.9	BLTU rs1, rs2, imm	. :	38
	4.6.10	BGEU rs1, rs2, imm	. :	38
	4.6.11	LB rd, imm(rs1) \dots	. ;	39
	4.6.12	LH rd, imm(rs1)	. ;	39
	4.6.13	LW rd, imm(rs1)	. ;	39
	4.6.14	LBU rd, imm(rs1)	. 4	40
	4.6.15	LHU rd, imm(rs1)	. 4	40
	4.6.16	SB rs2, imm(rs1)	. 4	41
	4.6.17	SH rs2, imm(rs1)	. 4	41
	4.6.18	SW rs2, imm(rs1)	. 4	41
	4.6.19	ADDI rd, rs1, imm	. 4	42
	4.6.20	SLTI rd, rs1, imm	. 4	42
	4.6.21	SLTIU rd, rs1, imm	. 4	43
	4.6.22	XORI rd, rs1, imm	. 4	43
	4.6.23	ORI rd, rs1, imm	. 4	44
	4.6.24	ANDI rd, rs1, imm	. 4	44
	4.6.25	SLLI rd, rs1, shamt	. 4	45
	4.6.26	SRLI rd, rs1, shamt	. 4	45
	4.6.27	SRAI rd, rs1, shamt	. 4	45
	4.6.28	ADD rd, rs1, rs2	. 4	46
	4.6.29	SUB rd, rs1, rs2	. 4	46

	4.6.30 SLL rd, rs1, rs2	 47
	4.6.31 SLT rd, rs1, rs2	 47
	4.6.32 SLTU rd, rs1, rs2	 48
	4.6.33 XOR rd, rs1, rs2	 48
	4.6.34 SRL rd, rs1, rs2	 49
	4.6.35 SRA rd, rs1, rs2	 49
	4.6.36 OR rd, rs1, rs2	 50
	4.6.37 AND rd, rs1, rs2	 50
	4.6.38 FENCE predecessor, successor	 51
	4.6.39 FENCE.I	 51
	4.6.40 ECALL	 52
	4.6.41 EBREAK	 52
	4.6.42 CSRRW rd, csr, rs1	 52
	4.6.43 CSRRS rd, csr, rs1	 52
	4.6.44 CSRRC rd, csr, rs1	 52
	4.6.45 CSRRWI rd, csr, imm	 53
	4.6.46 CSRRSI rd, csr, rs1	 53
	4.6.47 CSRRCI rd, csr, rs1	 53
4.7	7 RV32M Standard Extension	 53
	4.7.1 MUL rd, rs1, rs2	 53
	4.7.2 MULH rd, rs1, rs2	 54
	4.7.3 MULHS rd, rs1, rs2	 54
	4.7.4 MULHU rd, rs1, rs2	 54
	4.7.5 DIV rd, rs1, rs2	 54
	4.7.6 DIVU rd, rs1, rs2	 54
	4.7.7 REM rd, rs1, rs2	 55
	4.7.8 REMU rd, rs1, rs2	 55
4.8	RV32A Standard Extension	 55
4.9	RV32F Standard Extension	 55
4.10	0 RV32D Standard Extension	 55
A Att	tribution 4.0 International	56
Bibliog	ography	60
Index	C C	61
Glossa	sary	63

Preface

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

The closest thing to what I sought when deciding to collect my thoughts into this document would be select portions of *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version* 2.2[1], The RISC-V Reader[2], and Computer Organization and Design RISC-V Edition: The Hardware Software Interface[3].

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

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

I learned assembler from data sheets and texts (that are still sitting on my shelves) such as:

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

One way or another all of them discuss each CPU instruction in excruciating detail with both a logical and narrative description. For RISC-V this is also the case for the RISC-V Reader[2] and the Computer Organization and Design RISC-V Edition[3] books and is also present in this text (I consider that to be the minimal level of responsibility.)

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

Chapter 1

Introduction

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 mneumonic 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 mneumonics, parameters and rules for specifying their use for the purpose of programming a CPU is called an Assembly Language.

1.1 The Digital Computer

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*.

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

This text focuses on digital computing.

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.)

1.1.1 Storage Systems

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

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

1.1.1.1 Volatile Storage

Volatile storage is characterized by the fact that it will lose its contents (forget) any time that it is powered off.

One type of volatile storage is provided inside the CPU itself in small blocks called registers. These CPU registers are used to hold individual data values that can be manipulated by the instructions that are ISA executed by the CPU.

Another type of volatile storage is main memory. Main memory is connected to a computer's CPU and is used to hold the data and instructions that can not fit into the CPU registers.

Typically, a CPU's registers can hold tens of data values while the main memory can contain many billions of data values.

A CPU can process data in a register at a speed that can be an order of magnitude faster than the rate that it can process (specifically, transfer data and instructions to and from) the main memory.

Register storage costs an order of magnitude more to manufacture than main memory. While it is desirable to have many registers the economics dictate that the vast majority of volatile computer storage be provided in its main memory. As a result, optimizing the copying of data between the registers and main memory is a desirable trait of good programs.

1.1.1.2 Non-Volatile Storage

Non-volatile storage is characterized by the fact that it will NOT lose its contents when it is powered off.

Common types of non-volatile storage are disc drives, flash cards and USB drives. Prices can vary widely depending on size and transfer speeds.

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

This text is not particularly concerned with non-volatile storage.

1.1.2 CPU

The CPU is a collection of registers and circuitry designed to read data and instructions from the storage system. The instructions tell the CPU to perform various mathematical and logical operations on the data in its registers and where to save the results of those operations.

1.1.3Peripherals

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

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

Instruction Set Architecture 1.2

The catalog of rules that describes the details of the instructions and features that a given CPU provides is called its Instruction Set Architecture (ISA).

An ISA is typically expressed in terms of the specific meaning of each binary instruction that a CPU RV32I can recognize and how it will process each one.

RV32M RV32A

The RISC-V ISA is defined as a set of modules. The purpose of dividing the ISA into modules is to allow an implementor to select which features to incorporate into a CPU design.

RV32F RV32D RV32Q

RV32C

Any given RISC-V implementation must provide one of the base modules and zero or more of the extension modules.

RV32G instruction cycle

1.2.1 **RV** Base Modules

The base modules are RV32I (32-bit general purpose), RV32E (32-bit embedded), RV64I (64-bit general purpose) and RV128I (128-bit general purpose).

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

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

1.2.2Extension Modules

RISC-V extension modules may be included by an implementor interested in optimizing a design for one or more purposes.

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

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

1.3 An Example Computer

The machine used to execute the programs presented in this text has one RV32I CPU with 32 registers, >> Fix Me: one hart (analogous to what is called a *core* on other CPUs such as an ARM) and 65536 bytes of memory.

Need a block diagram and description of the virtual machine that is used in this

1.4 Executing a Program

To observe the operation of our example computer an RV32I simulator will be used that will print a message describing the status of the CPU and the instructions that it executes as it goes along.

The process of executing an instruction is called an instruction cycle and it is comprised of an instruction fetch and an instruction execute phase.

The status of the CPU is entirely embodied in the data values that are stored in its registers at any moment in time. The simulator can print all of the register values before it executes an instruction for reference.

When an instruction is executed the simulator can print a message describing where in main memory it

came from, its numeric machine code value, its mneumonic, a description of any associated parameters, the values of those parameters and then carry out the operation as defined by the ISA.

instruction fetch instruction execute

For this to work, the instructions to be executed will have been previously stored in a list in the main memory and any parameters that an instruction specifies will either be part of the instruction itself or read from (or stored into) one or more of the registers.

1.4.1 Instruction Fetch

In order to *fetch* an instruction from the main memory the CPU must have a method to identify which instruction should be fetched and a method to fetch it.

To make this possible the main memory is broken up into small blocks called bytes that are each given a unique identifying number called an address. The process of identifying which instruction to fetch is therefore a matter of knowing what address it is stored in.

A byte is comprised of eight binary digits called bits.

Every possible instruction that the RV32I can execute contains exactly 32 bits. Therefore each instruction must be stored in four bytes of the main memory.

To simplify the hardware, each instruction must be placed into four adjacent bytes whose numeric address sequence begins with a multiple four. For example, an instruction might be located in bytes 12, 13, 14 and 15 (but not in 15, 16, 17 and 18 nor 8, 207, 5, and 1073...).

This sort of addressing requirement is common and is referred to as alignment. An aligned instruction begins at a memory address that is a multiple of four. An *unaligned* instruction would be one beginning at any other address and is *illegal*.

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

Given a properly aligned instruction address, the CPU can request that the main memory locate and deliver the values of the four bytes in the address sequence to the CPU using what is called a memory read operation. Some systems can deliver four (or more) bytes at the same time while others might only be capable of delivering one or two bytes at a time. These differences in hardware typically impact the cost and performance of a system.¹

1.4.2 Instruction Execute

Once an instruction has been fetched by the CPU, it can be executed.

Typical instructions do things like add a number to the value currently stored in one of the registers or store the contents of a register into the main memory at some given address.

Also part of every instruction is a notion of what should be done next.

¹The 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.

Most of the time an instruction will be complete by indicating that the CPU should proceed to fetch and execute the instruction at the next larger main memory address.

Some instructions can specify that the CPU proceed to execute an instruction at an address other than the one that follows itself. This class of instructions have names like *jump* and *branch* and are available in a variety of different styles.

The RV ISA uses the word *jump* to refer to an *unconditional* change in the sequential processing of instructions and the word *branch* to refer to a *conditional* change.

For example, a (conditional) branch instruction might instruct the CPU to proceed to the instruction at the next main memory address if the value in register number 8 is currently less than the value in register number 24 but otherwise proceed to an instruction at a different address when it is not. This type of instruction can therefore result in having one of two different actions pending the resulting condition of the comparison.²

1.4.3 A Sample Program Source Listing

A simple program that illustrates how this text presents program source code is seen in Listing 1.1. This program will place a zero in each of the 4 registers named x28, x29, x30 and x31.

Listing 1.1: zero4regs.S Setting four registers to zero.

```
# put this into the text section
       .text
      .align
                                # align to 2^2
      .globl
3
              _start
4
  start:
               x28, x0, 0
5
      addi
                                # set register x28 to zero
               x29, x0, 0
      addi
                                # set register x29 to zero
6
      addi
               x30, x0, 0
                                # set register x30 to zero
               x31, x0, 0
      addi
                                # set register x31 to zero
```

This program listing illustrates a number of things:

- Listings are identified by the name of the file within which they are stored. This listing is from a file named: zero4regs.S.
- The assembly language programs discussed in this text will be saved in files that end with: .S (Alternately you can use .sx on systems that don't understand the difference between upper and lowercase letters.³)
- A description of the listing's purpose appears under the name of the file. The description of Listing 1.1 is Setting four registers to zero.
- The lines of the listing are numberd on the left margin for easy reference.
- An assembly program consists of lines of plain text.
- The RISC-V ISA does not provide an operation that will simply set a register to a numeric value.
 To accomplish our goal this program will add zero to zero and place the sum in in each of the four registers.

²This is the fundamental method used by a CPU to make decisions.

³The author of this text prefers to avoid using such systems.

- The lines that start with a dot '.' (on lines 1, 2 and 3) are called assembler directives as they tell rvddt the assembler itself how we want it to translate the following assembly language instructions into machine language instructions.
- Line 4 shows a *label* named _start. The colon at the end is the indicator to the assembler that causes it to recognize the preceeding characters as a label.
- Lines 5-8 are the four assembly language instructions that make up the program. Each instruction in this program consists of four fields. (Different instructions can have a different number of fields.) The fields on line 5 are:
 - addi The instruction mneumonic. It indicates the operation that the CPU will perform.
 - x28 The destination register that will receive the sum when the addi instruction is finished. The names of the 32 registers are expressed as x0 - x31.
 - x0 One of the addends of the sum operation. (The x0 register will always contain the value zero. It can never be changed.)
 - 0 The second addend is the number zero.
- # set ... Any text anywhere in a RISC-V assembly language program that starts with the pound-sign is ignored by the assembler. They are used to place a *comment* in the program to help the reader better understand the motive of the programmer.

Running a Program With rvddt

To illustrate what a CPU does when it executes instructions this text will use a simulator that shows sequence of events and the binary values involved. Listing 1.2 shows the operation of the four addi instructions from Listing 1.1 when executed using the rvddt simulator.⁴

Listing 1.2: zero4regs.out Running a program with the rvddt simulator

```
[winans@w510 src]$ ./rvddt -f ../t1/load4regs.bin
 Loading '../t1/load4regs.bin' to 0x0
 ddt> t4
  pc: 00000000
                \# x28 = 0x00000000 = 0x00000000 + 0x00000000
 00000000: 00000e13 addi
           x28, x0, 0
  10
  x8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
11
 12
 13
  pc: 00000004
 00000004: 00000e93
           x29, x0, 0
                # x29 = 0x00000000 = 0x00000000 + 0x00000000
        addi
15
16
  x8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
17
 18
 19
  pc: 00000008
20
                \# x30 = 0x00000000 = 0x00000000 + 0x00000000
 00000008: 00000f13 addi
           x30, x0, 0
  22
  23
 24
 25
  pc: 0000000c
```

⁴The rvddt application was written to generate the listings for this text. It is similar to the fancier spike simulator. Given the simplicity of the RV32I ISA, ryddt is less than 1700 lines of C++ and was written in one (long) afternoon.

```
0000000c: 00000f93
                       \# x31 = 0x00000000 = 0x00000000 + 0x00000000
           addi
                x31. x0. 0
28
 ddt> r
   x0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
29
   30
  31
  32
   pc: 0000010
33
 ddt> x
34
 [winans@w510 src]$
35
```

- 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 extructions.
- ℓ 4-8 Prior to executing the first instruction, the state of the CPU registers is displayed.
 - \$\ell\$ 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 dash '-' character 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.
 - \$\ell\$ 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:
- 00000000 The memory address from which the instruction was fetched. This address is displayed in big endian, hexadecimal form.
- 00000e13 The machine code of the instruction displayed in big endian, hexadecimal form.
 - addi The mneumonic for the machine instruction.
 - x28 The rd field of the addi instruction.
 - x0 The rs1 field of the addi instruction that holds one of the two addends of the operation.
 - 0 The imm field of the addi instruction that holds the second of the two addends of the operation.
 - $\#\ldots$ A simulator-generated comment that exaplains what the instruction is doing. For this instruction it indicates that x28 will have the value zero stored into it as a result of performing the addition: 0+0.
- \$\ell\$ 10-14 These lines are printed as the prelude while tracing the second instruction. Lines 7 and 13 show that x28 has changed from f0f0f0f0 to 00000000 as a result of executing the first instruction and lines 8 and 14 show that the pc has advanced from zero (the location of the first instruction) to four, where the second instruction will be fetched. None of the rest of the registers have changed values.
 - ℓ 15 The second instruction decoded executed and described. This time register x29 will be assigned a value.
- ℓ 16-27 The third and fourth instructions are traced.

- ℓ 28 Tracing has completed. The simulator prints its prompt and the user enters the 'r' command to see the register state after the fourth instruction has completed executing.
- ℓ 29-33 Following the fourth instruction it can be observed that registers x28, x29, x30 and x31 have been set to zero and that the pc has advanced from zero to four, then eight, then 12 (the hex value for 12 is c) and then to 16 (which, in hex, is 10).
 - ℓ 34 The simulator exit command 'x' is entered by the user and the terminal displays the shell prompt.

Chapter 2

Number Systems

RISC-V systems represent information using binary values stored in little-endian order. ¹

2.1 Integers

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

Counting in binary is the same as in decimal. For example, when adding 1 to 9, the carry is added to the next place value. When subtracting 1 from 0, a borrow is required and so on.

Figure Figure 2.1 shows an abridged table of the decimal, binary and hexadecimal values from 0 to 129.

One way to look at this table is on a per-row basis where each place value is represented by the base raised to the power of the place value position (shown in the column headings.) This is useful when converting arbitrary values between bases. For example to interpret the decimal value on the fourth row:

$$0 \times 10^2 + 0 \times 10^1 + 3 \times 10^0 = 3_{10}$$

And to interpret binary value on the same row by converting it to decimal:

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

And the same for the hexadecimal value:

$$0 \times 16^1 + 3 \times 16^0 = 3_{10}$$

¹See[13] for some history of the big/little-endian "controversy."

Decimal			Binary					Н	ex			
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.

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

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

The relationship between binary and hex values is also worth taking note. Because $2^4 = 16$, there is a clean and simple grouping of 4 bits to 1 hit. There is no such relationship between binary and decimal.

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

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

Binary value: 0010 0111 1011 1010 1100 1100 1111 0101

Hex Value: 2 7 B A C C F 5

Decimal Value: 666553589

Empirically we can see that grouping the bits into sets of four allows an easy conversion to hex and expressing it as such is $\frac{1}{4}$ as long as in binary while at the same time allowing for easy conversion back to binary.

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

2.1.1 Converting Between Bases

2.1.1.1 From Binary to Decimal

Alas, it is occasionally necessary to convert between decimal, binary and/or hex.

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

```
128 64 32 16 8 4 2 1
0 0 0 1 1 0 1 1
```

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

2.1.1.2 From Binary to Hexadecimal

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

Grouping the bits into sets of four and summing:

Place:	8 4 2 1	8 4 2 1 8 4 2 1	8 4 2 1
Binary:	0 1 1 0	1 1 0 1 1 0 1 0	1 1 1 0
Decimal:	4+2 =6	8+4+ 1=13 8+ 2 =10	8+4+2 =14

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

Continuing this example:

Decimal:	6	13	10	14
Hex:	6	D	Α	E

2.1.1.3 From Hexadecimal to Binary

Again, the four-bit mapping between binary and hex makes this task as straight forward as using a look-up table.

For each hit (Hex digIT), translate it to its unique four-bit pattern. Perform this task either by memorizing each of the 16 patterns or by converting each hit to decimal first and then converting each four-bit binary value to decimal using the place-value summing method discussed in subsubsection 2.1.1.1.

For example:

```
Hex: 4 C
Binary: 0 1 0 0 1 1 0 0
Decimal: 128 64 32 16 8 4 2 1
Sum: 64+ 8+4 = 76
```

2.1.1.4 From Decimal to Binary

To convert arbitrary decimal numbers to binary, extend the list of binary place values until it exceeds the value of the decimal number being converted. Then make successive subtractions of each of the place values that would yield a non-negative result.

For example, to convert 1234_{10} to binary:

Place values: 2048-1024-512-256-128-64-32-16-8-4-2-1

```
0
            2048
                        (too big)
    1234 - 1024 = 210
1
0
            512
                        (too big)
0
            256
                        (too big)
1
     210 - 128
                  = 82
1
       82 - 64
                  = 18
0
            32
                        (too big)
1
       18 - 16
                  = 2
0
            8
                        (too big)
0
            4
                        (too big)
1
            2
0
            1
                        (too big)
```

The answer using this notation is listed vertically in the left column with the $\overline{\text{MSB}}$ on the top and the $\overline{\text{LSB}}$ on the bottom line: 010011010010_2 .

2.1.1.5 From Decimal to Hex

Conversion from decimal to hex can be done by using the place values for base-16 and the same math as from decimal to binary or by first converting the decimal value to binary and then from binary to hex by using the methods discussed above.

Because binary and hex are so closely related, performing a conversion by way of binary is quite straight forward.

2.1.2 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.

For example:

```
111111 1111 <== carries
01101011111001111 <== addend
+ 0000011101100011 <== addend
------
0111001100110010 <== sum
```

2.1.3 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:

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

```
-128 64 32 16 8 4 2 1
1 1 1 1 1 1 1 1 1
```

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

This format has the virtue of allowing the same addition logic discussed above to be used to calculate -1+1=0.

In order for this to work, the overflow carry out of the sum of the MSBs is ignored.

2.1.3.1 Converting between Positive and Negative

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

For example, inverting the number four:

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

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

```
-128 64 32 16
        1
  1
     1
           1
              1
                         <== carries
  1
     1
        1
           1
              1
                 1
                    0
                       0 <== -4
              0
        0
           0
                 1
                    0
                         <== 5
  0 0 0 0 0 0 0 1
```

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

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

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

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

2.1.4 Subtraction of Binary Numbers

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

To calculate -4 - 8 = -12

▶ Fix Me:

This section needs more examples of subtracting signed an 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.

1 1 1 1 0 0 0 <== -8

2.1.5 Truncation and Overflow

Disscuss the details of truncation and overflow here.

2.1.6 Logical/Boolean Functions

Unlike addition and subtraction, boolean functions apply on a per-bit basis. When applied to multi-bit values, each bit position is operated upon independently of the other bits.

2.1.6.1 NOT

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

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.

This text will use the operator used in the C language when discussing the NOT operator in symbolic form. Specifically the tilde: ' \sim '.

```
~ 1 1 1 1 0 1 0 1 <== A
------
0 0 0 0 1 0 1 0 <== output
```

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

2.1.6.2 AND

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

This text will use the operator used in the C language when discussing the AND operator in symbolic form. Specifically the ampersand: '&'.

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

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

▶ Fix Me:

This chapter should be made consistent in its use of truncation and overflow as occur with signed and unsigned addition and subtraction.

➡ Fix Me:

This is unclear. Need to define bit positions and probably should add basic truth table diagrams.

Fix Me:

Need to define 1 as true and 0 as false somewhere.

→ Fix Me:

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

2.1.6.3 OR

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

This text will use the operator used in the C language when discussing the OR operator in symbolic form. Specifically the pipe: '|'.

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

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

2.1.6.4 XOR

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

This text will use the operator used in the C language when discussing the XOR operator in symbolic form. Specifically the carrot: '^'.

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

For example:

```
1 1 1 1 0 1 0 1 <== A

^ 1 0 0 1 0 0 1 1 <== B

------

0 1 1 0 0 1 1 0 <== output
```

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

2.2 IEEE-754 Floating Point Number Representation

This section provides an overview of the IEEE-754 32-bit binary floating point format.

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

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

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

```
 \dots \ 128 \ 64 \ 32 \ 16 \ 8 \ 4 \ 2 \ 1 \ . \ 1/2 \ 1/4 \ 1/8 \ 1/16 \ 1/32 \ 1/64 \ 1/128 \ \dots
```

The '.' in a binary number is a binary point, not a decimal point.

- 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.
- The format of a number in scientific notation is $mantissa \times base^{exponent}$
- In binary we have $mantissa \times 2^{exponent}$
- 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.
 - The unnormalized binary value of -2.625 is 10.101
 - The normalized value of -2.625 is 1.0101×2^{1}
- We need not store the '1.' because *all* normalized floating point numbers will start that way. Thus we can save memory when storing normalized values by adding 1 to the significand.

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

• IEEE754 formats:

	IEEE754 32-bit	IEEE754 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

- When the exponent is all ones, the mantissa is all zeros, and the sign is zero, the number represents positive infinity.
- When the exponent is all ones, the mantissa is all zeros, and the sign is one, the number represents negative infinity.
- Note that the binary representation of an IEEE754 number in memory can be compared for magnitude with another one using the same logic as for comparing two's complement signed integers because the magnitude of an IEEE number grows upward and downward in the same fashion as signed integers. This is why we use excess notation and locate the significand's sign bit on the left of the exponent.
- Note that zero is a special case number. Recall that a normalized number has an implied 1-bit to the left of the significand... which means that there is no way to represent zero! Zero is represented by an exponent of all-zeros and a significand of all-zeros. This definition allows for a positive and a negative zero if we observe that the sign can be either 1 or 0.
- On the number-line, numbers between zero and the smallest fraction in either direction are in the underflow areas.
- On the number line, numbers greater than the mantissa of all-ones and the largest exponent allowed are in the *overflow* areas.
- Note that numbers have a higher resolution on the number line when the exponent is smaller.

→ Fix Me:

Need to add the standard lecture numberline diagram showing where the over/under-flow areas are and why.

2.2.1 Floating Point Number Accuracy

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

2.2.1.1 Powers Of Two

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.)

Listing 2.1: powersoftwo.c

Precise Powers of Two

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

Listing 2.2: powersoftwo.out Output from powersoftwo.c

```
1.00000000000 = 3f800000
                                             -1.00000000000 = 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
9
   0.0019531250 = 3b000000
                                             -0.0019531250 = bb000000
10
```

2.2.1.2 Clean Decimal Numbers

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

Note how the decimal numbers are not accurately represented as they get larger. The decimal number on line 10 of Listing 2.4 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 2.3: cleandecimal.c Print Clean Decimal Numbers

```
#include <stdio.h>
   #include <stdlib.h>
   #include <unistd.h>
3
   union floatbin
5
6
        unsigned int
                          i;
        float
                          f;
   };
   int main()
10
11
        union floatbin
12
                          х, у;
13
        int
                          i;
14
        x.f = 10;
15
16
        while (x.f \le 10000000000000.0)
17
            y.f = -x.f;
18
            printf("%25.10f = %08x)
19
                                            %25.10f = %08x\n", x.f, x.i, y.f, y.i);
            x.f = x.f*10.0;
20
        }
21
   }
22
```

Listing 2.4: cleandecimal.out Output from cleandecimal.c

```
10.00000000000 = 41200000
                                              -10.00000000000 = c1200000
           100.0000000000 = 42c80000
                                             -100.00000000000 = c2c80000
           1000.00000000000 = 447a0000
                                            -1000.00000000000 = c47a0000
3
4
          10000.00000000000 = 461c4000
                                           -10000.000000000000 = c61c4000
         100000.00000000000 = 47c35000
                                           5
        1000000.00000000000 = 49742400
                                          -1000000.00000000000 = c9742400
7
       10000000.00000000000 = 4b189680
                                         8
     -1000000000.0000000000 = ce6e6b28
9
     100000000000.000000000000000 = 501502f9
                                       -1000000000000.00000000000000 = d01502f9
10
     99999997952.0000000000 = 51ba43b7
                                      -99999997952.000000000000000 = d1ba43b7
    -99999999999994.00000000000 = d368d4a5
12
   9999999827968.0000000000 = 551184e7
                                     -9999999827968.000000000 = d51184e7
13
```

2.2.1.3 Accumulation of Error

These rounding errors can be exaggerated when the number we multiply the $\mathtt{x.f}$ value by is, itself, something that can not be accurately represented in IEEE form.²

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

Listing 2.5: erroraccumulation.c Accumulation of Error

▶ 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.

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.

```
#include <stdio.h>
   #include <stdlib.h>
   #include <unistd.h>
   union floatbin
        unsigned int
                          i:
7
        float
                          f;
8
   }:
9
   int main()
10
11
   {
        union floatbin
12
                         х, у;
13
        int
                          i;
14
        x.f = .1;
15
16
        while (x.f \le 2.0)
17
            y.f = -x.f;
18
            printf("%25.10f = %08x
                                           25.10f = 00x^n, x.f, x.i, y.f, y.i);
19
            x.f += .1;
        }
21
   }
22
```

Listing 2.6: erroraccumulation.out Output from erroraccumulation.c

```
0.1000000015 = 3dccccd
                                            -0.1000000015 = bdccccd
   0.2000000030 = 3e4cccd
                                            -0.2000000030 = be4cccd
   0.300000119 = 3e99999a
                                             -0.3000000119 = be99999a
   0.4000000060 = 3eccccd
                                             -0.4000000060 = beccccd
   0.50000000000 = 3f000000
                                             -0.50000000000 = bf000000
   0.6000000238 = 3f19999a
                                            -0.6000000238 = bf19999a
   0.7000000477 = 3f333334
                                            -0.7000000477 = bf333334
   0.8000000715 = 3f4cccce
                                             -0.8000000715 = bf4cccce
   0.9000000954 = 3f666668
                                            -0.9000000954 = bf666668
   1.0000001192 = 3f800001
                                            -1.0000001192 = bf800001
10
                                            -1.1000001431 = bf8cccce
   1.1000001431 = 3f8cccce
11
   1.2000001669 = 3f99999b
                                            -1.2000001669 = bf99999b
   1.3000001907 = 3fa66668
                                             -1.3000001907 = bfa66668
13
   1.4000002146 = 3fb33335
                                             -1.4000002146 = bfb33335
   1.5000002384 = 3fc00002
                                             -1.5000002384 = bfc00002
15
   1.6000002623 = 3fcccccf
                                            -1.6000002623 = bfccccf
   1.7000002861 = 3fd9999c
17
                                            -1.7000002861 = bfd9999c
   1.8000003099 = 3fe66669
                                             -1.8000003099 = bfe66669
18
   1.9000003338 = 3ff33336
                                             -1.9000003338 = bff33336
```

2.2.2 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 2.7: errorcompensation.c Accumulation of Error

```
#include <stdio.h>
#include <stdib.h>
#include <unistd.h>

union floatbin
{
    unsigned int i;
```

```
float
                          f;
   };
9
   int main()
10
   {
11
        union floatbin x, y;
12
13
                          i;
14
        i = 1;
15
        while (i <= 20)
16
17
            x.f = i/10.0;
18
            y.f = -x.f;
19
            printf("%25.10f = %08x
                                          25.10f = %08x\n", x.f, x.i, y.f, y.i);
20
21
            i++;
22
        return(0);
23
24
   }
```

Listing 2.8: errorcompensation.out Output from erroraccumulation.c

```
0.1000000015 = 3dccccd
                                            -0.1000000015 = bdccccd
   0.2000000030 = 3e4cccd
                                             -0.2000000030 = be4cccd
   0.300000119 = 3e99999a
                                            -0.3000000119 = be99999a
   0.4000000060 = 3eccccd
                                            -0.4000000060 = beccccd
   0.5000000000 = 3f000000
                                            -0.50000000000 = bf0000000
   0.6000000238 = 3f19999a
                                            -0.6000000238 = bf19999a
   0.6999999881 = 3f333333
                                            -0.6999999881 = bf333333
                                            -0.8000000119 = bf4cccd
   0.800000119 = 3f4cccd
   0.8999999762 = 3f666666
                                            -0.8999999762 = bf666666
   1.00000000000 = 3f800000
                                            -1.00000000000 = bf800000
   1.1000000238 = 3f8cccd
                                            -1.1000000238 = bf8cccd
11
   1.2000000477 = 3f99999a
                                            -1.2000000477 = bf99999a
12
   1.2999999523 = 3fa66666
                                            -1.2999999523 = bfa66666
13
   1.3999999762 = 3fb33333
                                             -1.3999999762 = bfb33333
14
   1.50000000000 = 3fc00000
                                            -1.50000000000 = bfc00000
15
   1.6000000238 = 3fccccd
                                            -1.6000000238 = bfccccd
16
   1.7000000477 = 3fd9999a
                                            -1.7000000477 = bfd9999a
   1.7999999523 = 3fe66666
                                            -1.7999999523 = bfe66666
18
   1.8999999762 = 3ff33333
                                             -1.8999999762 = bff33333
19
   2.00000000000 = 40000000
                                            -2.00000000000 = c00000000
20
```

Chapter 3

The RISC-V GNU Toolchain

This chapter discusses the GNU toolchain elements used to experiment with the material in this book.

The instructions and examples here were all implemented on Ubuntu 16.04 LTS.

Fix Me:

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

Install custom code in a location that will not cause interference with other applications and allow for easy cleanup. These instructions install the toolchain in /usr/local/riscv. At any time you can proper variations on this remove the lot and start over by executing the following command:

```
rm -rf /usr/local/riscv/*
```

Tested on Ubuntu 16.04 LTS. 18.04 was just released... update accordingly.

These are the only commands that you should perform as root when installing the toolchain:

```
sudo apt-get install autoconf automake autotools-dev curl libmpc-dev \
libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf \
libtool patchutils bc zlib1g-dev libexpat-dev
sudo mkdir -p /usr/local/riscv/
sudo chmod 777 /usr/local/riscv/
```

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.

To download, compile and "install" the toolchain:

```
# riscv toolchain:
# https://riscv.org/software-tools/risc-v-gnu-compiler-toolchain/
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain
./configure --prefix=/usr/local/riscv/rv32i --with-arch=rv32i --with-abi=ilp32
make
make install
```

Need to discuss augmenting the PATH environment variable.

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

Discuss rv32im and note that the details are found in chapter 4.

Disciuss installing and using one of the RISC-V simulators here.

Describe the pre-processor, compiler, assemler and linker.

Source, object, and binary files

Assembly syntax (label: mnemonic op1, op2, op3 # comment).

text, data, bss, stack

Labels and scope.

Forward & backward references to throw-away labels.

The entry address of an application.

.s file contain assembler code. .S (or .sx) files contain assembler code that must be preprocessed. [14, p. 29]

Pre-processing conditional assembly using #if.

Building with -mabi=ilp32 -march=rv32i -mno-fdiv -mno-div to match the config options on the toolchain.

Linker scripts.

Makefiles

objdump

nm

hexdump -C

Chapter 4

RV32 Machine Instructions

4.1 Introduction

Fix Me:

Discuss what the IMAFD, G and other ISA extensions mean as well as the 32, 64 and 128-bit versions.

4.2 Conventions and Terminology

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

4.2.1 XLEN

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

$4.2.2 \quad \text{sx(val)}$

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 binary numbers see chapter 2.

Figure 4.1 illustrates extending the negative sign bit of *val* to the left by replicating it. When *val* is negative, its MSB (bit 19 in this example) will be set to 1. Extending this value to the left will set all the new bits to the left of it to 1 as well.

Figure 4.2 illustrates extending the positive sign bit of *val* to the left by replicating it. When *val* is positive, its MSB will be set to 0. Extending this value to the left will set all the new bits to the left of it to 0 as well.

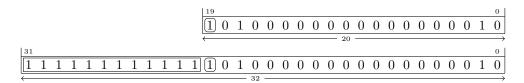


Figure 4.1: Sign-extending a negative integer from 20 bits to 32 bits.

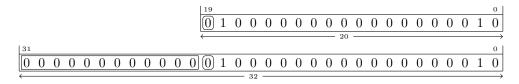


Figure 4.2: Sign-extending a positive integer from 20 bits to 32 bits.

4.2.3 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. Figure 4.3 illustrates zero-extending a 20-bit val to the left to form a 32-bit fullword.

For more on binary numbers see chapter 2.

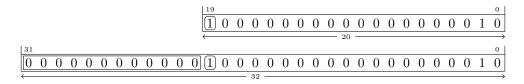


Figure 4.3: Zero-extending an unsigned integer from 20 bits to 32 bits.

4.2.4 zr(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 4.4 illustrates converting a 20-bit val to a 32-bit fullword.

4.2.5 Sign Entended Left and Zero Extend Right

Some instructions such as the J-type (see subsection 4.4.2) include immediate operands that are extended in both directions.

Figure 4.5 and Figure 4.6 illustrates zero-extending a 20-bit negative number one bit to the right and

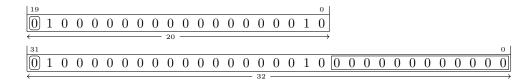


Figure 4.4: Zero-extending an integer to the right from 20 bits to 32 bits.

sign-extending it 11 bits to the left:

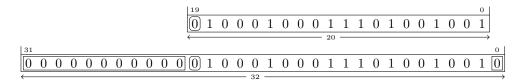


Figure 4.5: Sign-extending a positive 20-bit number 11 bits to the left and one bit to the right.

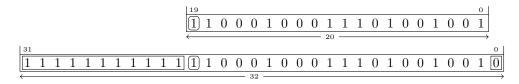


Figure 4.6: Sign-extending a negative 20-bit number 11 bits to the left and one bit to the right.

4.2.6 m8(addr)

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

Given the contents of the memory dump shown in Figure 4.7, m8(42) refers to the memory location at address 42_{16} that currently contains the 8-bit value fc_{16} .

The mn(addr) notation can be used to refer to memory that is being read or written depending on the context.

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

$$m8(addr) \leftarrow source$$

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

$$dest \leftarrow m8(addr)$$

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

$4.2.7 \quad m16(addr)$

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

```
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 4.7: Sample memory contents.

Given the contents of the memory dump shown in Figure 4.7, m16(42) refers to the memory location at address 42_{16} that currently contains $65fc_{16}$. See also subsection 4.2.6.

$4.2.8 \quad m32(addr)$

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

Given the contents of the memory dump shown in Figure 4.7, m32(42) refers to the memory location at address 42_{16} that currently contains $662065fc_{16}$. See also subsection 4.2.6.

$4.2.9 \quad m64(addr)$

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

Given the contents of the memory dump shown in Figure 4.7, m64(42) refers to the memory location at address 42_{16} that currently contains $656c6c69662065fc_{16}$. See also subsection 4.2.6.

$4.2.10 \quad m128 (addr)$

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

Given the contents of the memory dump shown in Figure 4.7, m128(42) refers to the memory location at address 42_{16} that currently contains $7220687469772064656c6c69662065fc_{16}$. See also subsection 4.2.6.

4.2.11 .+offset

The address of the current instruction plus a numeric offset.

4.2.12 .-offset

The address of the current instruction minus a numeric offset.

4.2.13 pc

The current value of the program counter.

4.2.14 rd

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

4.2.15 rs1

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

4.2.16 rs2

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

4.2.17 imm

An immediate numeric operand. The word *immediate* refers to the fact that the operand is stored within an instruction.

$4.2.18 \quad rsN[h:l]$

The value of bits from h through l of x-register rsN. For example: rs1[15:0] refers to the contents of the 16 LSBs of rs1.

4.3 Addressing Modes

 $immediate,\ register,\ base-displacement,\ pc\text{-relative}$

→ Fix Me:

Write this section.

4.4 Instruction Encoding Formats

This document concerns itself with the following RISC-V instruction formats.

XXX Show and discuss a stack of formats explaining how the unnatural ordering of the *imm* fields reduces the number of possible locations that the hardware has to be prepared to *look* for various bits. For example, the opcode, rd, rs1, rs1, func3 and the sign bit (when used) are all always in the same position. Also note that imm[19:12] and imm[10:5] can only be found in one place. imm[4:0] can only be found in one of two places...

The point to all this is that it is easier to build a machine if it does not have to accommodate many different ways to perform the same task. This simplification can also allow it operate faster.

Figure 4.8 Shows the RISC-V instruction formats.

→ Fix Me:

Should discuss types and sizes beyond the fundamentals. Will add if/when instruction details are added in the future.

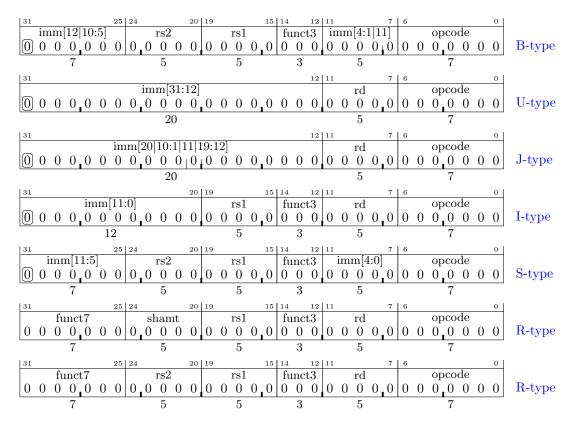
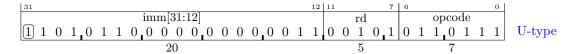


Figure 4.8: RISC-V instruction formats.

4.4.1 U Type

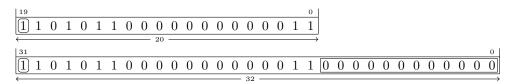
The U-Type format is used for instructions that use a 20-bit immediate operand and a destination register.



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

The imm field contains a 20-bit value that will be converted into XLEN bits by using the *imm* operand for bits 31:12 and then sign-extending it to the left¹ and zero-extending the LSBs as discussed in subsection 4.2.4.

If XLEN=32 then the imm value in this example will be converted as shown below.



Notice that the 20-bits of the imm field are mapped in the same order and in the same relative position

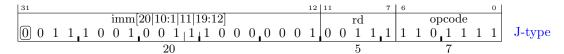
¹When XLEN is larger than 32.

that they appear in the instruction when they are used to create the value of the immediate operand. Shifting the imm value to the left, into the "upper bits" of the immediate value suggests a rationale for the name of this format.

If XLEN=64 then the imm value in this example will be converted to the same two's complement integer value by extending the sign to the left.

4.4.2 J Type

The J-type format is used for instructions that use a 20-bit immediate operand and a destination register. It is similar to the U-type. However, the immediate operand is constructed by arranging the *imm* bits in a different manner.



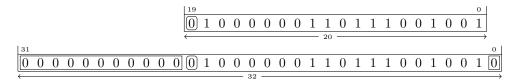
The rd field contains an x register number to be set to a value that depends on the instruction.

In the J-type format the 20 imm bits are arranged such that they represent the "lower" portion of the immediate value. Unlike the U-type instructions, the J-type requires the bits to be re-ordered and shifted to the right before they are used.²

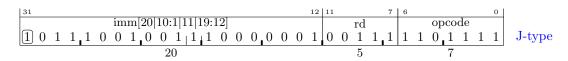
The example above shows that the bit positions in the imm field description. We see that the 20 imm bits are re-ordered according to: [20|10:1|11|19:12]. This means that the MSB of the imm field is to be placed into bit 20 of the immediate integer value ultimately used by the instruction when it is converted into XLEN bits. The next bit to the right in the imm field is to be placed into bit 10 of the immediate value and so on.

After the *imm* bits are re-positioned into bits 20:1 of the immediate value being constructed, a zero-bit will be added to the LSB and the value in bit-position 20 will be replicated to sign-extend the value to XLEN bits as discussed in subsection 4.2.5.

If XLEN=32 then the *imm* value in this example will be converted as shown below.

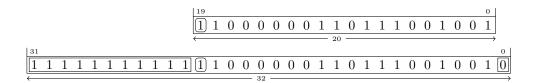


A J-type example with a negative imm field:



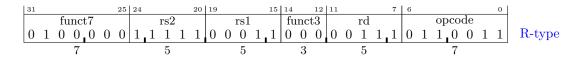
If XLEN=32 then the *imm* field in this example will be converted as shown below.

 $^{^2}$ The reason that the J-type bits are reordered like this is because it simplifies the implementation of hardware as discussed in section 4.4.



The J-type format is used by the Jump And Link instruction that calculates a target address by adding a signed immediate value to the current program counter. Since no instruction can be placed at an odd address the 20-bit imm value is zero-extended to the right to represent a 21-bit signed offset capable of representing numbers twice the magnitude of the 20-bit imm value.

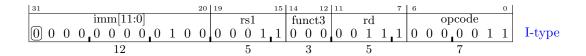
4.4.3 R Type



A special case of the R-type used for shift-immediate instructions where the rs2 field is used as an immediate value named shamt representing the number of bit positions to shift:



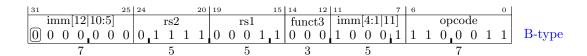
4.4.4 I Type



4.4.5 S Type



4.4.6 B Type



4.4.7 **CPU Registers**

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

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

discusses the calling conventions

The lack of grouping the temporary and saved registers is due to the fact that the C extension provides access to only the first 16 registers when executing instructions in the compressed format.

Reg	Alias	Description	Saved
x0	zero	Hard-wired zero	
x1	ra	Return address	
x2	$_{\rm sp}$	Stack pointer	yes
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary/alternate link register	
x6	t1	Temporary	
x7	t2	Temporary	
x8	s0/fp	Saved register/frame pointer	yes
x9	s1	Saved register	yes
x10	a0	Function argument/return value	
x11	a1	Function argument/return value	
x12	a2	Function argument	
x13	a3	Function argument	
x14	a4	Function argument	
x15	a5	Function argument	
x16	a6	Function argument	
x17	a7	Function argument	
x18	s2	Saved register	yes
x19	s3	Saved register	yes
x20	s4	Saved register	yes
x21	s5	Saved register	yes
x22	s6	Saved register	yes
x23	s7	Saved register	yes
x24	s8	Saved register	yes
x25	s9	Saved register	yes
x26	s10	Saved register	yes
x27	s11	Saved register	yes
x28	t3	Temporary	
x29	t4	Temporary	
x30	t5	Temporary	
x31	t6	Temporary	

4.5 memory

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

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

RV32I Instruction!LUI

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

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 mempry access from operating atomically. [1, p.19] See also ??.

4.6 RV32I Base Instruction Set

RV32I refers to the basic 32-bit integer instructions.

4.6.1 LUI rd, imm

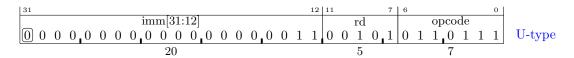
Load Upper Immediate.

 $rd \leftarrow zr(imm)$

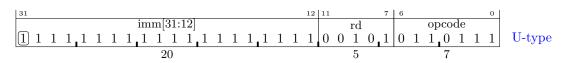
Copy the immediate value into bits 31:12 of the destination register and place zeros into bits 11:0. When XLEN is 64 or 128, the immediate value is sign-extended to the left.

Instruction Format and Example:

LUI to. 3



LUI to, 0xfffff



4.6.2 AUIPC rd, imm

Instruction!AUIPC Instruction!JAL

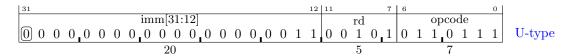
Add Upper Immediate to PC.

$$\texttt{rd} \leftarrow \texttt{pc} + \texttt{zr(imm)}$$

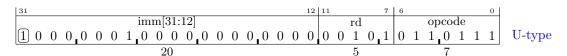
Create a signed 32-bit value by zero-extending imm[31:12] to the right (see subsection 4.2.4) and add this value to the pc register, placing the result into rd.

When XLEN is 64 or 128, the immediate value is also sign-extended to the left prior to being added to the pc register.

AUIPC t0, 3



AUIPC t0, 0x81000



The AUIPC instruction supports two-instruction sequences to access arbitrary offsets from the PC for both control-flow transfers and data accesses. The combination of an AUIPC and the 12-bit immediate in a JALR can transfer control to any 32-bit PC-relative address, while an AUIPC plus the 12-bit immediate offset in regular load or store instructions can access any 32-bit PC-relative data address. [1, p. 14]

4.6.3 JAL rd, imm

Jump and link.

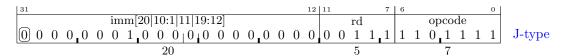
$$rd \leftarrow pc + 4$$

 $pc \leftarrow pc + sx(imm << 1)$

This instruction saves the address of the next instruction that would otherwise execute (located at pc+4) Instruction!JALR into rd and then adds immediate value to the pc causing an unconditional branch to take place.

The standard software conventions for calling subroutines use x1 as the return address (rd register in this case). [1, p. 16]

Encoding:



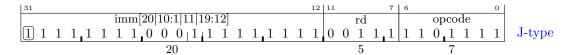
State of registers before execution:

$$pc = 0x111144444$$

State of registers after execution:

$$pc = 0x11114454 x7 = 0x11114448$$

JAL provides a method to call a subroutine using a pc-relative address.



imm demultiplexed value = 1111111111111111110002 $\ll 1 = -16_{10}$

State of registers before execution:

$$pc = 0x111144444$$

State of registers after execution:

$$pc = 0x11114434 x7 = 0x11114448$$

4.6.4 JALR rd, rs1, imm

Jump and link register.

$$\begin{array}{l} \texttt{rd} \leftarrow \texttt{pc} + \texttt{4} \\ \texttt{pc} \leftarrow (\texttt{rs1} + \texttt{sx(imm)}) \& ~\texttt{1} \end{array}$$

This instruction saves the address of the next instruction that would otherwise execute (located at pc+4) into rd and then adds the immediate value to the rs1 register and stores the sum into the pc register causing an unconditional branch to take place.

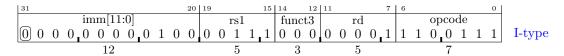
Note that the branch target address is calculated by sign-extending the imm[11:0] bits from the instruc-

tion, adding it to the rs1 register and then the LSB of the sum is to zero and the result is stored into Instruction!BEQ the pc register. The discarding of the LSB allows the branch to refer to any even address.

The standard software conventions for calling subroutines use x1 as the return address (rd register in this case). [1, p. 16]

Encoding:

JALR x1, x7, 4



Before:

pc = 0x11114444

x7 = 0x44444444

After

pc = 0x5555888c

x1 = 0x111114448

JALR provides a method to call a subroutine using a base-displacement address.

JALR x1, x0, 5

31 20	19	1	15	14 12	11	7	6 0	
imm[11:0]		rs1		funct3		rd	opcode	
$ \boxed{0} \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 $	0	0 0 0	0	0 0 0	0 0	$0 \ 0 \ 1$	1 1 0 0 1 1 1	I-type
12		5		3		5	7	

Note that the least significant bit in the result of rs1+imm is discarded/set to zero before the result is saved in the pc.

pc = 0x111144444

After

pc = 0x000000004

x1 = 0x11114448

4.6.5 BEQ rs1, rs2, imm

Branch if equal.

$$pc \leftarrow (rs1 == rs2) ? pc+sx(imm[12:1] << 1) : pc+4$$

Encoding:

BEQ x3, x15, 2064



Instruction!BNE Instruction!BLT

 $\begin{aligned} &\text{imm}[12\text{:}1] = 010000001000_2 = 1032_{10} \\ &\text{imm} = 2064_{10} \\ &\text{funct3} = 000_2 \\ &\text{rs1} = \text{x3} \\ &\text{rs2} = \text{x15} \end{aligned}$

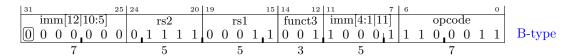
4.6.6 BNE rs1, rs2, imm

Branch if Not Equal.

 $pc \leftarrow (rs1 != rs2) ? pc+sx(imm[12:1] << 1) : pc+4$

Encoding:

BNE x3, x15, 2064



 $\mathrm{imm}[12\text{:}1] = 010000001000_2 = 1032_{10}$

 $imm = 2064_{10}$

 $funct3 = 001_2$

rs1 = x3

rs2 = x15

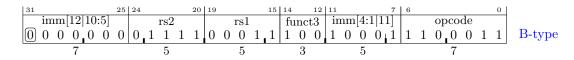
4.6.7 BLT rs1, rs2, imm

Branch if Less Than.

 $pc \leftarrow (rs1 < rs2) ? pc+sx(imm[12:1] << 1) : pc+4$

Encoding:

BLT x3, x15, 2064



 $imm[12:1] = 010000001000_2 = 1032_{10}$

 $\mathrm{imm} = 2064_{10}$

 $funct3 = 100_2$

rs1 = x3

rs2 = x15

4.6.8 BGE rs1, rs2, imm

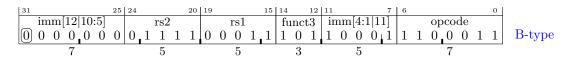
Instruction!BGE Instruction!BLTU Instruction!BGEU

Branch if Greater or Equal.

$$pc \leftarrow (rs1 \ge rs2) ? pc+sx(imm[12:1] << 1) : pc+4$$

Encoding:

BGE x3, x15, 2064



 $\mathrm{imm}[12:1] = 010000001000_2 = 1032_{10}$

 $imm = 2064_{10}$

 $funct3 = 101_2$

rs1 = x3

rs2 = x15

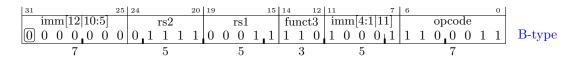
4.6.9 BLTU rs1, rs2, imm

Branch if Less Than Unsigned.

$$pc \leftarrow (rs1 < rs2) ? pc+sx(imm[12:1] << 1) : pc+4$$

Encoding:

BLTU x3, x15, 2064



 $\mathrm{imm}[12{:}1] = 010000001000_2 = 1032_{10}$

 $imm = 2064_{10}$

 $funct3 = 110_2$

rs1 = x3

rs2 = x15

4.6.10 BGEU rs1, rs2, imm

Branch if Greater or Equal Unsigned.

$$pc \leftarrow (rs1 \ge rs2) ? pc+sx(imm[12:1] << 1) : pc+4$$

Encoding:

BGEU x3, x15, 2064

- 1	31 25	24 20	19 15	14 12 11 7	6 0	
	imm[12 10:5]	rs2	rs1	funct3 imm[4:1 11]	opcode	
	$\boxed{0} \ 0 \ 0 \ 0 \ 0 \ 0 \ 0$	0,1 1 1 1	0 0 0 1 1	1 1 1 1 0 0 0 1	1 1 0 0 0 1 1	B-type
	7	5	5	3 5	7	

Instruction!LB Instruction!LH Instruction!LW

→ Fix Me:

use symbols in branch examples

```
\begin{aligned} &\text{imm}[12\text{:}1] = 010000001000_2 = 1032_{10} \\ &\text{imm} = 2064_{10} \\ &\text{funct3} = 111_2 \\ &\text{rs1} = \text{x3} \end{aligned}
```

4.6.11 LB rd, imm(rs1)

Load byte.

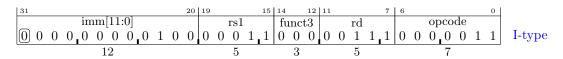
rs2 = x15

```
 \begin{array}{l} \texttt{rd} \leftarrow \texttt{sx}(\texttt{m8}(\texttt{rs1+sx}(\texttt{imm}))) \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}
```

Load an 8-bit value from memory at address rs1+imm, then sign-extend it to 32 bits before storing it in rd

Encoding:

LB x7, 4(x3)



4.6.12 LH rd, imm(rs1)

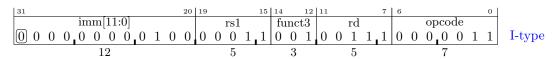
Load halfword.

$$\begin{array}{l} \texttt{rd} \leftarrow \texttt{sx}(\texttt{m16}(\texttt{rs1+sx}(\texttt{imm}))) \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

Load a 16-bit value from memory at address rs1+imm, then sign-extend it to 32 bits before storing it in rd

Encoding:

LH x7, 4(x3)



4.6.13 LW rd, imm(rs1)

Load word.

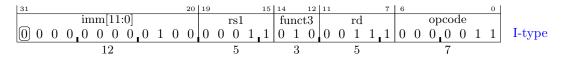
 $\begin{array}{l} \texttt{rd} \leftarrow \texttt{sx}(\texttt{m32}(\texttt{rs1+sx}(\texttt{imm}))) \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$

Instruction!LBU Instruction!LHU

Load a 32-bit value from memory at address rs1+imm, then store it in rd

Encoding:

LW
$$x7, 4(x3)$$



4.6.14 LBU rd, imm(rs1)

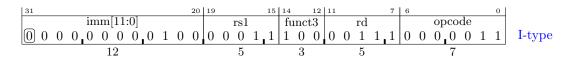
Load byte unsigned.

$$\begin{array}{l} \texttt{rd} \leftarrow \texttt{zx}(\texttt{m8}(\texttt{rs1+sx}(\texttt{imm}))) \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

Load an 8-bit value from memory at address rs1+imm, then zero-extend it to 32 bits before storing it in rd

Encoding:

LBU x7, 4(x3)



4.6.15 LHU rd, imm(rs1)

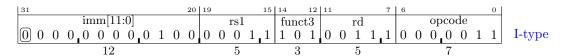
Load halfword unsigned.

$$\begin{array}{l} \texttt{rd} \leftarrow \texttt{zx}(\texttt{m16}(\texttt{rs1+sx}(\texttt{imm}))) \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

Load an 16-bit value from memory at address rs1+imm, then zero-extend it to 32 bits before storing it in rd

Encoding:

LHU x7, 4(x3)



4.6.16 SB rs2, imm(rs1)

Instruction!SB Instruction!SH Instruction!SW

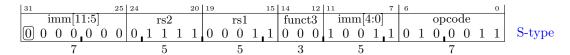
Store Byte.

$$\begin{array}{l} \texttt{m8(rs1+sx(imm))} \leftarrow \texttt{rs2[7:0]} \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

Store the 8-bit value in rs2[7:0] into memory at address rs1+imm.

Encoding:

SB x3, 19(x15)



4.6.17 SH rs2, imm(rs1)

Store Halfword.

$$\begin{array}{l} \texttt{m16(rs1+sx(imm))} \leftarrow \texttt{rs2[15:0]} \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

Store the 16-bit value in rs2[15:0] into memory at address rs1+imm.

Encoding:

SH x3, 19(x15)



4.6.18 SW rs2, imm(rs1)

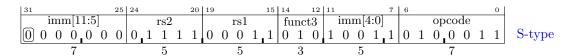
Store Word

$$\begin{array}{l} \texttt{m16(rs1+sx(imm))} \leftarrow \texttt{rs2[31:0]} \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

Store the 32-bit value in rs2 into memory at address rs1+imm.

Encoding:

SW x3, 19(x15)



Show pos & neg imm examples.

Instruction!ADDI Instruction!SLTI

4.6.19 ADDI rd, rs1, imm

Add Immediate

 $\begin{array}{l} \texttt{rd} \leftarrow \texttt{rs1+sx(imm)} \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$

Encoding:

ADDI x1, x7, 4



Before:

x7 = 0x111111111

After:

x1 = 0x111111115

4.6.20 SLTI rd, rs1, imm

Set LessThan Immediate

$$\begin{array}{l} \texttt{rd} \leftarrow (\texttt{rs1} < \texttt{sx(imm)}) \ ? \ 1 \ : \ 0 \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

If the sign-extended immediate value is less than the value in the rs1 register then the value 1 is stored in the rd register. Otherwise the value 0 is stored in the rd register.

Encoding:

SLTI x1, x7, 4



Before:

x7 = 0x111111111

After:

x1 = 0x000000000

4.6.21 SLTIU rd, rs1, imm

Instruction!SLTIU Instruction!XORI

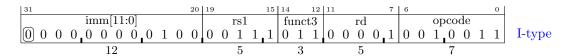
Set LessThan Immediate Unsigned

$$\begin{array}{l} rd \leftarrow (rs1 < sx(imm)) ? 1 : 0 \\ pc \leftarrow pc+4 \end{array}$$

If the sign-extended immediate value is less than the value in the rs1 register then the value 1 is stored in the rd register. Otherwise the value 0 is stored in the rd register. Both the immediate and rs1 register values are treated as unsigned numbers for the purposes of the comparison.³

Encoding:

SLTIU x1, x7, 4



Before:

x7 = 0x81111111

After:

x1 = 0x00000001

4.6.22 XORI rd, rs1, imm

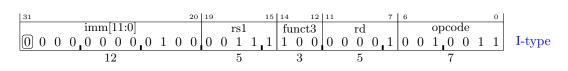
Exclusive Or Immediate

$$\begin{array}{l} \texttt{rd} \leftarrow \texttt{rs1} \ \hat{} \ \texttt{sx(imm)} \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

The logical XOR of the sign-extended immediate value and the value in the rs1 register is stored in the rd register.

Encoding:

XORI x1, x7, 4



Before:

x7 = 0x811111111

After:

x1 = 0x81111115

 $^{^3}$ The immediate value is first sign-extended to XLEN bits then treated as an unsigned number.[1, p. 14]

4.6.23 ORI rd, rs1, imm

Instruction!ORI Instruction!ANDI

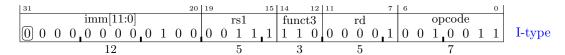
Or Immediate

$$\begin{array}{llll} rd \leftarrow rs1 & | & sx(imm) \\ pc \leftarrow pc+4 & & \end{array}$$

The logical OR of the sign-extended immediate value and the value in the rs1 register is stored in the rd register.

Encoding:

ORI x1, x7, 4



Before:

x7 = 0x81111111

After:

x1 = 0x81111115

4.6.24 ANDI rd, rs1, imm

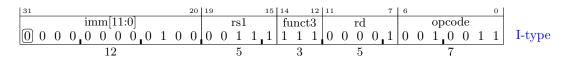
And Immediate

$$rd \leftarrow rs1 \& sx(imm) \\ pc \leftarrow pc+4$$

The logical AND of the sign-extended immediate value and the value in the rs1 register is stored in the rd register.

Encoding:

ANDI x1, x7, 4



Before:

x7 = 0x81111111

After:

x1 = 0x81111115

4.6.25 SLLI rd, rs1, shamt

Instruction!SLLI Instruction!SRLI Instruction!SRAI

Shift Left Logical Immediate

$$\begin{array}{l} rd \leftarrow rs1 << shamt \\ pc \leftarrow pc+4 \end{array}$$

SLLI is a logical left shift operation (zeros are shifted into the lower bits). The value in rs1 shifted left shamt number of bits and the result placed into rd. [1, p. 14]

Encoding:

SLLI x7, x3, 2



x3 = 0x811111111

After:

x7 = 0x04444444

4.6.26 SRLI rd, rs1, shamt

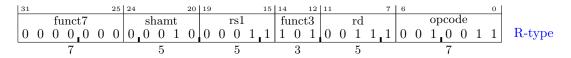
Shift Right Logical Immediate

$$\begin{array}{l} rd \leftarrow rs1 >\!\! > \text{ shamt} \\ pc \leftarrow pc +\! 4 \end{array}$$

SRLI is a logical right shift operation (zeros are shifted into the higher bits). The value in rs1 shifted right shamt number of bits and the result placed into rd. [1, p. 14]

Encoding:

SRLI x7, x3, 2



x3 = 0x811111111

After:

x7 = 0x204444444

4.6.27 SRAI rd, rs1, shamt

Shift Right Arithmetic Immediate

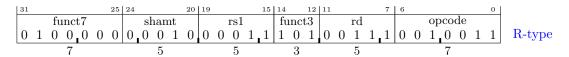
$$\begin{array}{l} \texttt{rd} \leftarrow \texttt{rs1} >\!\!\!> \texttt{shamt} \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

Instruction!ADD Instruction!SUB

SRAI is a logical right shift operation (zeros are shifted into the higher bits). The value in rs1 shifted right shamt number of bits and the result placed into rd. [1, p. 14]

Encoding:

SRAI x7, x3, 2



x3 = 0x81111111

After:

x7 = 0xe0444444

4.6.28 ADD rd, rs1, rs2

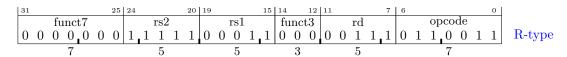
Add

$$\begin{array}{l} \texttt{rd} \leftarrow \texttt{rs1} + \texttt{rs2} \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

ADD performs addition. Overflows are ignored and the low 32 bits of the result are written to rd. [1, p. 15]

Encoding:

ADD x7, x3, x31



 $x3 = 0x811111111 \ x31 = 0x22222222$

After:

x7 = 0xa3333333

4.6.29 SUB rd, rs1, rs2

Subtract

$$\begin{array}{l} \texttt{rd} \leftarrow \texttt{rs1} - \texttt{rs2} \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

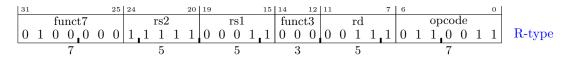
SUB performs subtraction. Underflows are ignored and the low 32 bits of the result are written to rd. [1,

p. 15]

Instruction!SLT Instruction!SLT

Encoding:

SUB x7, x3, x31



 $x3 = 0x83333333 \ x31 = 0x011111111$

After:

x7 = 0x82222222

4.6.30 SLL rd, rs1, rs2

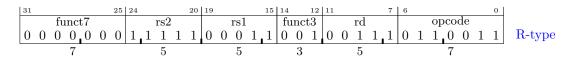
Shift Left Logical

$$\begin{array}{l} \texttt{rd} \leftarrow \texttt{rs1} & \texttt{<< rs2} \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

SLL performs a logical left shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2. [1, p. 15]

Encoding:

SLL x7, x3, x31



x3 = 0x83333333x31 = 0x00000002

After:

x7 = 0x0cccccc

4.6.31 SLT rd, rs1, rs2

Set Less Than

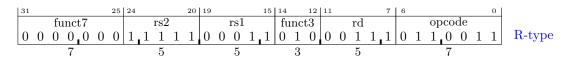
$$\begin{array}{l} \texttt{rd} \leftarrow (\texttt{rs1} < \texttt{rs2}) ? 1 : 0 \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

SLT performs a signed compare, writing 1 to rd if rs1; rs2, 0 otherwise. [1, p. 15]

Encoding:

SLT x7, x3, x31

Instruction!XOR



x3 = 0x83333333x31 = 0x00000002

After:

x7 = 0x00000001

4.6.32 SLTU rd, rs1, rs2

Set Less Than Unsigned

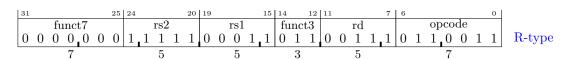
$$rd \leftarrow (rs1 < rs2) ? 1 : 0$$

$$pc \leftarrow pc+4$$

SLTU performs an unsigned compare, writing 1 to rd if rs1 ; rs2, 0 otherwise. Note, SLTU rd, x0, rs2 sets rd to 1 if rs2 is not equal to zero, otherwise sets rd to zero (assembler pseudo-op SNEZ rd, rs). [1, p. 15]

Encoding:

SLTU x7, x3, x31



x3 = 0x83333333x31 = 0x00000002

After:

x7 = 0x000000000

4.6.33 XOR rd, rs1, rs2

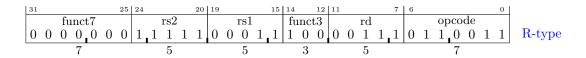
Exclusive Or

$$\begin{array}{l} \texttt{rd} \leftarrow \texttt{rs1} \ \hat{} \ \texttt{rs2} \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

XOR performs a bit-wise exclusive or on rs1 and rs2. The result is stored on rd.

Encoding:

XOR x7, x3, x31



Instruction!SRL Instruction!SRA

x3 = 0x83333333x31 = 0x1888ffff

After:

x7 = 0x9bbbcccc

4.6.34 SRL rd, rs1, rs2

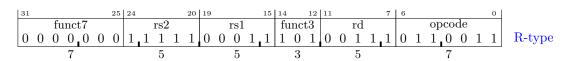
Shift Right Logical

$$\begin{array}{l} \texttt{rd} \leftarrow \texttt{rs1} >> \texttt{rs2} \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

SRL performs a logical right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2. [1, p. 15]

Encoding:

SRL x7, x3, x31



x3 = 0x83333333x31 = 0x00000010

After:

x7 = 0x00008333

4.6.35 SRA rd, rs1, rs2

Shift Right Arithmetic

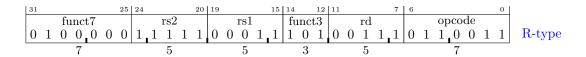
$$rd \leftarrow rs1 >> rs2$$

 $pc \leftarrow pc+4$

SRA performs an arithmetic right shift (the original sign bit is copied into the vacated upper bits) on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2. [1, p. 14, 15]

Encoding:

SLA x7, x3, x31



Instruction!OR Instruction!AND

x3 = 0x83333333x31 = 0x00000010

After:

x7 = 0xffff8333

4.6.36 OR rd, rs1, rs2

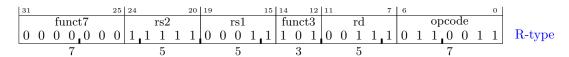
Or

$$\begin{array}{l} \texttt{rd} \leftarrow \texttt{rs1} \; \mid \; \texttt{rs2} \\ \texttt{pc} \leftarrow \texttt{pc+4} \end{array}$$

OR is a logical operation that performs a bit-wise OR on register rs1 and rs2 and then places the result in rd. [1, p. 14]

Encoding:

OR x7, x3, x31



x3 = 0x83333333x31 = 0x00000440

After:

x7 = 0x83333773

4.6.37 AND rd, rs1, rs2

And

$$rd \leftarrow rs1 \& rs2$$

 $pc \leftarrow pc+4$

AND is a logical operation that performs a bit-wise AND on register rs1 and rs2 and then places the result in rd. [1, p. 14]

Encoding:

AND x7, x3, x31



Instruction!FENCE Instruction!FENCE.I

x3 = 0x83333333x31 = 0x00000fe2

After:

x7 = 0x00000322

4.6.38 FENCE predecessor, successor

The FENCE instruction is used to order device I/O and memory accesses as viewed by other RISC-V harts and external devices or co-processors. Any combination of device input (I), device output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V hart or external device can observe any operation in the successor set following a FENCE before any operation in the predecessor set preceding the FENCE. The execution environment will define what I/O operations are possible, and in particular, which load and store instructions might be treated and ordered as device input and device output operations respectively rather than memory reads and writes. For example, memory-mapped I/O devices will typically be accessed with uncached loads and stores that are ordered using the I and O bits rather than the R and W bits. Instruction-set extensions might also describe new coprocessor I/O instructions that will also be ordered using the I and O bits in a FENCE. [1, p. 21]

Operation:

 $pc \leftarrow pc+4$

Encoding:

 $0\;0\;0\;1\;1\;1\;1\;0\;0\;0\;1\;1\;1\;1\;1\;1\;1\;1$

4.6.39 FENCE.I

The FENCE.I instruction is used to synchronize the instruction and data streams. RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches on the same RISC-V hart until a FENCE.I instruction is executed. A FENCE.I instruction only ensures that a subsequent instruction fetch on a RISC-V hart will see any previous data stores already visible to the same RISC-V hart. FENCE.I does not ensure that other RISC-V harts' instruction fetches will observe the local hart's stores in a multiprocessor system. To make a store to instruction memory visible to all RISC-V harts, the writing hart has to execute a data FENCE before requesting that all remote RISC-V harts execute a FENCE.I. [1, p. 21]

Operation:

 $\texttt{pc} \leftarrow \texttt{pc+4}$

Encoding:

 $0\ 0\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0$

4.6.40 ECALL

Instruction!ECALL Instruction!EBREAK Instruction!CSRRW Instruction!CSRRS

The ECALL instruction is used to make a request to the supporting execution environment, which is Instruction!CSRRS usually an operating system. The ABI for the system will define how parameters for the environment Instruction!CSRRC request are passed, but usually these will be in defined locations in the integer register file. [1, p. 24]

4.6.41 EBREAK

The EBREAK instruction is used by debuggers to cause control to be transferred back to a debugging environment. [1, p. 24]

4.6.42 CSRRW rd, csr, rs1

The CSRRW (Atomic Read/Write CSR) instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to XLEN bits, then writes it to integer register rd. The initial value in rs1 is written to the CSR. If rd=x0, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read. [1, p. 22]

 $1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0$

4.6.43 CSRRS rd, csr, rs1

The CSRRS (Atomic Read and Set Bits in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written). [1, p. 22]

If rs1=x0, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write, such as raising illegal instruction exceptions on accesses to read-only CSRs. Note that if rs1 specifies a register holding a zero value other than x0, the instruction will still attempt to write the unmodified value back to the CSR and will cause any attendant side effects. [1, p. 22]

 $1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0$

4.6.44 CSRRC rd, csr, rs1

The CSRRC (Atomic Read and Clear Bits in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected. [1, p. 22]

If rs1=x0, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write, such as raising illegal instruction exceptions on accesses to read-only CSRs. Note that if rs1 specifies a register holding a zero value other than x0, the instruction will still attempt to write the unmodified value back to the CSR and will cause any attendant side effects. [1, p. 22]

Instruction!CSRRWI Instruction!CSRRSI Instruction!CSRRCI RV32M Instruction!MUL

 $1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0$

4.6.45 CSRRWI rd, csr, imm

This instruction is the same as CSRRW except a 5-bit unsigned (zero-extended) immediate value is used rather than the value from a register.

 $1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0$

4.6.46 CSRRSI rd, csr, rs1

This instruction is the same as CSRRS except a 5-bit unsigned (zero-extended) immediate value is used rather than the value from a register.

If the uimm[4:0] field is zero, then this instruction will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write. For CSRRWI, if rd=x0, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read. [1, p. 22]

4.6.47 CSRRCI rd, csr, rs1

This instruction is the same as CSRRC except a 5-bit unsigned (zero-extended) immediate value is used rather than the value from a register.

If the uimm[4:0] field is zero, then this instruction will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write. For CSRRWI, if rd=x0, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read. [1, p. 22]

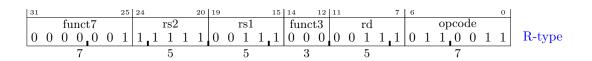
4.7 RV32M Standard Extension

32-bit integer multiply and divide instructions.

4.7.1 MUL rd, rs1, rs2

Multiply rs1 by rs2 and store the least significant 32-bits of the result in rd.

MUL x7, x3, x31



Instruction!MULH Instruction!MULHS Instruction!MULHU Instruction!DIV Instruction!DIVU

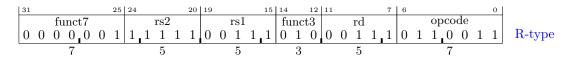
4.7.2 MULH rd, rs1, rs2

MULH x7, x3, x31

1	31 25	24 20	19 15	14 12 11 7	6 0	
	funct7	rs2	rs1	funct3 rd	opcode	
	0 0 0 0 0 0 1	1,1 1 1 1	0 0 1 1 1	0 0 1 0 0 1 1 1	0 1 1 0 0 1 1	R-type
	7	5	5	3 5	7	

4.7.3 MULHS rd, rs1, rs2

MULHS x7, x3, x31



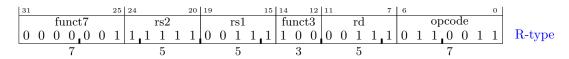
4.7.4 MULHU rd, rs1, rs2

MULHU x7, x3, x31



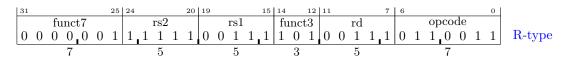
4.7.5 DIV rd, rs1, rs2

DIV x7, x3, x31



4.7.6 DIVU rd, rs1, rs2

DIVU x7, x3, x31



4.7.7 REM rd, rs1, rs2

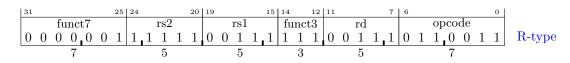
REM x7, x3, x31

31						25	24				20	19				15	14		12	11				7	6						0	
		fı	ınc	:t7					rs2	2				rs1			fu	ınc	t3			rd					or	со	de			
0	0	0	0	.0	0	1	1	.1	1	1	1	0	0	1	1	1	1	1	0	0	0	1	1	1	0	1	1	0	0	1	1	R-type
			7						5					5				3				5						7				

Instruction!REM Instruction!REMU RV32A RV32F RV32D

4.7.8 REMU rd, rs1, rs2

REMU x7, x3, x31



4.8 RV32A Standard Extension

32-bit atomic operations.

4.9 RV32F Standard Extension

 $32\mbox{-bit}$ IEEE floating point instructions.

4.10 RV32D Standard Extension

64-bit IEEE floating point instructions.

Appendix A

Attribution 4.0 International

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.

Using Creative Commons Public Licenses

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.

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

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

Creative Commons Attribution 4.0 International Public License

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.

Section 1. Definitions

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

- c. Copyright and Similar Rights means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.
- d. Effective Technological Measures means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.
- e. Exceptions and Limitations means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.
- f. Licensed Material means the artistic or literary work, database, or other material to which the Licensor applied this Public License
- g. Licensed Rights means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.
- h. Licensor means the individual(s) or entity(ies) granting rights under this Public License.
- i. Share means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.
- j. Sui Generis Database Rights means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.
- k. You means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding meaning.

Section 2. Scope

- a. License grant.
 - 1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:
 - a. reproduce and Share the Licensed Material, in whole or in part; and
 - b. produce, reproduce, and Share Adapted Material.
 - 2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.
 - 3. Term. The term of this Public License is specified in Section 6(a).
 - 4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a) (4) never produces Adapted Material.
 - 5. Downstream recipients
 - a. Offer from the Licensor Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.
 - b. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.
 - 6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).
- b. Other rights.
 - 1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.
 - 2. Patent and trademark rights are not licensed under this Public License.
 - 3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3. License Conditions

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

a. Attribution.

- 1. If You Share the Licensed Material (including in modified form), You must:
 - a. retain the following if it is supplied by the Licensor with the Licensed Material:
 - i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);
 - ii. a copyright notice;
 - iii. a notice that refers to this Public License;
 - iv. a notice that refers to the disclaimer of warranties;
 - v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;
 - b. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and
 - c. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.
- 2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.
- 3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.
- 4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4. Sui Generis Database Rights

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

- a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;
- b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and
- c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

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

Section 5. Disclaimer of Warranties and Limitation of Liability

- a. UNLESS OTHERWISE SEPARATELY UNDERTAKEN BY THE LICENSOR, TO THE EXTENT POSSIBLE, THE LICENSOR OFFERS THE LICENSED MATERIAL AS-IS AND AS-AVAILABLE, AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE LICENSED MATERIAL, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHER. THIS INCLUDES, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OR ABSENCE OF ERRORS, WHETHER OR NOT KNOWN OR DISCOVERABLE. WHERE DISCLAIMERS OF WARRANTIES ARE NOT ALLOWED IN FULL OR IN PART, THIS DISCLAIMER MAY NOT APPLY TO YOU.
- b. TO THE EXTENT POSSIBLE, IN NO EVENT WILL THE LICENSOR BE LIABLE TO YOU ON ANY LEGAL THE-ORY (INCLUDING, WITHOUT LIMITATION, NEGLIGENCE) OR OTHERWISE FOR ANY DIRECT, SPECIAL, INDIRECT, INCIDENTAL, CONSEQUENTIAL, PUNITIVE, EXEMPLARY, OR OTHER LOSSES, COSTS, EXPENSES, OR DAMAGES ARISING OUT OF THIS PUBLIC LICENSE OR USE OF THE LICENSED MATERIAL, EVEN IF THE LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSSES, COSTS, EXPENSES, OR DAMAGES. WHERE A LIMITATION OF LIABILITY IS NOT ALLOWED IN FULL OR IN PART, THIS LIMITATION MAY NOT APPLY TO YOU.
- c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6. Term and Termination

- a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.
- b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:
 - 1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or
 - 2. upon express reinstatement by the Licensor.

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

- c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.
- d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7. Other Terms and Conditions

- a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.
- b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8. Interpretation

- a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.
- b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.
- c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.
- d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

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

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

Bibliography

- [1] RISC-V Foundation, The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2, 5 2017. Editors Andrew Waterman and Krste Asanović. vi, 33, 34, 35, 36, 43, 45, 46, 47, 48, 49, 50, 51, 52, 53
- [2] D. Patterson and A. Waterman, *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 11 2017. ISBN: 978-0999249116. vi
- [3] D. Patterson and J. Hennessy, Computer Organization and Design RISC-V Edition: The Hardware Software Interface. Morgan Kaufmann, 4 2017. ISBN: 978-0128122754. vi, 4
- [4] W. F. Decker, "A modern approach to teaching computer organization and assembly language programming," SIGCSE Bull., vol. 17, pp. 38–44, 12 1985. vi
- [5] Texas Instruments, SN54190, SN54191, SN54LS190, SN54LS191, SN74190, SN74191, SN74LS190, SN74LS191 Synchronous Up/Down Counters With Down/Up Mode Control, 3 1988. vi
- [6] Texas Instruments, SN54154, SN74154 4-line to 16-line Decoders/Demultiplexers, 12 1972. vi
- [7] Intel, MCS-85 User's Manual, 9 1978. vi
- [8] Radio Shack, TRS-80 Editor/Assembler Operation and Reference Manual, 1978. vi
- [9] Motorola, MC68000 16-bit Microprocessor User's Manual, 2nd ed., 1 1980. MC68000UM(AD2). vi
- [10] R. A. Overbeek and W. E. Singletary, Assembler Language With ASSIST. Science Research Associates, Inc., 2nd ed., 1983. vi
- [11] IBM, IBM System/370 Principals of Operation, 7th ed., 3 1980. vi
- [12] IBM, OS/VS-DOS/VSE-VM/370 Assembler Language, 6th ed., 3 1979. vi
- [13] D. Cohen, "IEN 137, On Holy Wars and a Plea for Peace," Apr. 1980. This note discusses the Big-Endian/Little-Endian byte/bit-order controversy, but did not settle it. A decade later, David V. James in "Multiplexed Buses: The Endian Wars Continue", IEEE Micro, 10(3), 9–21 (1990) continued the discussion.
- [14] R. M. Stallman and the GCC Developer Community, Using the GNU Compiler Collection (For GCC version 7.3.0). Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA: GNU Press, 2017. 23
- [15] RISC-V Foundation, The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.10, 5 2017. Editors Andrew Waterman and Krste Asanović.
- [16] P. Dabbelt, S. O'Rear, K. Cheng, A. Waterman, M. Clark, A. Bradbury, D. Horner, M. Nordlund, and K. Merker, RISC-V ELF psABI specification, 2017.
- [17] National Semiconductor Corporation, Series 32000 Databook, 1986.

Index

C CPU, 2 I Instruction ADD, 46 ADDI, 42 AND, 50 ANDI, 44 AUIPC, 34 BEQ, 36 BGE, 38 BGEU, 38 BLT, 37 BLTU, 38 BNE, 37 CSRRC, 52 CSRRCI, 53 CSRRS, 52 CSRRSI, 53 CSRRW, 52 CSRRWI, 53 DIV, 54 DIVU, 54 EBREAK, 52 ECALL, 52 FENCE, 51 FENCE.I, 51 JAL, 34 JALR, 35 LB, 39 LBU, 40 LH, 39 LHU, 40 LUI, 33 LW, 39 MULH, 54 MULHS, 54 MULHS, 54 MULHU, 54 OR, 50	SH, 41 SLL, 47 SLLI, 45 SLT, 47 SLTI, 42 SLTIU, 43 SLTU, 48 SRA, 49 SRAI, 45 SRL, 49 SRLI, 45 SUB, 46 SW, 41 XOR, 48 XORI, 43 instruction cycle, 3 instruction fetch, 4 ISA, 2 R register, 1 RV32, 24 RV32A, 3, 55 RV32C, 3 RV32D, 3, 55 RV32F, 3, 55 RV32G, 3 RV32I, 3, 33 RV32M, 3, 53 RV32Q, 3 rvddt, 6
MULH, 54 MULHS, 54 MULHU, 54	

Glossary

```
alignment Refers to a range of numeric values that begin at a multiple of some number. Primairly
     used when referring to a memory address. For example an alignment of two refers to one or more
     addresses starting at even address and continuing onto subsequent adjacent, increasing memory
     addresses. 4, 62
big endian A number format where the most significant values are printed to the left of the lesser
     significant values. This is the method that everyone used to write decimal numbers every day. 7,
     62
binary Something that has two parts or states. In computing these two states are represented by the
     numbers one and zero or by the conditions true and false and can be stored in one bit. 62
bit One binary digit. 4, 10, 62
byte A binary value represented by 8 bits. 4, 62
CPU Central Processing Unit. 1, 2, 62
Doubleword A binary value represented by 64 bits. 62
exception An error encountered by the CPU while executing an instruction that can not be completed.
     4, 62
Fullword A binary value represented by 32 bits. 62
Halfword A binary value represented by 16 bits. 62
hart Hardware Thread. 3, 62
hexadecimal A base-16 numbering system whose digits are 0123456789abcdef. The hex digits (hits)
     are not case-sensitive. 7, 62
High order bits Some number of MSBs. 62
hit One hex digit. 10, 11, 62
ISA Instruction Set Architecture. 2, 62
LaTeX Is a mark up language specially suited for scientific documents. 62
```

address A numeric value used to uniquely identify each byte of main memory. 4, 62

little endian A number format where the least significant values are printed to the left of the more significant values. This is the opposite ordering that everyone learns in grade school when learning how to count. For example a big endian number written as "1234" would be written in little endian form as "4321". 62

Low order bits Some number of LSBs. 62

LSB Least Significant Bit. 12, 25, 28, 30, 62

machine language The instructions that are executed by a CPU that are expressed in the form of binary values. 1, 62

mneumonic A method used to remember something. In the case of assembly language, each machine instruction is given a name so the programmer need not memorize the biary values of each machine instruction. 1, 62

MSB Most Significant Bit. 12, 13, 24, 25, 30, 62

overflow The situation where the result of an addition or subtraction operation is approaching positive or negative infinity and exceeds the number of bits alloted to contain the result. This is typically caused by high-order truncation. 13, 17, 62

program A ordered list of one or more instructions. 1, 62

Quadword A binary value represented by 128 bits. 62

register A unit of storage inside a CPU with the capacity of XLEN bits. 1, 62

RV32 Short for RISC-V 32. The number 32 refers to the XLEN. 33, 62

RV64 Short for RISC-V 64. The number 64 refers to the XLEN. 62

rvddt A RV32I simulator and debugging tool inspired by the simplicity of the Dynamic Debugging Tool (ddt) that was part of the CP/M operating system. 6, 62

underflow The situation where the result of an addition or subtraction operation is approaching zero and exceeds the number of bits alloted to contain the result. This is typically caused by low-order truncation. 17, 62

XLEN The number of bits a RISC-V x integer register (such as x0). For RV32 XLEN=32, RV64 XLEN=64 etc. 29, 30, 62