RISC-V
Assembly Language Programming

(Draft v0.2-0-gfd5c875)

John Winans
jwinans@niu.edu

May 28, 2018

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.

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

Contents

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 mnemonic can be used to specify each machine instruction and any of its parameters... rather than require that programs be expressed as a series of binary values. A set of mnemonics, parameters and rules for specifying their use for the purpose of programming a CPU is called an *Assembly Language*.

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

register
CPU

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 registers are used to hold individual data values that can be manipulated by the instructions that are executed by the CPU.

Another type of volatile storage is main memory (sometimes called RAM) 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.

To keep track of the data values, each register is assigned a number and the main memory is broken up into small blocks called bytes that are also each assigned number called an address (an address is often referred to as a *location.*

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, ROM 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 manipulate the register data and to exchange data and instructions with the storage system. The instructions that are read from the main memory tell the CPU to perform various mathematic and logical operations on the data in its registers and where to save the results of those operations.

➽ Fix Me:
Add a block diagram of the CPU components described here.

1.1.2.1 Execution Unit

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

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

ALU
register
hart

1.1.2.2 Arithmetic and Logic Unit

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

1.1.2.3 Registers

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

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

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

The number of bits in each register is defined by the Instruction Set Architecture (ISA).

Fix Me:
Say something about XLEN?

1.1.2.4 Harts

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

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

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

1.1.3 Peripherals

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.

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

1.2 Instruction Set Architecture

ISA
RV32I
RV32M
RV32A
RV32F
RV32D
RV32Q
RV32C
RV32G
instruction cycle

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 can recognize and how it will process each one.

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

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

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).[1, p. 4]

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

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

1.2.2 Extension Modules

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

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 How the CPU Executes a Program

The process of executing a program is continuously repeating series of *instruction cycles* that are each comprised of a *fetch*, *decode* and *execute* phase.

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

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

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

1.3.1 Instruction Fetch

instruction fetch
instruction decode
instruction execute

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.

Given that the main memory is broken up and that each of its bytes is assigned an address, the `pc` is used to hold the address of the location where the next instruction to execute is located.

Given an instruction address, the CPU can request that the main memory locate and return the value of the data stored there using what is called a *memory read* operation and then the CPU can treat that *fetched* value as an instruction and execute it.[3]

1.3.2 Instruction Decode

Once an instruction has been fetched, it must be inspected to determine what operation(s) are to be performed. This primairly boils down to inspecting the portions of the instruction that dictate which registers are involved and, if the ALU is required, what it should do.

1.3.3 Instruction Execute

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.

Most of the time an instruction will complete by indicating that the CPU should proceed to fetch and execute the instruction at the next larger main memory address. In these cases the `pc` is incremented to point to the memory address after the current instruction.

Any parameters that an instruction requires must either be part of the instruction itself or read from (or stored into) one or more of the general purpose registers.

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 RISC-V 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.[4]

Once the instruction execution phase has completed, the next instruction cycle will be performed using the new value in the `pc` register.

[3] RV32I instructions are more than one byte in size, but this general description is suitable for now.
[4] This is the fundamental method used by a CPU to make decisions.

Chapter 2

Numbers and Storage Systems

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

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

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

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

2.1 Boolean Functions

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

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

The reason for this relaxation is because, while a single binary digit (bit) can represent the two values zero and one, the vast majority of the time data is processed by the CPU in groups of bits. These groups have names like byte (8 bits), halfword (16 bits) and fullword (32 bits).

2.1.1 NOT

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

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

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

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

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

A	$\overline{A}$
0	1
1	0

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

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

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

```
~ 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`

2.1.2 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 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).

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

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

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

An eight-bit example:

```
  1 1 1 1 0 1 0 1  <== A
& 1 0 0 1 0 0 1 1  <== B
-----------------
  1 0 0 1 0 0 0 1  <== output
```

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

2.1.3 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 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).

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

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

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

An eight-bit example:

```
  1 1 1 1 0 1 0 1  <== A
| 1 0 0 1 0 0 1 1  <== B
-----------------
  1 1 1 1 0 1 1 1  <== output
```

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

2.1.4 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.

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.

In mathamatical notion, the *xor* operator is expressed using the plus in a circle ($\oplus$).

A	B	A$\oplus$B
0	0	0
0	1	1
1	0	1
1	1	0

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

An eight-bit example:

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

Figure 2.1: Counting in decimal, binary and hexadecimal.

```
  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 Integers and Counting

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

Figure 2.1 shows an abridged table of the decimal, binary and hexadecimal values ranging from 0_{10} to 129_{10}.

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.) For example to interpret the decimal value on the fourth row:

Most significant bit
MSB—see Most significant bit
Least significant bit
LSB—see Least significant bit

$$0 \times 10^2 + 0 \times 10^1 + 3 \times 10^0 = 3_{10} \tag{2.2.1}$$

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

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

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

$$0 \times 16^1 + 3 \times 16^0 = 3_{10} \tag{2.2.3}$$

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

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

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 (aka nybble). 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

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

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.

2.2.1 Converting Between Bases

2.2.1.1 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:

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

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}.

2.2.1.2 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:

```
Base-2 place values:   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           A           E
```

2.2.1.3 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 section 2.2.1.1.

For example:

```
Hex:                      7          C
Decimal Sum:        4+2+1=7  8+4    =12
Binary:           0 1 1 1    1 1 0 0
```

2.2.1.4 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:

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

   0          2048       (too big)
   1   1234 - 1024 = 210
   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 - 2    = 0
   0           1         (too big)
```

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

2.2.1.5 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.

2.2.2 Addition of Binary Numbers

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

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

For example:

```
       111111  1111  <== carries
    0110101111001111 <== addend
  + 0000011101100011 <== addend
  ------------------
    0111001100110010 <== sum
```

2.2.3 Signed Numbers

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

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

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

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

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

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

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

Calculating $4+5=9$

```
   1    <== carries
 000100 <== 4
+000101 <== 5
     ------
 001001 <== 9
```

Calculating $-4+-5=-9$

```
1 11     <== carries
  111100 <== -4
 +111011 <== -5
   ---------
1 110111 <== -9 (with a truncation)
```

```
 -32 16 8 4 2 1
   1  1 0 1 1 1
-32 + 16 + 4 + 2 + 1 = -9
```

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

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

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

2.2.3.1 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

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

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

```
  -128 64 32 16  8  4  2  1
     1  1  1  1  1           <== carries
     1  1  1  1  1  1  0  0 <== -4
   + 0  0  0  0  0  1  0  1 <== 5
     ----------------------
  1  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  0  1 <== plus 1
      -----------------------
      0  0  0  0  0  1  0  0 <== 4
```

2.2.4 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 $-4 - 8 = -12$

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


                  1  1  1    <== carries
      1  1  1  1  0  1  1  1 <== one's complement of -8
    + 0  0  0  0  0  0  0  1 <== plus 1
      -----------------------
      1  1  1  1  1  0  0  0 <== -8


      1  1  1  1             <== carries
      1  1  1  1  1  1  0  0 <== -4
    + 1  1  1  1  1  0  0  0 <== -8
      -----------------------
   1  1  1  1  1  0  1  0  0 < == -12
```

2.2.5 Truncation and Overflow

Discuss the details of truncation and overflow here.

I prefer to define *truncation* as the loss of data as result of the bit-length of the destination being too small to hold result of an operation and *overflow* as when the carry into a sign bit is not the same as the carry out of the sign bit.

Where addition and subtraction on the RV32 is concerned, the sum or difference of two unsigned 32-bit numbers will be *truncated* when the operation results in a carry out of bit 31. Unsinged operations can not overflow (as defined above).

(show a truncation picture here)

An Overflow occurs with signed numbers when the two addends are positive and sum is negative or the addends are both negative and the sum is positive.

(show an overflow picture here)

➽ 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.

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

(show mixed overflow and truncation situations here to drive home the need to ignore truncation when dealing with signed numbers.)

sign extension

0xffffffff + 0x00000002 has truncation but not overflow (OK for signed, not OK for unsigned).

0xffffffff + 0xfffffffe also has truncation but not overflow.

0x40000000 + 0x40000000 has overflow but not truncation. (We care if are signed numbers.)

0x80000000 + 0x80000000 has both overflow and truncation. (we care regardless of signedness)

Where subtraction is concerned the notion of a borrow is the same as carry.

Page 13 of [1] mixes these two notions of (and never mentions the word *truncate*) like this:

Fix Me:
I think that overloading the word overflow *like this can be is confusing to new programmers.*

> We did not include special instruction set support for overflow checks on integer arithmetic operations in the base instruction set, as many overflow checks can be cheaply implemented using RISC-V branches. Overflow checking for unsigned addition requires only a single additional branch instruction after the addition: `add t0, t1, t2; bltu t0, t1, overflow`.
>
> For signed addition, if one operand's sign is known, overflow checking requires only a single branch after the addition: `addi t0, t1, +imm; blt t0, t1, overflow`. This covers the common case of addition with an immediate operand.
>
> For general signed addition, three additional instructions after the addition are required, leveraging the observation that the sum should be less than one of the operands if and only if the other operand is negative.

```
add t0, t1, t2
slti t3, t2, 0
slt t4, t0, t1
bne t3, t4, overflow
```

> In RV64, checks of 32-bit signed additions can be optimized further by comparing the results of ADD and ADDW on the operands.

2.3 Sign and Zero Extension

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

```
                                1111 <== -1
                            11111111 <== -1
                    1111111111111111 <== -1
            11111111111111111111111111111111 <== -1
```

As do these:

```
                                01100 <== 12
                            0000001100 <== 12
    00000000000000000000000000001100 <== 12
```

The phenomenon illustrated here is called *sign extension*. That is any signed number can have any quantity of additional MSBs added to it, provided that they repeat the value of the sign bit.

Figure 2.2 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.

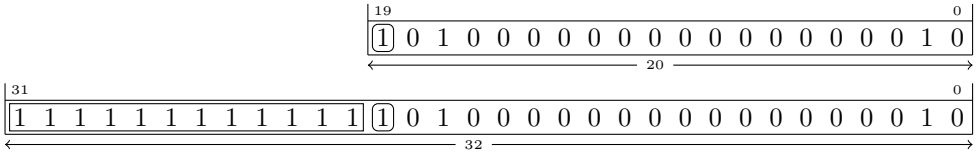


Figure 2.2: Sign-extending a negative integer from 20 bits to 32 bits.

Figure 2.3 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.

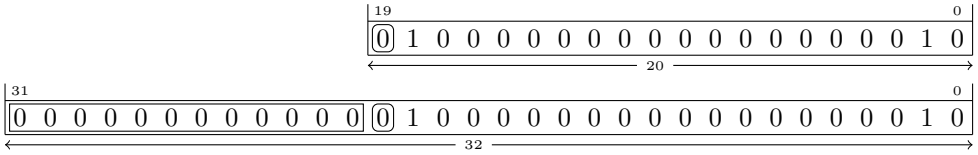


Figure 2.3: Sign-extending a positive integer from 20 bits to 32 bits.

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

```
                                1111 <== 15
                               01111 <== 15
00000000000000000000000000001111 <== 15
```

The observation here is that any *unsigned* number may be *zero extended* to any size.

Figure 2.4 illustrates zero-extending a 20-bit *val* to the left to form a 32-bit fullword.

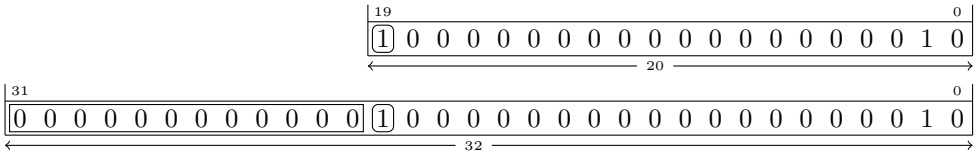


Figure 2.4: Zero-extending an unsigned integer from 20 bits to 32 bits.

2.4 Shifting

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

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

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

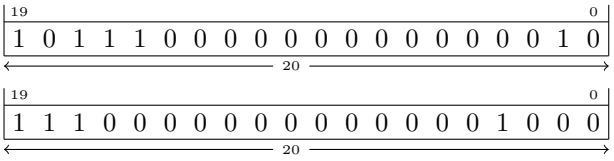
Note that when dealing with numeric values, any truncation performed during a right-shift will manifest itself as rounding toward zero.

2.4.1 Logical Shifting

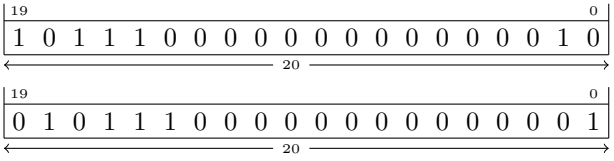
Shifting *logically* to the left or right is a matter of re-aligning the bits in a register and trncating the result.

To shift left two positions:

Fix Me:
Redraw these with arrows tracking the shifted bits and the truncated values



To shift right one position:



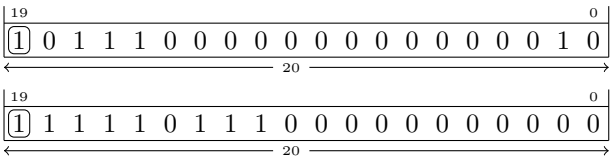
Note that the vacated bit positions are always filled with zero.

2.4.2 Arithmetic Shifting

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

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

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



2.5 Main Memory Storage

As mentioned in section 1.1.1.1, the main memory in a RISC-V system is byte-addressable. For that reason we will visualize it by displaying ranges of bytes displayed in hex and in ASCII. As will become obvious, the ASCII part makes it easier to find text messages.[3]

2.5.1 Memory Dump

Listing 2.1 shows a memory dump from the rvddt 'd' command requesting a dump starting at address `0x00002600` for the default quantity (`0x100`) of bytes.

Listing 2.1: `rvddt_memdump.out`
`rvddt` memory dump

```
1  ddt> d 0x00002600
2   00002600: 93 05 00 00 13 06 00 00-93 06 00 00 13 07 00 00 *................*
3   00002610: 93 07 00 00 93 08 d0 05-73 00 00 00 63 54 05 02 *........s...cT..*
4   00002620: 13 01 01 ff 23 24 81 00-13 04 05 00 23 26 11 00 *....#$......#&..*
5   00002630: 33 04 80 40 97 00 00 00-e7 80 40 01 23 20 85 00 *3..@......@.# ..*
6   00002640: 6f 00 00 00 6f 00 00 00-b7 87 00 00 03 a5 07 43 *o...o...........C*
7   00002650: 67 80 00 00 00 00 00 00-76 61 6c 3d 00 00 00 00 *g.......val=....*
8   00002660: 00 00 00 00 80 84 2e 41-1f 85 45 41 80 40 9a 44 *.......A..EA.@.D*
9   00002670: 4f 11 f3 c3 6e 8a 67 41-20 1b 00 00 20 1b 00 00 *O...n.gA ... ...*
10  00002680: 44 1b 00 00 14 1b 00 00-14 1b 00 00 04 1c 00 00 *D...............*
11  00002690: 44 1b 00 00 14 1b 00 00-04 1c 00 00 14 1b 00 00 *D...............*
12  000026a0: 44 1b 00 00 10 1b 00 00-10 1b 00 00 10 1b 00 00 *D...............*
13  000026b0: 04 1c 00 00 54 1f 00 00-54 1f 00 00 d4 1f 00 00 *....T...T.......*
14  000026c0: 4c 1f 00 00 4c 1f 00 00-34 20 00 00 d4 1f 00 00 *L...L...4 ......*
15  000026d0: 4c 1f 00 00 34 20 00 00-4c 1f 00 00 d4 1f 00 00 *L...4 ..L.......*
16  000026e0: 48 1f 00 00 48 1f 00 00-48 1f 00 00 34 20 00 00 *H...H...H...4 ..*
17  000026f0: 00 01 02 02 03 03 03 03-04 04 04 04 04 04 04 04 *................*
```

ℓ 1 The rvddt prompt shwing the dump command.

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

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

2.5.2 Endianness

The choice of which end of a multi-byte value is to be stored at the lowest byte address is referred to as *endianness.* For example, if a CPU were to store a halfword into memory, should the byte containing

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

the Most Significant Bit (MSB) (the *big* end) go first or does the byte with the Least Significant Bit (LSB) (the *little* end) go first/into the lowest memory address?

On the one hand the choice is arbitrairy. On the other hand, it is possible that the choice could impact the performance of the system.[4]

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

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

2.5.2.1 Big-Endian

Using the contents of Listing 2.1, a *big-endian* CPU would recognize the contents as follows:

- The 8-bit value stored at address `0x00002658` is `0x76`.
- The 16-bit value stored at address `0x00002658` is `0x7661`.
- The 32-bit value stored at address `0x00002658` is `0x76616c3d`.

Observe that the bytes in the dump are in the same order as they would be used by the CPU if it were to read them as a multi-byte value.

2.5.2.2 Little-Endian

Using the contents of Listing 2.1, a *little-endian* CPU would recognize the contents as follows:

- The 8-bit value stored at address `0x00002658` is `0x76`.
- The 16-bit value stored at address `0x00002658` is `0x6176`.
- The 32-bit value stored at address `0x00002658` is `0x3d6c6176`.

Observe that the bytes in the dump are in backwards order as they would be used by the CPU if it were to read them as a multi-byte value.

Note that in a little-endian system, the number of bytes used to represent the value does not change the place value of the first byte(s). In this example, the `0x76` at address `0x00002658` is the least significant byte in all representations.

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

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

2.5.3 Arrays and Character Strings

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

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

In an array, the elements are stored adjacent to one another such that the address of any element may be defined as:

$$e = a + n * s \tag{2.5.1}$$

Where n is the element number of interest, e is the address of element of interest, a is the address of the first element in the array, s is the size of each element, $a[0]$ is the first element of the array and $a[n-1]$ is the last element of the array.[5]

Using this definition, Listing 2.1, knowledge that we are using a little-endian machine and given that $a =$`0x00002656` and $s = 2$, the values of the first 8 elements of array a are:

- $a[0]$ is `0x0000` and is stored at `0x00002656`.
- $a[1]$ is `0x6176` and is stored at `0x00002658`.
- $a[2]$ is `0x3d6c` and is stored at `0x0000265a`.
- $a[3]$ is `0x0000` and is stored at `0x0000265c`.
- $a[4]$ is `0x0000` and is stored at `0x00002660`.
- $a[5]$ is `0x0000` and is stored at `0x00002662`.
- $a[6]$ is `0x8480` and is stored at `0x00002664`.
- $a[7]$ is `0x412e` and is stored at `0x00002666`.

As a general rule, there is no fixed rule or notion as to how many elements an array has. It is up to the programmer to ensure that the starting address and the nubmer of elements in any given array (its size) are used properly so that data bytes outside an array are not accidently used as elements.

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

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

In Listing 2.1, the 5-byte long array starting at address `0x00002658` contains a string whose value can be expressed as either of:

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

- `76 61 6c 3d 00`
- `"val="`

ASCII
ASCIIZ

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

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

It is a common mistake to create a string with a missing null terminator. The result of printing such a "string" is that the string is printed and as well as whatever random data bytes in memory that follows it until a byte whose value is zero is found by chance.

2.5.4 Context is Important!

Data values can be interpreted differently depending on the context in which they are used. Assuming what a set of bytes is used for based on their contents can be very misleading! For example, there is a 0x76 at address 0x00002658. This is a 'v' is you use it as an ASCII (see Appendix E) character, a 118_{10} if it is an integer value and TRUE if it is a conditional.

2.5.5 Alignment

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

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

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

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

Such alignments are important when exchanging data between the CPU and memory because the hardware implementations are optimized to transfer aligned data. Therefore, aligning data used by any program will reap the benefit of running faster.

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

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

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

Some CPUs 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. Such differences in hardware typically impact the cost and

911 performance of a system.[6]

912 2.5.6 Instruction Alignment

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

914 Every possible instruction that an RV32I CPU can execute contains exactly 32 bits. Therefore they
915 are always stored on a full word boundary. Any *unaligned* instruction is *illegal*.[7]

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

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

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

Chapter 3

The Elements of a Assembly Language Program

3.1 Assembly Language Statements

Introduce the assembly language grammar. Statement = 1 line of text containing an instruction or directive.

Instruction = label, mnemonic, operands, comment.

Directive = Used to control the operation of the assembler.

3.2 Memory Layout

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

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

3.3 A Sample Program Source Listing

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

Listing 3.1: `zero4regs.S`
Setting four registers to zero.

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

This program listing illustrates a number of things: rvddt

- 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.[1])
- A description of the listing's purpose appears under the name of the file. The description of Listing 3.1 is *Setting four registers to zero.*
- The lines of the listing are numbered 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.
- The lines that start with a dot '.' (on lines 1, 2 and 3) are called *assembler directives* as they tell 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 preceding 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 mnemonic. 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.

3.4 Running a Program With rvddt

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

Listing 3.2 shows the operation of the four *addi* instructions from Listing 3.1 when it is executed in trace-mode.

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

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

Listing 3.2: `zero4regs.out`
Running a program with the rvddt simulator

```
1  [winans@w510 src]$ ./rvddt -f ../examples/load4regs.bin
2  Loading '../examples/load4regs.bin' to 0x0
3  ddt> t4
4     x0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
5     x8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
6    x16: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
7    x24: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
8     pc: 00000000
9  00000000: 00000e13  addi    x28, x0, 0    # x28 = 0x00000000 = 0x00000000 + 0x00000000
10    x0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
11    x8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
12   x16: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
13   x24: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0
14    pc: 00000004
15 00000004: 00000e93  addi    x29, x0, 0    # x29 = 0x00000000 = 0x00000000 + 0x00000000
16    x0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
17    x8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
18   x16: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
19   x24: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-00000000 00000000 f0f0f0f0 f0f0f0f0
20    pc: 00000008
21 00000008: 00000f13  addi    x30, x0, 0    # x30 = 0x00000000 = 0x00000000 + 0x00000000
22    x0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
23    x8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
24   x16: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
25   x24: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-00000000 00000000 00000000 f0f0f0f0
26    pc: 0000000c
27 0000000c: 00000f93  addi    x31, x0, 0    # x31 = 0x00000000 = 0x00000000 + 0x00000000
28 ddt> r
29    x0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
30    x8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
31   x16: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
32   x24: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-00000000 00000000 00000000 00000000
33    pc: 00000010
34 ddt> x
35 [winans@w510 src]$
```

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

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

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

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

ℓ 4 The values in registers 0, 1, 2, 3, 4, 5, 6 and 7 are printed from left to right in big endian, hexadecimal form. The 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.

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

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

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 mnemonic 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.

... A simulator-generated comment that explains 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: $0 + 0$.

ℓ 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 4

Writing RISC-V Programs

This chapter introduces each of the RV32I instructions by developing programs that demonstrate their usefulness.

Fix Me:
Introduce the ISA register names and aliases in here?

4.1 Use `ebreak` to Stop `rvddt` Execution

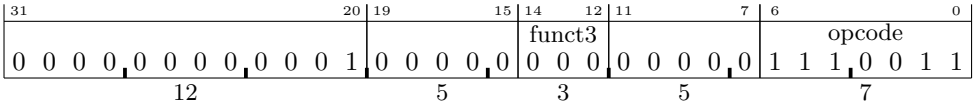
It is a good idea to learn how to stop before learning how to go!

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

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

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

ebreak



I-type

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

Listing 4.1: `ebreak/ebreak.S`
A one-line `ebreak` program.

```
1  .text                # put this into the text section
2  .align  2            # align to a multiple of 4
3  .globl  _start
4
5  _start:
6      ebreak
```

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

Listing 4.2: ebreak/ebreak.out

Instruction!addi
Instruction!nop

ebreak stopps rvddt without advancing pc.

```
$ rvddt -f ebreak.bin
sp initialized to top of memory: 0x0000fff0
Loading 'ebreak.bin' to 0x0
This is rvddt.  Enter ? for help.
ddt> d 0 16
 00000000: 73 00 10 00 a5 a5 a5 a5-a5 a5 a5 a5 a5 a5 a5 a5 *s...............*
ddt> r
   x0 00000000 f0f0f0f0 0000fff0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
   x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
   pc 00000000
ddt> ti 0 1000
00000000: ebreak
ddt> ti
00000000: ebreak
ddt> g 0
00000000: ebreak
ddt> r
   x0 00000000 f0f0f0f0 0000fff0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
   x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
   pc 00000000
ddt> x
```

4.2 Using the addi Instruction

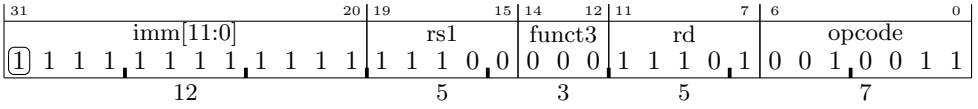
The detailed description of how the addi instruction is executed is that it:

Fix Me:
Define what constant and immediate values are somewhere.

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

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

addi x29, x28, -1



I-type

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

4.2.1 No Operation

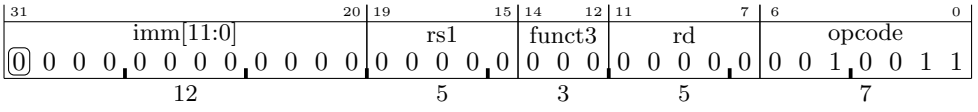
It might seem odd but it is sometimes important to be able to execute an instruction that accomplishes nothing while simply advancing the pc to the next instruction. One reason for this is to fill unused

memory between two instructions in a program.[2] objdump

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

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

addi x0, x0, 0



I-type

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

The RISC-V assembler provides a pseudoinstruction specifically for this purpose that you can use to improve the readability of your code. Note that the `addi` and `nop` instructions in Listing 4.3 are assembled into the exact same binary machine instruction (The `0x00000013` you can see are stored at addresses `0x0` and `0x4`) as seen by looking at the objdump listing in Listing 4.4. In fact, you can see that objdump shows both instructions as a `nop` while Listing 4.5 shows that `rvddt` displays both as `addi x0, x0, 0`.

Listing 4.3: `nop/nop.S`
Demonstrate that an `addi` can be the same as `nop`.

```
1     .text                # put this into the text section
2     .align  2            # align to a multiple of 4
3     .globl  _start
4
5  _start:
6     addi    x0, x0, 0    # these two instructions assemble into the same thing!
7     nop
8
9     ebreak
```

Listing 4.4: `nop/nop.lst`
Using `addi` to perform a `nop`

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

Listing 4.5: `nop/nop.out`
Using `addi` to perform a `nop`

```
1  $ rvddt -f nop.bin
2  sp initialized to top of memory: 0x0000fff0
3  Loading 'nop.bin' to 0x0
4  This is rvddt.  Enter ? for help.
5  ddt> d 0 16
6   00000000: 13 00 00 00 13 00 00 00-73 00 10 00 a5 a5 a5 a5 *........s.......*
7  ddt> r
8     x0 00000000 f0f0f0f0 0000fff0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
```

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

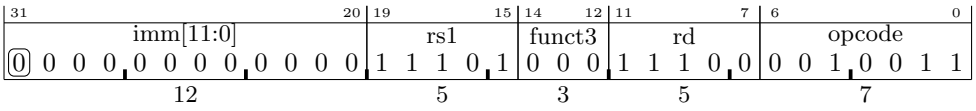
```
   x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
   pc 00000000
ddt> ti 0 1000
00000000: 00000013  addi    x0, x0, 0     # x0 = 0x00000000 = 0x00000000 + 0x00000000
00000004: 00000013  addi    x0, x0, 0     # x0 = 0x00000000 = 0x00000000 + 0x00000000
00000008: ebreak
ddt> r
   x0 00000000 f0f0f0f0 0000fff0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
   x8 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  x16 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
  x24 f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
   pc 00000008
ddt> x
```

Instruction!mv

4.2.2 Copying the Contents of One Register to Another

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

addi t3, t4, 0



I-type

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

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

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

```
    .text               # put this into the text section
    .align  2           # align to a multiple of 4
    .globl  _start

_start:
    addi    t3, t4, 0   # t3 = t4
    mv      t3, t4      # t3 = t4

    ebreak
```

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

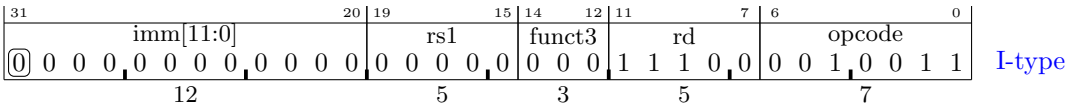
```
mv:     file format elf32-littleriscv
Disassembly of section .text:
00000000 <_start>:
   0:   000e8e13            mv  t3,t4
   4:   000e8e13            mv  t3,t4
   8:   00100073            ebreak
```

4.2.3 Setting a Register to Zero

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

For example, to set t3 to zero:

addi t3, x0, 0



I-type

Listing 4.8: mvzero/mv.S
Using mv (aka addi) to zero-out a register.

```
1     .text                # put this into the text section
2     .align  2            # align to a multiple of 4
3     .globl  _start
4
5  _start:
6     mv      t3, x0       # t3 = 0
7
8     ebreak
```

Listing 4.8 traces the execution of the program in Listing 4.8 showing how t3 is changed from 0xf0f0f0f0 (seen on ℓ16) to 0x00000000 (seen on ℓ26.)

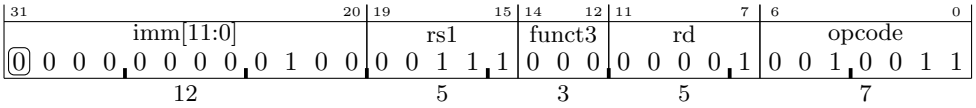
Listing 4.9: mvzero/mv.out
Setting t3 to zero.

```
1  $ rvddt -f mv.bin
2  sp initialized to top of memory: 0x0000fff0
3  Loading 'mv.bin' to 0x0
4  This is rvddt.  Enter ? for help.
5  ddt> a
6  ddt> d 0 16
7   00000000: 13 0e 00 00 73 00 10 00-a5 a5 a5 a5 a5 a5 a5 a5 *....s...........*
8  ddt> t 0 1000
9   zero   x0 00000000  ra   x1 f0f0f0f0  sp   x2 0000fff0  gp   x3 f0f0f0f0
10    tp   x4 f0f0f0f0  t0   x5 f0f0f0f0  t1   x6 f0f0f0f0  t2   x7 f0f0f0f0
11    s0   x8 f0f0f0f0  s1   x9 f0f0f0f0  a0 x10 f0f0f0f0  a1 x11 f0f0f0f0
12    a2 x12 f0f0f0f0  a3 x13 f0f0f0f0  a4 x14 f0f0f0f0  a5 x15 f0f0f0f0
13    a6 x16 f0f0f0f0  a7 x17 f0f0f0f0  s2 x18 f0f0f0f0  s3 x19 f0f0f0f0
14    s4 x20 f0f0f0f0  s5 x21 f0f0f0f0  s6 x22 f0f0f0f0  s7 x23 f0f0f0f0
15    s8 x24 f0f0f0f0  s9 x25 f0f0f0f0 s10 x26 f0f0f0f0 s11 x27 f0f0f0f0
16    t3 x28 f0f0f0f0  t4 x29 f0f0f0f0  t5 x30 f0f0f0f0  t6 x31 f0f0f0f0
17        pc 00000000
18 00000000: 00000e13   addi    t3, zero, 0   # t3 = 0x00000000 = 0x00000000 + 0x00000000
19  zero   x0 00000000  ra   x1 f0f0f0f0  sp   x2 0000fff0  gp   x3 f0f0f0f0
20    tp   x4 f0f0f0f0  t0   x5 f0f0f0f0  t1   x6 f0f0f0f0  t2   x7 f0f0f0f0
21    s0   x8 f0f0f0f0  s1   x9 f0f0f0f0  a0 x10 f0f0f0f0  a1 x11 f0f0f0f0
22    a2 x12 f0f0f0f0  a3 x13 f0f0f0f0  a4 x14 f0f0f0f0  a5 x15 f0f0f0f0
23    a6 x16 f0f0f0f0  a7 x17 f0f0f0f0  s2 x18 f0f0f0f0  s3 x19 f0f0f0f0
24    s4 x20 f0f0f0f0  s5 x21 f0f0f0f0  s6 x22 f0f0f0f0  s7 x23 f0f0f0f0
25    s8 x24 f0f0f0f0  s9 x25 f0f0f0f0 s10 x26 f0f0f0f0 s11 x27 f0f0f0f0
26    t3 x28 00000000  t4 x29 f0f0f0f0  t5 x30 f0f0f0f0  t6 x31 f0f0f0f0
27        pc 00000004
28 00000004: ebreak
29 ddt> x
```

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

4.2.4 Adding a 12-bit Signed Value

addi x1, x7, 4



I-type

```
    addi    t0, zero, 4     # t0 = 4
    addi    t1, t1, 100     # t1 = 104

    addi    t0, zero, 0x123     # t0 = 0x123
    addi    t0, t0, 0xfff       # t0 = 0x122 (subtract 1)

    addi    t0, zero, 0xfff     # t0 = 0xffffffff (-1)  (diagram out the chaining carry)
                                # refer back to the overflow/truncation discussion in binary chapter

addi x0, x0, 0 # no operation (pseudo: nop)
addi rd, rs, 0 # copy reg rs to rd (pseudo: mv rd, rs)
```

Demonstrate various addi instructions.

4.3 todo

Ideas for the order of introducing instructions.

4.4 Other Instructions With Immediate Operands

```
    addi
    andi
    ori
    xori

    slti
    sltiu
    srai
    slli
    srli
```

4.5 Transferring Data Between Registers and Memory

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

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

```
lb
lh
lw
lbu
lhu
```

Copying values from a register to memory:

```
sb
sh
sw
```

➽ Fix Me:
Mention the rvddt UART I/O address for writing to the console here?

4.6 RR operations

```
add
sub
and
or
sra
srl
sll
xor
sltu
slt
```

4.7 Setting registers to large values using lui with addi

```
addi        // useful for values from -2048 to 2047
lui         // useful for loading any multiple of 0x1000

Setting a register to any other value must be done using a combo of insns:

auipc       // Load an address relative the the current PC (see la pseudo)
addi


lui         // Load constant into into bits 31:12  (see li pseudo)
addi        // add a constant to fill in bits 11:0
                if bit 11 is set then need to +1 the lui value to compensate
```

4.8 Labels and Branching

Start to introduce addressing here?

```
    beq
    bne
    blt
    bge
    bltu
    bgeu

    bgt rs, rt, offset      # pseudo for: blt rt, rs, offset    (reverse the operands)
    ble rs, rt, offset      # pseudo for: bge rt, rs, offset    (reverse the operands)
    bgtu rs, rt, offset     # pseudo for: bltu rt, rs, offset   (reverse the operands)
    bleu rs, rt, offset     # pseudo for: bgeu rt, rs, offset   (reverse the operands)

    beqz beqz rs, offset    # pseudo for: beq rs, x0, offset
    bnez rs, offset         # pseudo for: bne rs, x0, offset
    blez rs, offset         # pseudo for: bge x0, rs, offset
    bgez rs, offset         # pseudo for: bge rs, x0, offset
    bltz rs, offset         # pseudo for: blt rs, x0, offset
    bgtz rs, offset         # pseudo for: blt x0, rs, offset
```

4.9 Relocation

```
Absolute:
    %hi(symbol)
    %lo(symbol)

PC-relative:
    %pcrel_hi(symbol)
    %pcrel_lo(label)

Using the auipc & addi pair with label references:
    The %pcrel_lo() uses the label to find the associated %pcrel_hi()
    The label MUST be on a line that used a %pcrel_hi() or get an error.
    This is needed to calculate the proper offset.
    Things like this are legal (though not sure of the value):
    label:  auipc   t1, %pcrel_hi(symbol)
        addi    t2, t1, %pcrel_lo(label)
        addi    t3, t1, %pcrel_lo(label)
        lw      t4, %pcrel_lo(label)(t1)
        sw      t5, %pcrel_lo(label)(t1)

Discuss how relaxation works.
see:  https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md
```

4.10 Jumps

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

```
    jal
    jalr
```

4.11 Pseudo Operations

Fix Me:
Explain why we have pseudo ops. These mappings are lifted from the ISM, Vol 1, V2.2

```
    la rd, symbol
                    auipc rd, symbol[31:12]
                    addi rd, rd, symbol[11:0]

    l{b|h|w|d} rd, symbol
                    auipc rd, symbol[31:12]
                    l{b|h|w|d} rd, symbol[11:0](rd)

    s{b|h|w|d} rd, symbol, rt                   # rt is the temp reg to use for the operation
                    auipc rt, symbol[31:12]
                    s{b|h|w|d} rd, symbol[11:0](rt)


    j offset        jal x0, offset
    jal offset      jal x1, offset
    jr rs           jalr x0, rs, 0
    jalr rs         jalr x1, rs, 0
    ret             jalr x0, x1, 0

    call offset     auipc x6, offset[31:12]
                    jalr x1, x6, offset[11:0]

    tail offset     auipc x6, offset[31:12]     # same as call but no x1
                    jalr x0, x6, offset[11:0]
```

4.12 The Linker and Relaxation

I don't know where this should go just yet.

Fix Me:
Needs research. I'm not sure if/how the linker alone can relax the AUIPC+JALR pair since the assembler could have used a pcrel branch across one of these pairs.

4.13 pic and nopic

pic is *needed* for shared libs. Should discuss it but probably best to leave the topic for a later chapter.

Chapter 5

RV32 Machine Instructions

5.1 Introduction

5.2 Conventions and Terminology

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

5.2.1 XLEN

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

5.2.2 sx(val)

Sign extend *val* to the left.

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

For more on sign-extension see section 2.3.

5.2.3 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.

For more on zero-extension see Figure 2.3.

5.2.4 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 5.1 illustrates converting a 20-bit *val* to a 32-bit fullword.

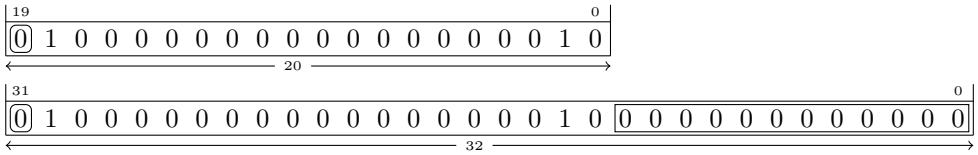


Figure 5.1: Zero-extending an integer to the right from 20 bits to 32 bits.

5.2.5 Sign Extended Left and Zero Extend Right

Some instructions such as the J-type (see section 5.4.2) include immediate operands that are extended in both directions.

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

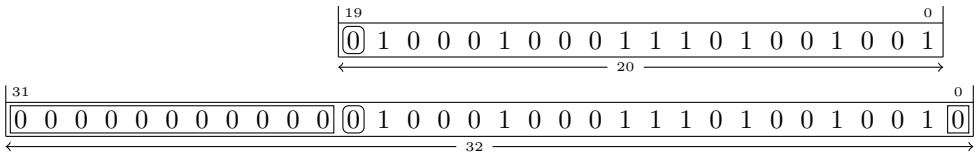


Figure 5.2: Sign-extending a positive 20-bit number 11 bits to the left and one bit to the right.

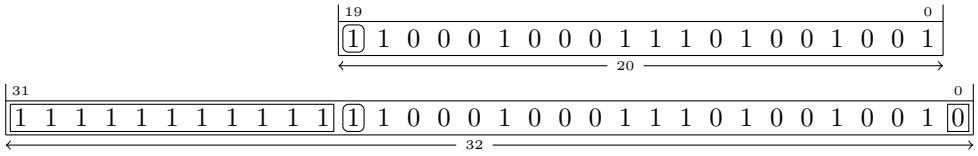


Figure 5.3: Sign-extending a negative 20-bit number 11 bits to the left and one bit to the right.

5.2.6 m8(addr)

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

Given the contents of the memory dump shown in Figure 5.4, `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)` ← `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` ← `m8(addr)`

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

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

Figure 5.4: Sample memory contents.

5.2.7 m16(addr)

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

Given the contents of the memory dump shown in Figure 5.4, `m16(42)` refers to the memory location at address 42_{16} that currently contains $65fc_{16}$. See also section 5.2.6.

5.2.8 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 5.4, `m32(42)` refers to the memory location at address 42_{16} that currently contains $662065fc_{16}$. See also section 5.2.6.

5.2.9 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 5.4, `m64(42)` refers to the memory location at address 42_{16} that currently contains $656c6c69662065fc_{16}$. See also section 5.2.6.

5.2.10 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 5.4, `m128(42)` refers to the memory location at address 42_{16} that currently contains $72206874697720646 56c6c69662065fc_{16}$. See also section 5.2.6.

5.2.11 .+offset

The address of the current instruction plus a numeric offset.

5.2.12 .-offset

The address of the current instruction minus a numeric offset.

5.2.13 pc

The current value of the program counter.

5.2.14 rd

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

5.2.15 rs1

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

5.2.16 rs2

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

5.2.17 imm

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

5.2.18 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.

5.3 Addressing Modes

immediate, register, base-displacement, pc-relative

Fix Me:
Write this section.

5.4 Instruction Encoding Formats

This document concerns itself with the following 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.*

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 5.5 Shows the RISC-V instruction formats.

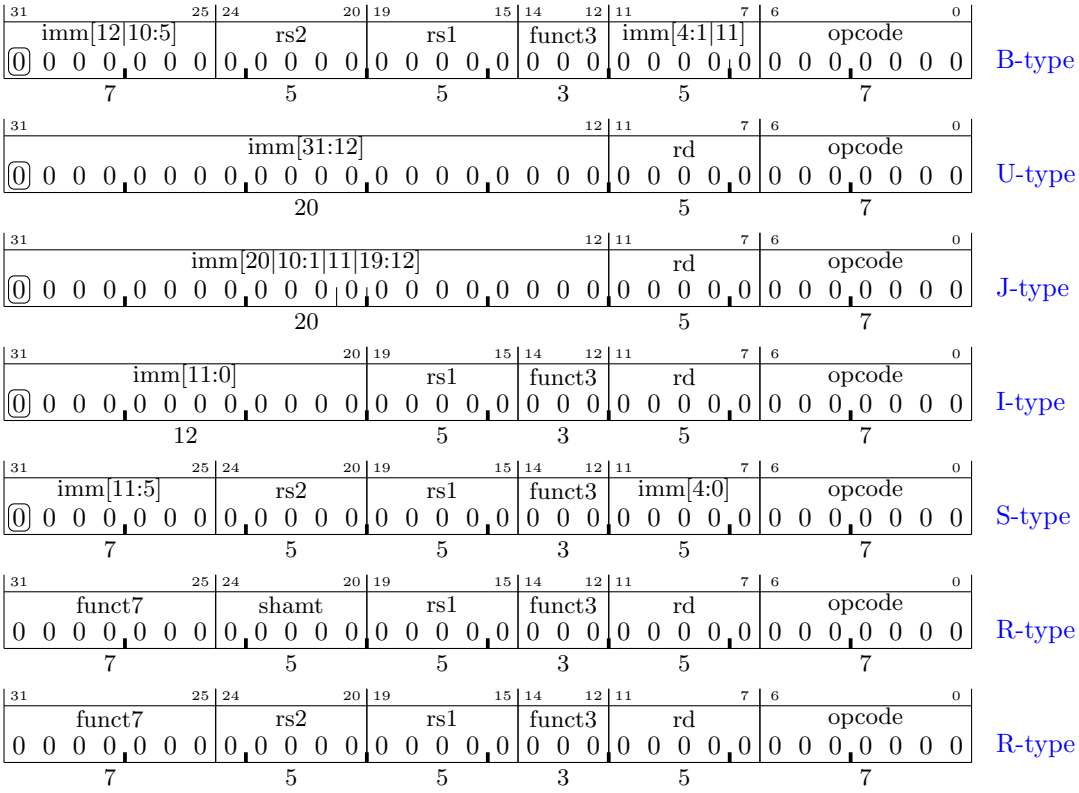
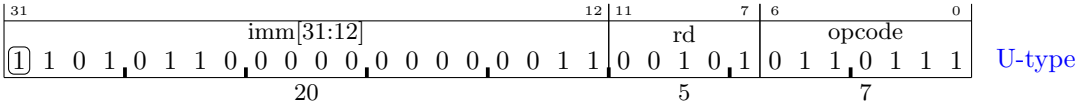


Figure 5.5: RISC-V instruction formats.

5.4.1 U Type

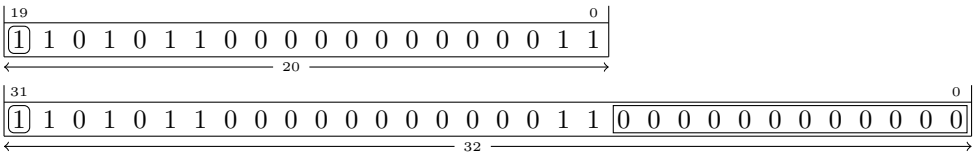
The U-Type format is used for instructions that use a 20-bit immediate operand and a destination register.



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 left[1] and zero-extending the LSBs as discussed in section 5.2.4.

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

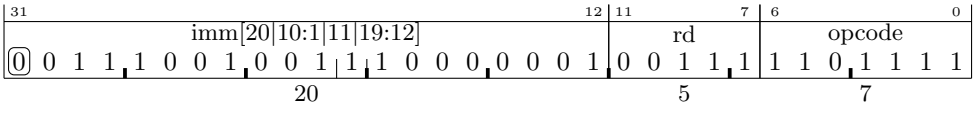


Notice that the 20-bits of the imm field are mapped in the same order and in the same relative position that they appear in the instruction when they are used to create the value of the immediate operand. Shifting the imm value to the left, into the "upper bits" of the immediate value suggests a rationale 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.

5.4.2 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.



J-type

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]

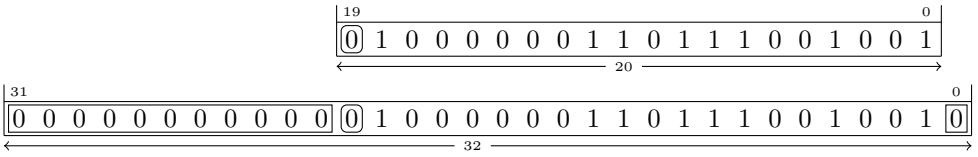
The example above shows that the bit positions in the *imm* field description. We see that the 20 *imm* bits are re-ordered according to: [20|10:1|11|19:12]. This means that the MSB of the *imm* field is to be placed into bit 20 of the immediate integer value ultimately used by the instruction when it 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 section 5.2.5.

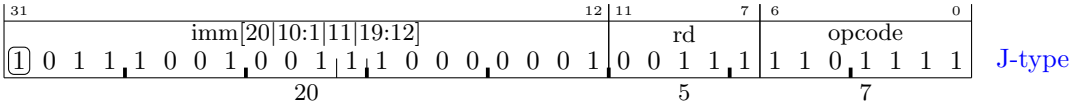
If XLEN=32 then the *imm* value in this example will be converted as shown below.

[1] When XLEN is larger than 32.

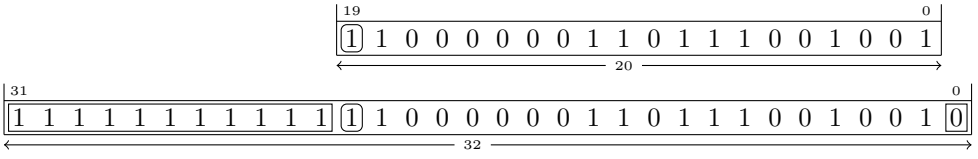
[2] The reason that the J-type bits are reordered like this is because it simplifies the implementation of hardware as discussed in section 5.4.



A J-type example with a negative imm field:

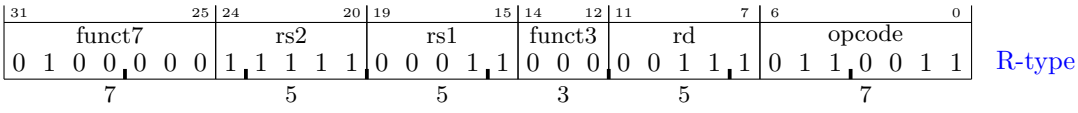


If XLEN=32 then the *imm* field in this example will be converted as shown below.

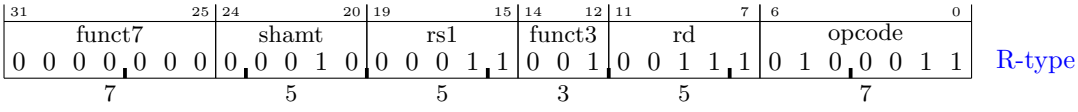


The J-type format is used by the Jump And Link instruction that calculates a target address by adding a signed immediate value to the current program counter. Since no instruction can be placed at an odd address the 20-bit imm value is zero-extended to the right to represent a 21-bit signed offset capable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

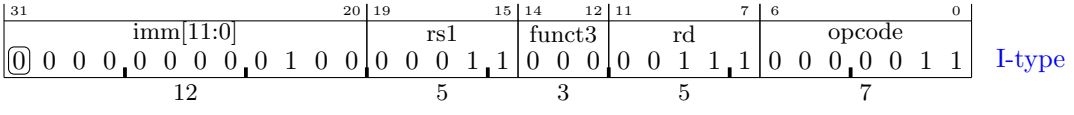
5.4.3 R Type



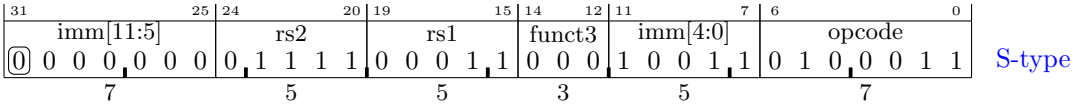
A special case of the R-type used for shift-immediate instructions where the *rs2* field is used as an immediate value named *shamt* representing the number of bit positions to shift:



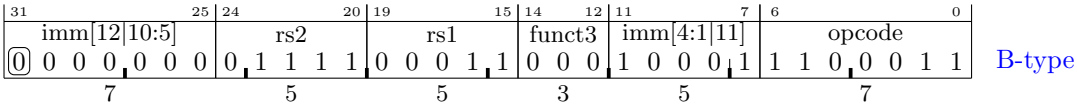
5.4.4 I Type



5.4.5 S Type



5.4.6 B Type



5.4.7 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 functions if they alter their contents. The idea being that accessing memory is time-consuming and 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.

Fix Me:
Need to add a section that 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.

RV32I

Reg	Alias	Description	Saved
x0	zero	Hard-wired zero	
x1	ra	Return address	
x2	sp	Stack pointer	yes
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary/alternate link register	
x6	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	

5.5 memory

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

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

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

5.6 RV32I Base Instruction Set

RV32I refers to the basic 32-bit integer instructions.

➽ Fix Me:
Migrate all te details into the programming chapter and reduce this section to the obligatory reference chapter.

5.6.1 LUI rd, imm

Instruction!LUI
Instruction!AUIPC

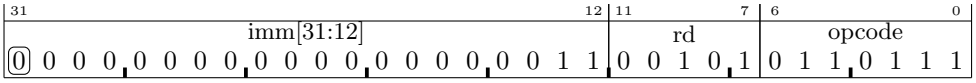
Load Upper Immediate.

`rd ← 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:

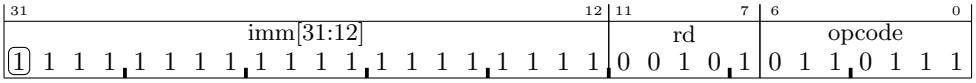
LUI t0, 3



U-type

```
00010074: 000032b7  lui     x5, 0x3         // x5 = 0x3000
 reg  0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 00003000 f0f0f0f0 f0f0f0f0
 reg  8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
 reg 16: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
 reg 24: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
     pc: 00010078
```

LUI t0, 0xfffff



U-type

```
00010078: fffff2b7  lui     x5, 0xfffff         // x5 = 0xfffff000
 reg  0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 fffff000 f0f0f0f0 f0f0f0f0
 reg  8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
 reg 16: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
 reg 24: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
     pc: 0001007c
```

5.6.2 AUIPC rd, imm

Add Upper Immediate to PC.

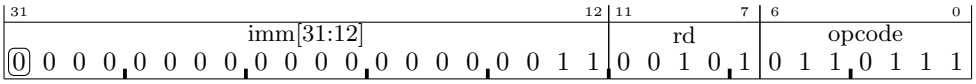
`rd ← pc + zr(imm)`

Create a signed 32-bit value by zero-extending imm[31:12] to the right (see section 5.2.4) 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

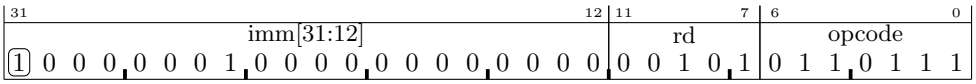
Instruction!JAL



U-type

```
0001007c: 00003297  auipc   x5, 0x3          // x5 = 0x1307c = 0x1007c + 0x3000
 reg  0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 0001307c f0f0f0f0 f0f0f0f0
 reg  8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
 reg 16: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
 reg 24: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
     pc: 00010080
```

AUIPC t0, 0x81000



U-type

```
00010080: 81000297  auipc   x5, 0x81000          // x5 = 0x81010080 = 0x10080 + 0x81000000
 reg  0: 00000000 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 81010080 f0f0f0f0 f0f0f0f0
 reg  8: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
 reg 16: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
 reg 24: f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0-f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
     pc: 00010084
```

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]

5.6.3 JAL rd, imm

Jump and link.

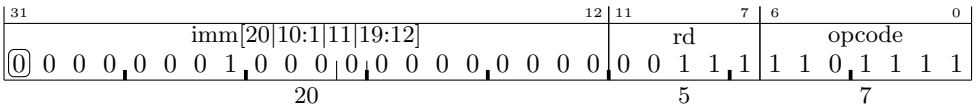
`rd` ← `pc + 4`
`pc` ← `pc + sx(imm<<1)`

This instruction saves the address of the next instruction that would otherwise execute (located at `pc`+4) 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:

JAL x7, .+16



J-type

imm demultiplexed value = $00000000000000001000_2 \ll 1 = 16_{10}$

Instruction!JALR

State of registers before execution:

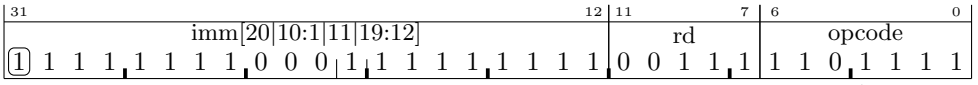
pc = 0x11114444

State of registers after execution:

pc = 0x11114454 x7 = 0x11114448

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

JAL x7, .-16



J-type

imm demultiplexed value = $1111111111111111000_2 \ll 1 = -16_{10}$

State of registers before execution:

pc = 0x11114444

State of registers after execution:

pc = 0x11114434 x7 = 0x11114448

5.6.4 JALR rd, rs1, imm

Jump and link register.

```
rd ← pc + 4
pc ← (rs1 + sx(imm)) & ~1
```

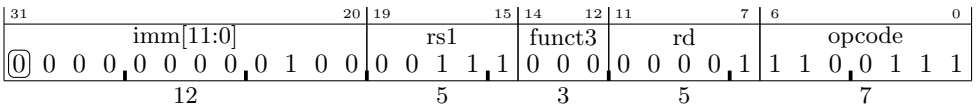
This instruction saves the address of the next instruction that would otherwise execute (located at `pc`+4) into `rd` and then adds the immediate value to the `rs1` register and stores the sum into the `pc` register causing an unconditional branch to take place.

Note that the branch target address is calculated by sign-extending the imm[11:0] bits from the instruction, adding it to the `rs1` register and *then* the LSB of the sum is to zero and the result is stored into 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



I-type

Before:

pc = 0x11114444
x7 = 0x44444444

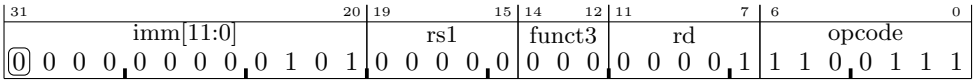
Instruction!BEQ
Instruction!BNE

After

pc = 0x5555888c
x1 = 0x11114448

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

JALR x1, x0, 5



I-type

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 = 0x11114444

After

pc = 0x00000004
x1 = 0x11114448

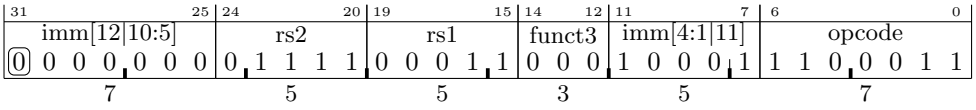
5.6.5 BEQ rs1, rs2, imm

Branch if equal.

`pc ← (rs1 == rs2) ? pc+sx(imm[12:1]<<1) : pc+4`

Encoding:

BEQ x3, x15, 2064



B-type

imm[12:1] = $010000001000_2 = 1032_{10}$
imm = 2064_{10}
funct3 = 000_2
rs1 = x3
rs2 = x15

5.6.6 BNE rs1, rs2, imm

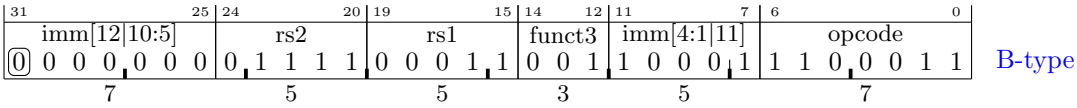
Branch if Not Equal.

`pc ← (rs1 != rs2) ? pc+sx(imm[12:1]<<1) : pc+4`

Encoding:

BNE x3, x15, 2064

Instruction!BLT
Instruction!BGE



B-type

imm[12:1] = $010000001000_2 = 1032_{10}$
imm = 2064_{10}
funct3 = 001_2
rs1 = x3
rs2 = x15

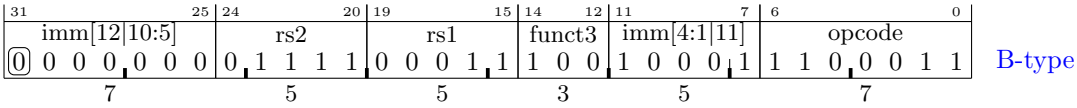
5.6.7 BLT rs1, rs2, imm

Branch if Less Than.

`pc ← (rs1 < rs2) ? pc+sx(imm[12:1]<<1) : pc+4`

Encoding:

BLT x3, x15, 2064



B-type

imm[12:1] = $010000001000_2 = 1032_{10}$
imm = 2064_{10}
funct3 = 100_2
rs1 = x3
rs2 = x15

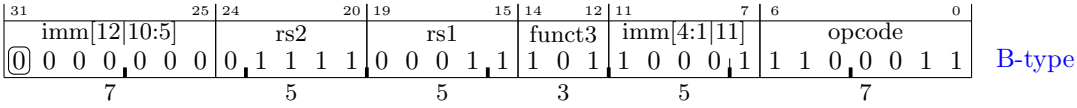
5.6.8 BGE rs1, rs2, imm

Branch if Greater or Equal.

`pc ← (rs1 >= rs2) ? pc+sx(imm[12:1]<<1) : pc+4`

Encoding:

BGE x3, x15, 2064



B-type

imm[12:1] = $010000001000_2 = 1032_{10}$
imm = 2064_{10}
funct3 = 101_2
rs1 = x3
rs2 = x15

5.6.9 BLTU rs1, rs2, imm

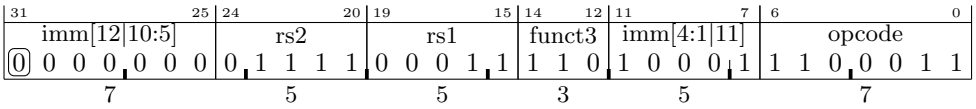
Instruction!BLTU
Instruction!BGEU
Instruction!LB

Branch if Less Than Unsigned.

`pc ← (rs1 < rs2) ? pc+sx(imm[12:1]<<1) : pc+4`

Encoding:

BLTU x3, x15, 2064



B-type

$\text{imm}[12:1] = 010000001000_2 = 1032_{10}$
$\text{imm} = 2064_{10}$
$\text{funct3} = 110_2$
rs1 = x3
rs2 = x15

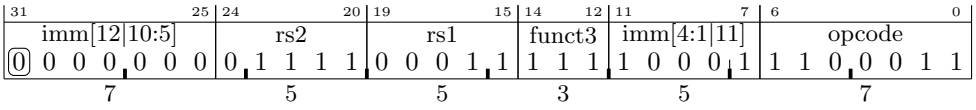
5.6.10 BGEU rs1, rs2, imm

Branch if Greater or Equal Unsigned.

`pc ← (rs1 >= rs2) ? pc+sx(imm[12:1]<<1) : pc+4`

Encoding:

BGEU x3, x15, 2064



B-type

Fix Me:
use symbols in branch examples

$\text{imm}[12:1] = 010000001000_2 = 1032_{10}$
$\text{imm} = 2064_{10}$
$\text{funct3} = 111_2$
rs1 = x3
rs2 = x15

5.6.11 LB rd, imm(rs1)

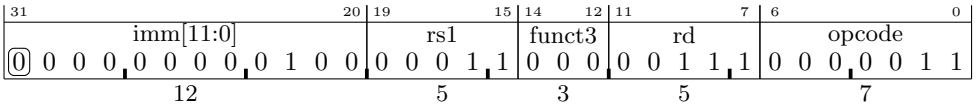
Load byte.

`rd ← sx(m8(rs1+sx(imm)))`
`pc ← pc+4`

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)



I-type

Instruction!LH
Instruction!LW
Instruction!LBU

5.6.12 LH rd, imm(rs1)

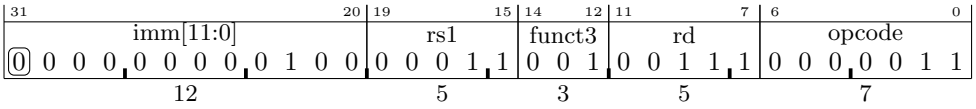
Load halfword.

`rd` ← `sx(m16(rs1+sx(imm)))`
`pc` ← `pc+4`

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)



I-type

5.6.13 LW rd, imm(rs1)

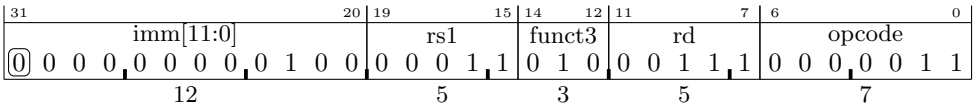
Load word.

`rd` ← `sx(m32(rs1+sx(imm)))`
`pc` ← `pc+4`

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

Encoding:

LW x7, 4(x3)



I-type

5.6.14 LBU rd, imm(rs1)

Load byte unsigned.

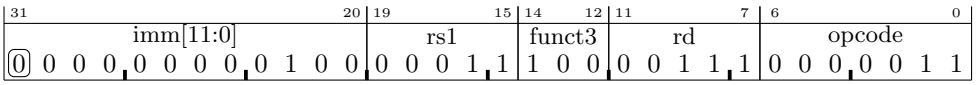
`rd` ← `zx(m8(rs1+sx(imm)))`
`pc` ← `pc+4`

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

Encoding:

Instruction!LHU
Instruction!SB
Instruction!SH

LBU x7, 4(x3)



I-type

5.6.15 LHU rd, imm(rs1)

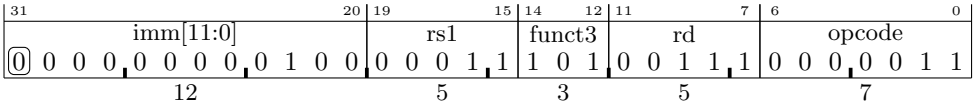
Load halfword unsigned.

`rd` ← `zx(m16(rs1+sx(imm)))`
`pc` ← `pc+4`

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)



I-type

5.6.16 SB rs2, imm(rs1)

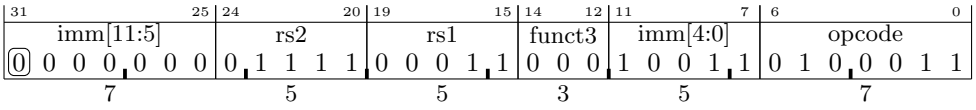
Store Byte.

`m8(rs1+sx(imm))` ← `rs2[7:0]`
`pc` ← `pc+4`

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

Encoding:

SB x3, 19(x15)



S-type

5.6.17 SH rs2, imm(rs1)

Store Halfword.

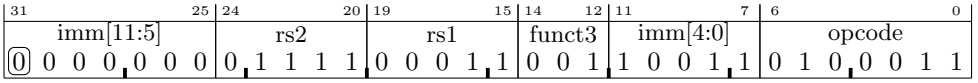
`m16(rs1+sx(imm))` ← `rs2[15:0]`
`pc` ← `pc+4`

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

Encoding:

Instruction!SW
Instruction!ADDI
Instruction!SLTI

SH x3, 19(x15)



S-type

5.6.18 SW rs2, imm(rs1)

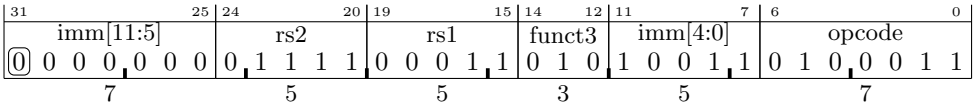
Store Word

`m16(rs1+sx(imm))` ← `rs2[31:0]`
`pc` ← `pc+4`

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

Encoding:

SW x3, 19(x15)



S-type

Show pos & neg imm examples.

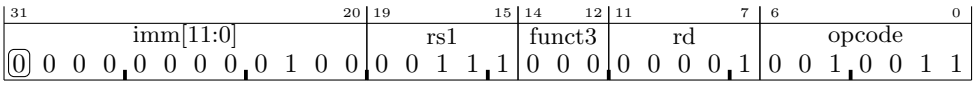
5.6.19 ADDI rd, rs1, imm

Add Immediate

`rd` ← `rs1+sx(imm)`
`pc` ← `pc+4`

Encoding:

ADDI x1, x7, 4



I-type

Before:

x7 = 0x11111111

After:

x1 = 0x11111115

5.6.20 SLTI rd, rs1, imm

Set LessThan Immediate

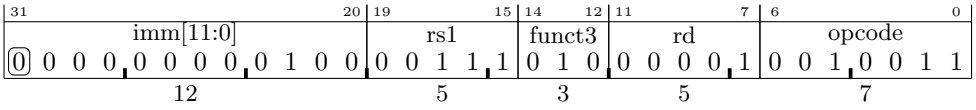
Instruction!SLTIU
Instruction!XORI

1826 `rd ← (rs1 < sx(imm)) ? 1 : 0`
1827 `pc ← pc+4`

1828 If the sign-extended immediate value is less than the value in the `rs1` register then the value 1 is
1829 stored in the `rd` register. Otherwise the value 0 is stored in the `rd` register.

1830 Encoding:

1831 SLTI x1, x7, 4



I-type

1832

1833 Before:

1834 x7 = 0x11111111

1835 After:

1836 x1 = 0x00000000

1837 5.6.21 SLTIU rd, rs1, imm

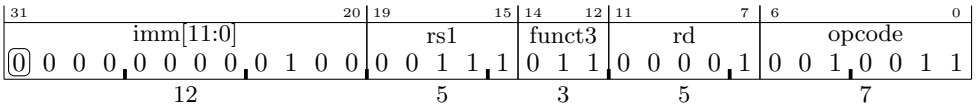
1838 Set LessThan Immediate Unsigned

1839 `rd ← (rs1 < sx(imm)) ? 1 : 0`
1840 `pc ← pc+4`

1841 If the sign-extended immediate value is less than the value in the `rs1` register then the value 1 is
1842 stored in the `rd` register. Otherwise the value 0 is stored in the `rd` register. Both the immediate and
1843 `rs1` register values are treated as unsigned numbers for the purposes of the comparison.[3]

1844 Encoding:

1845 SLTIU x1, x7, 4



I-type

1846

1847 Before:

1848 x7 = 0x81111111

1849 After:

1850 x1 = 0x00000001

1851 5.6.22 XORI rd, rs1, imm

1852 Exclusive Or Immediate

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

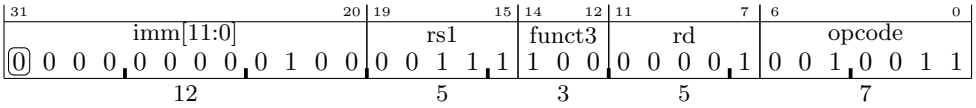
Instruction!ORI
Instruction!ANDI

```
rd ← rs1 ^ sx(imm)
pc ← pc+4
```

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



I-type

Before:

x7 = 0x81111111

After:

x1 = 0x81111115

5.6.23 ORI rd, rs1, imm

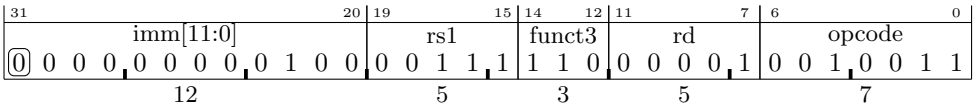
Or Immediate

```
rd ← rs1 | sx(imm)
pc ← pc+4
```

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



I-type

Before:

x7 = 0x81111111

After:

x1 = 0x81111115

5.6.24 ANDI rd, rs1, imm

And Immediate

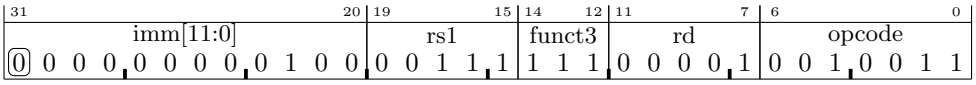
```
rd ← rs1 & sx(imm)
pc ← pc+4
```

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

Instruction!SLLI
Instruction!SRLI

Encoding:

ANDI x1, x7, 4



I-type

Before:

x7 = 0x81111111

After:

x1 = 0x81111115

5.6.25 SLLI rd, rs1, shamt

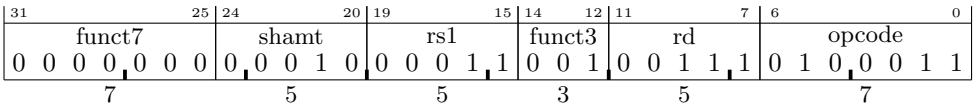
Shift Left Logical Immediate

```
rd ← rs1 << shamt
pc ← pc+4
```

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



R-type

x3 = 0x81111111

After:

x7 = 0x04444444

5.6.26 SRLI rd, rs1, shamt

Shift Right Logical Immediate

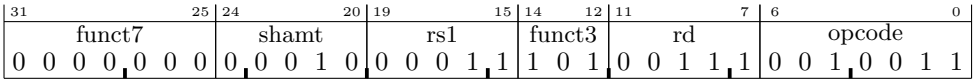
```
rd ← rs1 >> shamt
pc ← pc+4
```

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:

Instruction!SRAI
Instruction!ADD

SRLI x7, x3, 2



R-type

x3 = 0x81111111

After:

x7 = 0x20444444

5.6.27 SRAI rd, rs1, shamt

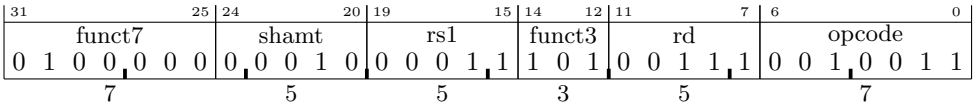
Shift Right Arithmetic Immediate

```
rd ← rs1 >> shamt
pc ← pc+4
```

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



R-type

x3 = 0x81111111

After:

x7 = 0xe0444444

5.6.28 ADD rd, rs1, rs2

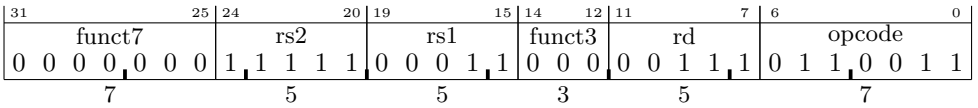
Add

```
rd ← rs1 + rs2
pc ← pc+4
```

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



R-type

x3 = 0x81111111 x31 = 0x22222222

Instruction!SUB
Instruction!SLL

After:

x7 = 0xa3333333

5.6.29 SUB rd, rs1, rs2

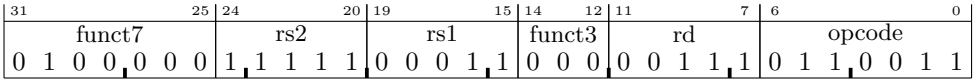
Subtract

`rd` ← `rs1 - rs2`
`pc` ← `pc+4`

SUB performs subtraction. Underflows are ignored and the low 32 bits of the result are written to rd. [1, p. 15]

Encoding:

SUB x7, x3, x31



R-type

x3 = 0x83333333 x31 = 0x01111111

After:

x7 = 0x82222222

5.6.30 SLL rd, rs1, rs2

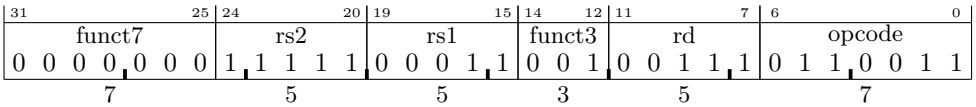
Shift Left Logical

`rd` ← `rs1 << rs2`
`pc` ← `pc+4`

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



R-type

x3 = 0x83333333
x31 = 0x00000002

After:

x7 = 0x0ccccccc

5.6.31 SLT rd, rs1, rs2

Instruction!SLT
Instruction!SLTU
Instruction!XOR

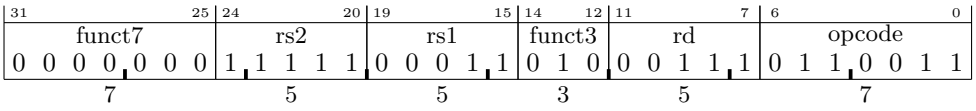
Set Less Than

`rd` ← `(rs1 < rs2) ? 1 : 0`
`pc` ← `pc+4`

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

Encoding:

SLT x7, x3, x31



R-type

x3 = 0x83333333
x31 = 0x00000002

After:

x7 = 0x00000001

5.6.32 SLTU rd, rs1, rs2

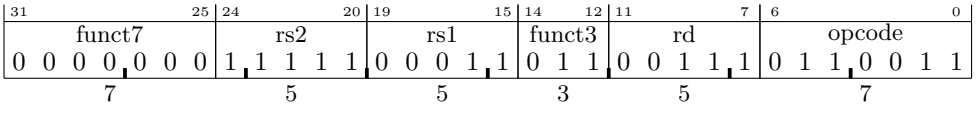
Set Less Than Unsigned

`rd` ← `(rs1 < rs2) ? 1 : 0`
`pc` ← `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



R-type

x3 = 0x83333333
x31 = 0x00000002

After:

x7 = 0x00000000

5.6.33 XOR rd, rs1, rs2

Exclusive Or

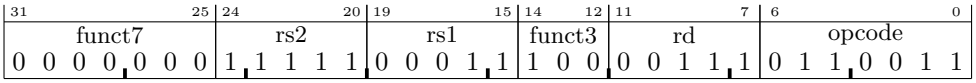
Instruction!SRL
Instruction!SRA

`rd` ← `rs1 ^ rs2`
`pc` ← `pc+4`

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

Encoding:

XOR x7, x3, x31



R-type

x3 = 0x83333333
x31 = 0x1888ffff

After:

x7 = 0x9bbbcccc

5.6.34 SRL rd, rs1, rs2

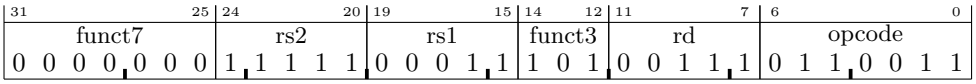
Shift Right Logical

`rd` ← `rs1 >> rs2`
`pc` ← `pc+4`

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



R-type

x3 = 0x83333333
x31 = 0x00000010

After:

x7 = 0x00008333

5.6.35 SRA rd, rs1, rs2

Shift Right Arithmetic

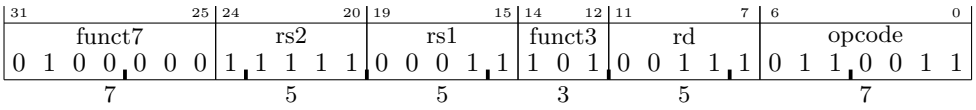
`rd` ← `rs1 >> rs2`
`pc` ← `pc+4`

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]

Instruction!OR
Instruction!AND

Encoding:

SRA x7, x3, x31



R-type

x3 = 0x83333333
x31 = 0x00000010

After:

x7 = 0xffff8333

5.6.36 OR rd, rs1, rs2

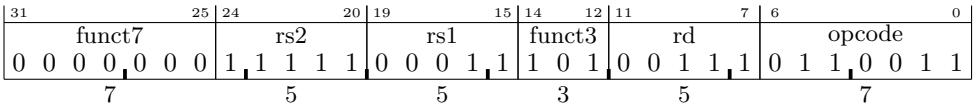
Or

`rd` ← `rs1 | rs2`
`pc` ← `pc+4`

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



R-type

x3 = 0x83333333
x31 = 0x00000440

After:

x7 = 0x83333773

5.6.37 AND rd, rs1, rs2

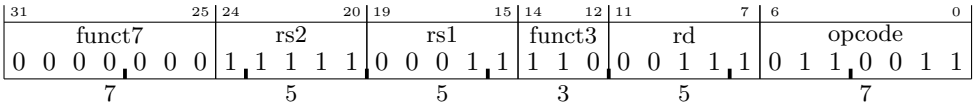
And

`rd` ← `rs1 & rs2`
`pc` ← `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



R-type

Instruction!FENCE
Instruction!FENCE.I

x3 = 0x83333333
x31 = 0x00000fe2

After:

x7 = 0x00000322

5.6.38 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]

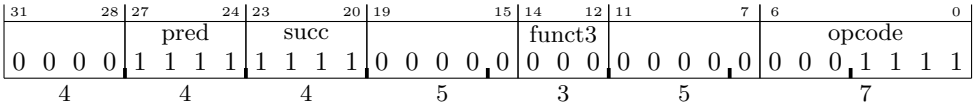
Fix Me:
Which of the i, o, r and w goes into each bit? See what gas does.

Operation:

`pc` ← `pc+4`

Encoding:

FENCE iorw, iorw



FENCE

5.6.39 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]

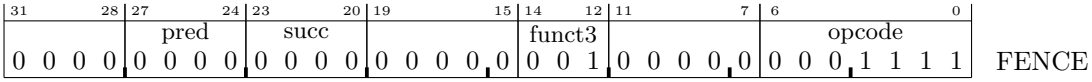
Operation:

`pc` ← `pc+4`

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

Encoding:

FENCE.I

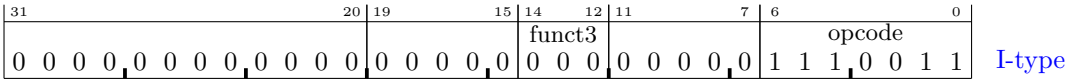


FENCE

5.6.40 ECALL

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

ECALL

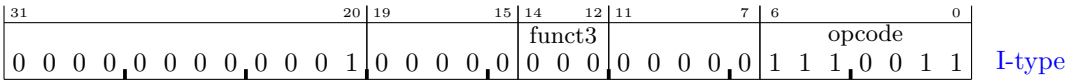


I-type

5.6.41 EBREAK

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

EBREAK

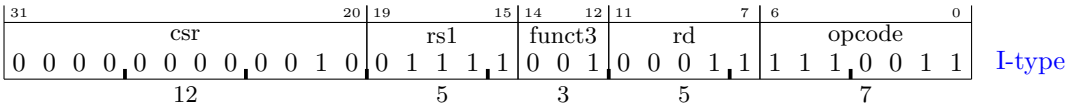


I-type

5.6.42 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]

CSRRW x3, 2, x15



I-type

5.6.43 CSRRS rd, csr, rs1

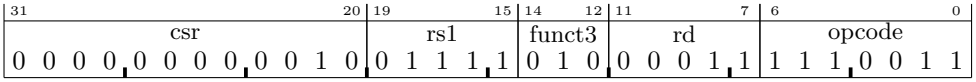
The CSRRS (Atomic Read and Set Bits in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in 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]

Instruction!CSRRC
Instruction!CSRRWI
Instruction!CSRRSI

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]

CSRRS x3, 2, x15



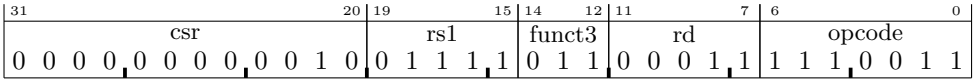
I-type

5.6.44 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]

CSRRC x3, 2, x15

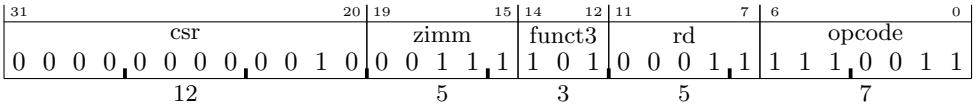


I-type

5.6.45 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.

CSRRWI x3, 2, 7



I-type

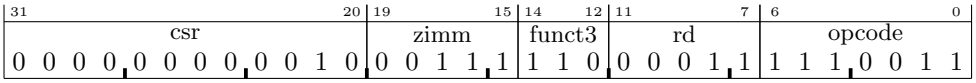
5.6.46 CSRRSI rd, csr, rs1

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]

Instruction!CSRRCI
RV32M
Instruction!MUL
Instruction!MULH

CSRRSI x3, 2, 7

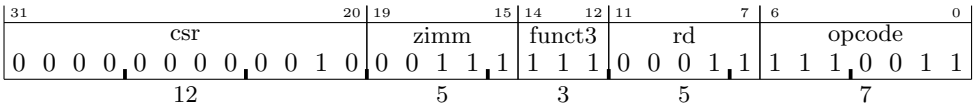


5.6.47 CSRRCI rd, csr, rs1

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]

CSRRCI x3, 2, 7



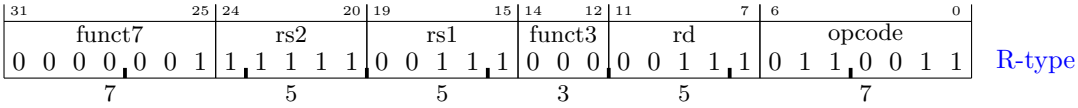
5.7 RV32M Standard Extension

32-bit integer multiply and divide instructions.

5.7.1 MUL rd, rs1, rs2

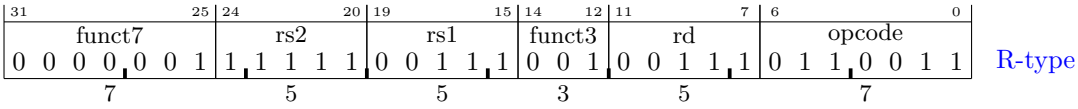
Multiply `rs1` by `rs2` and store the least significant 32-bits of the result in `rd`.

MUL x7, x3, x31



5.7.2 MULH rd, rs1, rs2

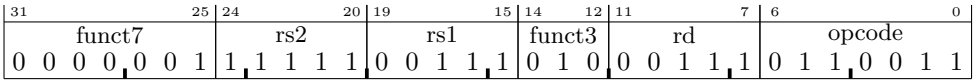
MULH x7, x3, x31



Instruction!MULHS
Instruction!MULHU
Instruction!DIV
Instruction!DIVU
Instruction!REM
Instruction!REMU

5.7.3 MULHS rd, rs1, rs2

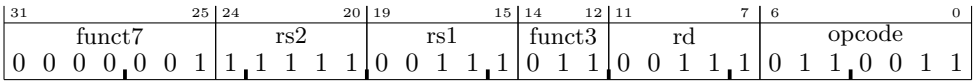
MULHS x7, x3, x31



R-type

5.7.4 MULHU rd, rs1, rs2

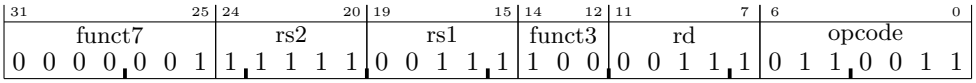
MULHU x7, x3, x31



R-type

5.7.5 DIV rd, rs1, rs2

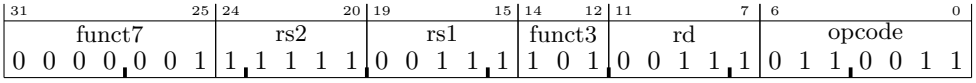
DIV x7, x3, x31



R-type

5.7.6 DIVU rd, rs1, rs2

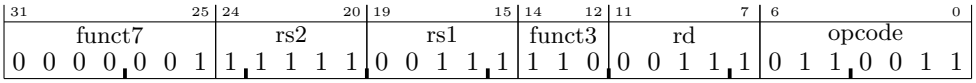
DIVU x7, x3, x31



R-type

5.7.7 REM rd, rs1, rs2

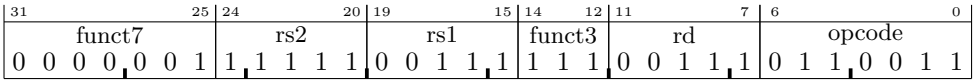
REM x7, x3, x31



R-type

5.7.8 REMU rd, rs1, rs2

REMU x7, x3, x31



R-type

5.8 RV32A Standard Extension

RV32A
RV32F
RV32D

32-bit atomic operations.

5.9 RV32F Standard Extension

32-bit IEEE floating point instructions.

5.10 RV32D Standard Extension

64-bit IEEE floating point instructions.

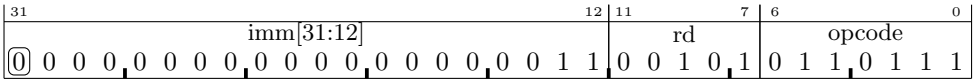
Appendix A

Instruction Set Summary

lui rd, imm — **Load Upper Immediate**

`lui t0, 3`

`rd ← pc + sx(imm<<1)`
`pc ← pc + 4`

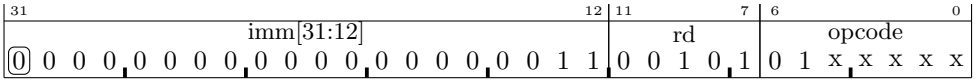


U-type

auipc rd, imm — **Add Upper Immediate PC**

`auipc t0, 3`

`rd ← pc + zr(imm)`
`pc ← pc + 4`

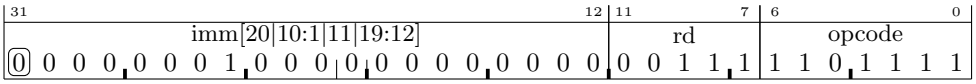


U-type

jal rd, imm — **Jump And Link**

`jal x7, .+16`

`rd ← pc + 4`
`pc ← pc + sx(imm<<1)`

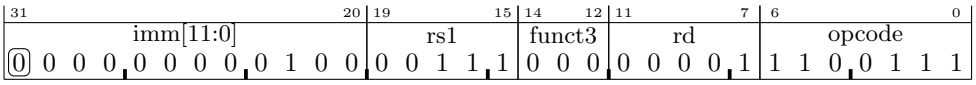


J-type

jalr rd, rs1, imm — **Jump And Link Register**

`jalr x1, x7, 4`

`rd ← pc + 4`
`pc ← (rs1 + sx(imm)) & ~1`

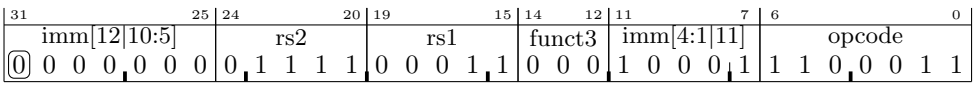


I-type

beq rs1, rs2, imm — **Branch Equal**

`beq x3, x15, 2064`

`pc ← (rs1==rs2) ? pc+sx(imm[12:1]<<1) : pc+4`



B-type

bne rs1, rs2, imm — **Branch Not Equal**

`bne x3, x15, 2064`

`pc ← (rs1!=rs2) ? pc+sx(imm[12:1]<<1) : pc+4`

Fix Me:
Once the RV32I section is re-factored, it may end up turning into this.

Fix Me:
These branches (and likely other insns) are not encoded properly!

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

blt rs1, rs2, imm **Branch Less Than** `pc ← (rs1<rs2) ? pc+sx(imm[12:1]<<1) : pc+4`

`blt x3, x15, 2064`

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

bge rs1, rs2, imm **Branch Greater or Equal** `pc ← (rs1>=rs2) ? pc+sx(imm[12:1]<<1) : pc+4`

`bge x3, x15, 2064`

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

bltu rs1, rs2, imm **Branch Less Than Unsigned** `pc ← (rs1<rs2) ? pc+sx(imm[12:1]<<1) : pc+4`

`bltu x3, x15, 2064`

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

bgeu rs1, rs2, imm **Branch Greater or Equal Unsigned** `pc ← (rs1>=rs2) ? pc+sx(imm[12:1]<<1) : pc+4`

`bgeu x3, x15, 2064`

31 25	24 20	19 15	14 12	11 7	6 0	
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	
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	

lb rd, imm(rs1) **Load Byte** `rd ← sx(m8(rs1+sx(imm)))`

`lb x7, 4(x3)` `pc ← pc+4`

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

lh rd, imm(rs1) **Load Halfword** `rd ← sx(m16(rs1+sx(imm)))`

`lh x7, 4(x3)` `pc ← pc+4`

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

lw rd, imm(rs1) **Load Word** `rd ← sx(m32(rs1+sx(imm)))`

`lw x7, 4(x3)` `pc ← pc+4`

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

lbu rd, imm(rs1) **Load Byte Unsigned** `rd ← zx(m8(rs1+sx(imm)))`

`lbu x7, 4(x3)` `pc ← pc+4`

31	20	19	15	14	12	11	7	6	0
imm[11:0]		rs1		funct3		rd		opcode	
0 0 0 0 0 0 0 0 0 1 0 0		0 0 0 1 1		1 0 0		0 0 1 1 1		0 0 0 0 0 1 1	
12		5		3		5		7	

I-type

lhu rd, imm(rs1) — Load Halfword Unsigned

`rd ← zx(m16(rs1+sx(imm)))`
`pc ← pc+4`

`lhu x7, 4(x3)`

31	20	19	15	14	12	11	7	6	0
imm[11:0]		rs1		funct3		rd		opcode	
0 0 0 0 0 0 0 0 0 1 0 0		0 0 0 1 1		1 0 1		0 0 1 1 1		0 0 0 0 0 1 1	
12		5		3		5		7	

I-type

sb rs2, imm(rs1) — Store Byte

`m8(rs1+sx(imm)) ← rs2[7:0]`
`pc ← pc+4`

`sb x3, 19(x15)`

31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode	
0 0 0 0 0 0 0		0 1 1 1 1		0 0 0 1 1		0 0 0		1 0 0 1 1		0 1 0 0 0 1 1	
7		5		5		3		5		7	

S-type

sh rs2, imm(rs1) — Store Halfword

`m16(rs1+sx(imm)) ← rs2[15:0]`
`pc ← pc+4`

`sh x3, 19(x15)`

31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode	
0 0 0 0 0 0 0		0 1 1 1 1		0 0 0 1 1		0 0 1		1 0 0 1 1		0 1 0 0 0 1 1	
7		5		5		3		5		7	

S-type

sw rs2, imm(rs1) — Store Word

`m16(rs1+sx(imm)) ← rs2[31:0]`
`pc ← pc+4`

`sw x3, 19(x15)`

31	25	24	20	19	15	14	12	11	7	6	0
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode	
0 0 0 0 0 0 0		0 1 1 1 1		0 0 0 1 1		0 1 0		1 0 0 1 1		0 1 0 0 0 1 1	
7		5		5		3		5		7	

S-type

addi rd, rs1, imm — Add Immediate

`rd ← rs1+sx(imm)`
`pc ← pc+4`

`addi x1, x7, 4`

31	20	19	15	14	12	11	7	6	0
imm[11:0]		rs1		funct3		rd		opcode	
0 0 0 0 0 0 0 0 0 1 0 0		0 0 1 1 1		0 0 0		0 0 0 0 1		0 0 1 0 0 1 1	
12		5		3		5		7	

I-type

slti rd, rs1, imm — Set Less Than Immediate

`rd ← (rs1 < sx(imm)) ? 1 : 0`
`pc ← pc+4`

`slti x1, x7, 4`

31	20	19	15	14	12	11	7	6	0
imm[11:0]		rs1		funct3		rd		opcode	
0 0 0 0 0 0 0 0 0 1 0 0		0 0 1 1 1		0 1 0		0 0 0 0 1		0 0 1 0 0 1 1	
12		5		3		5		7	

I-type

sltiu rd, rs1, imm — Set Less Than Immediate Unsigned

`rd ← (rs1 < sx(imm)) ? 1 : 0`
`pc ← pc+4`

`sltiu x1, x7, 4`

31	20	19	15	14	12	11	7	6	0
imm[11:0]		rs1		funct3		rd		opcode	
0 0 0 0 0 0 0 0 0 1 0 0		0 0 1 1 1		0 1 1		0 0 0 0 1		0 0 1 0 0 1 1	
12		5		3		5		7	

I-type

xori rd, rs1, imm — Exclusive Or Immediate

`rd ← rs1 ^ sx(imm)`
`pc ← pc+4`

`xori x1, x7, 4`

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

ori rd, rs1, imm **Or Immediate**

`rd ← rs1 | sx(imm)`
`pc ← pc+4`

`ori x1, x7, 4`

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

andi rd, rs1, imm **And Immediate**

`rd ← rs1 & sx(imm)`
`pc ← pc+4`

`andi x1, x7, 4`

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

slli rd, rs1, shamt **Shift Left Logical Immediate**

`rd ← rs1 << shamt`
`pc ← pc+4`

`slli x7, x3, 2`

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

srli rd, rs1, shamt **Shift Right Logical Immediate**

`rd ← rs1 >> shamt`
`pc ← pc+4`

`srli x7, x3, 2`

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

srai rd, rs1, shamt **Shift Right Arithmetic Immediate**

`rd ← rs1 >> shamt`
`pc ← pc+4`

`srai x7, x3, 2`

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

add rd, rs1, rs2 **Add**

`rd ← rs1 + rs2`
`pc ← pc+4`

`add x7, x3, x31`

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

sub rd, rs1, rs2 **Subtract**

`rd ← rs1 - rs2`
`pc ← pc+4`

`SUB x7, x3, x31`

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

sll rd, rs1, rs2 **Shift Left Logical**

`rd ← rs1 << rs2`
`pc ← pc+4`

`sll x7, x3, x31`

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	
0 0 0 0 0 0 0	1 1 1 1 1	0 0 0 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	

slt rd, rs1, rs2 **Set Less Than** `rd ← rs1 < rs2) ? 1 : 0`

`slt x7, x3, x31` `pc ← pc+4`

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

sltu rd, rs1, rs2 **Set Less Than Unsigned** `rd ← (rs1 < rs2) ? 1 : 0`

`sltu x7, x3, x31` `pc ← pc+4`

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

xor rd, rs1, rs2 **Exclusive Or** `rd ← rs1 ^ rs2`

`xor x7, x3, x31` `pc ← pc+4`

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

srl rd, rs1, rs2 **Shift Right Logical** `rd ← rs1 >> rs2`

`srl x7, x3, x31` `pc ← pc+4`

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

sra rd, rs1, rs2 **Shift Right Arithmetic** `rd ← rs1 >> rs2`

`sra x7, x3, x31` `pc ← pc+4`

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

or rd, rs1, rs2 **Or** `rd ← rs1 | rs2`

`or x7, x3, x31` `pc ← pc+4`

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

and rd, rs1, rs2 **And** `rd ← rs1 & rs2`

`and x7, x3, x31` `pc ← pc+4`

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	
0 0 0 0 0 0 0	1 1 1 1 1	0 0 0 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	

Appendix B

Installing a RISC-V Toolchain

B.1 The GNU Toolchain

Discuss 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 proper variations on this 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 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 5.

B.2 rvddt

Discuss installing the rvddt simulator here.

Appendix C

Using The RISC-V GNU Toolchain

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

See Appendix B if you do not already have the GNU crosscompiler toolchain available on your system.

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 5.

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

Describe the pre-processor, compiler, assembler 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

2319 hexdump -C

Appendix D

Floating Point Numbers

D.1 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:

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

  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.

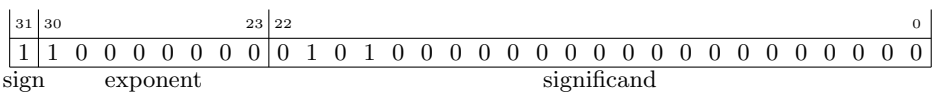
- The format of a number in scientific notation is $mantissa \times base^{exponent}$

- 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.



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

- 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 number-line diagram showing where the over/under-flow areas are and why.

D.1.1 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.

D.1.1.1 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 D.1: `powersoftwo.c`
Precise Powers of Two

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 union floatbin
6 {
7     unsigned int    i;
8     float           f;
9 };
```

```
10 int main()
11 {
12     union floatbin  x;
13     union floatbin  y;
14     int             i;
15     x.f = 1.0;
16     while (x.f > 1.0/1024.0)
17     {
18         y.f = -x.f;
19         printf("%25.10f = %08x     %25.10f = %08x\n", x.f, x.i, y.f, y.i);
20         x.f = x.f/2.0;
21     }
22 }
```

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

```
1  1.0000000000 = 3f800000                  -1.0000000000 = bf800000
2  0.5000000000 = 3f000000                  -0.5000000000 = bf000000
3  0.2500000000 = 3e800000                  -0.2500000000 = be800000
4  0.1250000000 = 3e000000                  -0.1250000000 = be000000
5  0.0625000000 = 3d800000                  -0.0625000000 = bd800000
6  0.0312500000 = 3d000000                  -0.0312500000 = bd000000
7  0.0156250000 = 3c800000                  -0.0156250000 = bc800000
8  0.0078125000 = 3c000000                  -0.0078125000 = bc000000
9  0.0039062500 = 3b800000                  -0.0039062500 = bb800000
10 0.0019531250 = 3b000000                  -0.0019531250 = bb000000
```

D.1.1.2 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 D.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 D.3: `cleandecimal.c`
Print Clean Decimal Numbers

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  union floatbin
6  {
7      unsigned int    i;
8      float           f;
9  };
10 int main()
11 {
12     union floatbin  x, y;
13     int             i;
14
15     x.f = 10;
16     while (x.f <= 10000000000000.0)
17     {
18         y.f = -x.f;
```

```
        printf("%25.10f = %08x     %25.10f = %08x\n", x.f, x.i, y.f, y.i);
        x.f = x.f*10.0;
    }
}
```

Listing D.4: `cleandecimal.out`
Output from `cleandecimal.c`

```
             10.0000000000 = 41200000               -10.0000000000 = c1200000
            100.0000000000 = 42c80000              -100.0000000000 = c2c80000
           1000.0000000000 = 447a0000             -1000.0000000000 = c47a0000
          10000.0000000000 = 461c4000            -10000.0000000000 = c61c4000
         100000.0000000000 = 47c35000           -100000.0000000000 = c7c35000
        1000000.0000000000 = 49742400          -1000000.0000000000 = c9742400
       10000000.0000000000 = 4b189680         -10000000.0000000000 = cb189680
      100000000.0000000000 = 4cbebc20        -100000000.0000000000 = ccbebc20
     1000000000.0000000000 = 4e6e6b28       -1000000000.0000000000 = ce6e6b28
    10000000000.0000000000 = 501502f9      -10000000000.0000000000 = d01502f9
    99999997952.0000000000 = 51ba43b7      -99999997952.0000000000 = d1ba43b7
   999999995904.0000000000 = 5368d4a5     -999999995904.0000000000 = d368d4a5
  9999999827968.0000000000 = 551184e7    -9999999827968.0000000000 = d51184e7
```

D.1.1.3 Accumulation of Error

These rounding errors can be exaggerated when the number we multiply the `x.f` value by is, itself, something that can not be accurately represented in IEEE form.[1]

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.

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

Listing D.5: `erroraccumulation.c`
Accumulation of Error

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

union floatbin
{
    unsigned int    i;
    float           f;
};
int main()
{
    union floatbin  x, y;
    int             i;

    x.f = .1;
    while (x.f <= 2.0)
    {
        y.f = -x.f;
        printf("%25.10f = %08x     %25.10f = %08x\n", x.f, x.i, y.f, y.i);
        x.f += .1;
    }
}
```

Listing D.6: `erroraccumulation.out`
Output from `erroraccumulation.c`

[1] Applications 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.

```
0.1000000015 = 3dcccccd                  -0.1000000015 = bdcccccd
0.2000000030 = 3e4ccccd                  -0.2000000030 = be4ccccd
0.3000000119 = 3e99999a                  -0.3000000119 = be99999a
0.4000000060 = 3ecccccd                  -0.4000000060 = becccccd
0.5000000000 = 3f000000                  -0.5000000000 = 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
1.1000001431 = 3f8cccce                  -1.1000001431 = bf8cccce
1.2000001669 = 3f99999b                  -1.2000001669 = bf99999b
1.3000001907 = 3fa66668                  -1.3000001907 = bfa66668
1.4000002146 = 3fb33335                  -1.4000002146 = bfb33335
1.5000002384 = 3fc00002                  -1.5000002384 = bfc00002
1.6000002623 = 3fcccccf                  -1.6000002623 = bfcccccf
1.7000002861 = 3fd9999c                  -1.7000002861 = bfd9999c
1.8000003099 = 3fe66669                  -1.8000003099 = bfe66669
1.9000003338 = 3ff33336                  -1.9000003338 = bff33336
```

D.1.2 Reducing Error Accumulation

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

Listing D.7: `errorcompensation.c`
Accumulation of Error

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

union floatbin
{
    unsigned int    i;
    float           f;
};
int main()
{
    union floatbin  x, y;
    int             i;

    i = 1;
    while (i <= 20)
    {
        x.f = i/10.0;
        y.f = -x.f;
        printf("%25.10f = %08x     %25.10f = %08x\n", x.f, x.i, y.f, y.i);
        i++;
    }
    return(0);
}
```

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

```
0.1000000015 = 3dcccccd                  -0.1000000015 = bdcccccd
0.2000000030 = 3e4ccccd                  -0.2000000030 = be4ccccd
0.3000000119 = 3e99999a                  -0.3000000119 = be99999a
0.4000000060 = 3ecccccd                  -0.4000000060 = becccccd
0.5000000000 = 3f000000                  -0.5000000000 = bf000000
```

```
0.6000000238 = 3f19999a                    -0.6000000238 = bf19999a
0.6999999881 = 3f333333                    -0.6999999881 = bf333333
0.8000000119 = 3f4ccccd                    -0.8000000119 = bf4ccccd
0.8999999762 = 3f666666                    -0.8999999762 = bf666666
1.0000000000 = 3f800000                    -1.0000000000 = bf800000
1.1000000238 = 3f8ccccd                    -1.1000000238 = bf8ccccd
1.2000000477 = 3f99999a                    -1.2000000477 = bf99999a
1.2999999523 = 3fa66666                    -1.2999999523 = bfa66666
1.3999999762 = 3fb33333                    -1.3999999762 = bfb33333
1.5000000000 = 3fc00000                    -1.5000000000 = bfc00000
1.6000000238 = 3fcccccd                    -1.6000000238 = bfcccccd
1.7000000477 = 3fd9999a                    -1.7000000477 = bfd9999a
1.7999999523 = 3fe66666                    -1.7999999523 = bfe66666
1.8999999762 = 3ff33333                    -1.8999999762 = bff33333
2.0000000000 = 40000000                    -2.0000000000 = c0000000
```

Appendix E

The ASCII Character Set

A slightly abriged version of the Linux "ASCII" man(1) page.

E.1 NAME

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

E.2 DESCRIPTION

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

The following table contains the 128 ASCII characters.

C program '\X' escapes are noted.

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

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

E.2.1 Tables

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

```
     2 3 4 5 6 7
    -------------
0:   0 @ P ` p
1: ! 1 A Q a q
2: " 2 B R b r
3: # 3 C S c s
4: $ 4 D T d t
5: % 5 E U e u
6: & 6 F V f v
7: ' 7 G W g w
8: ( 8 H X h x
9: ) 9 I Y i y
A: * : J Z j z
B: + ; K [ k {
C: , < L \ l |
D: - = M ] m }
E: . > N ^ n ~
F: / ? O _ o DEL
```

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

E.3 NOTES

E.3.1 History

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

On older terminals, the underscore code is displayed as a left arrow, called backarrow, the caret is displayed as an up-arrow and the vertical bar has a hole in the middle.

Uppercase and lowercase characters differ by just one bit and the ASCII character 2 differs from the double quote by just one bit, too. That made it much easier to encode characters mechanically or with a non-microcontroller-based electronic keyboard and that pairing was found on old teletypes.

The ASCII standard was published by the United States of America Stan- dards Institute (USASI) in 1968.

E.4 COLOPHON

This page is part of release 4.04 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at http://www.kernel.org/doc/man-pages/.

Appendix F

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 THEORY (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.

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. viii, 3, 4, 16, 20, 23, 28, 45, 47, 48, 55, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66

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

[3] D. Patterson and J. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 4 2017. ISBN: 978-0128122754. viii, 23

[4] W. F. Decker, "A modern approach to teaching computer organization and assembly language programming," *SIGCSE Bull.*, vol. 17, pp. 38–44, 12 1985. viii

[5] Texas Instruments, *SN54190, SN54191, SN54LS190, SN54LS191, SN74190, SN74191, SN74LS190, SN74LS191 Synchronous Up/Down Counters With Down/Up Mode Control*, 3 1988. viii

[6] Texas Instruments, *SN54154, SN74154 4–line to 16–line Decoders/Demultiplexers*, 12 1972. viii

[7] Intel, *MCS-85 User's Manual*, 9 1978. viii

[8] Radio Shack, *TRS-80 Editor/Assembler Operation and Reference Manual*, 1978. viii

[9] Motorola, *MC68000 16–bit Microprocessor User's Manual*, 2nd ed., 1 1980. MC68000UM(AD2). viii

[10] R. A. Overbeek and W. E. Singletary, *Assembler Language With ASSIST*. Science Research Associates, Inc., 2nd ed., 1983. viii

[11] IBM, *IBM System/370 Principals of Operation*, 7th ed., 3 1980. viii

[12] IBM, *OS/VS-DOS/VSE-VM/370 Assembler Language*, 6th ed., 3 1979. viii

[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. 20

[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. 76

[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 Coprporation, *Series 32000 Databook*, 1986.

Glossary

address A numeric value used to uniquely identify each byte of main memory. 2, 91

alignment Refers to a range of numeric values that begin at a multiple of some number. Primarily 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. 22, 91

ASCII American Standard Code for Information Interchange. See Appendix E. 18, 91

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 uses to write decimal numbers every day. 26, 27, 91, 92

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. 1, 3, 91, 92, 93

bit One binary digit. 3, 6, 10, 91, 92, 93

byte A binary value represented by 8 bits. 2, 6, 91, 92, 93

CPU Central Processing Unit. 1, 2, 91

doubleword A binary value represented by 64 bits. 91

exception An error encountered by the CPU while executing an instruction that can not be completed. 23, 91

fullword A binary value represented by 32 bits. 6, 91

halfword A binary value represented by 16 bits. 6, 19, 91

hart Hardware Thread. 3, 91

hexadecimal A base-16 numbering system whose digits are 0123456789abcdef. The hex digits (hits) are not case-sensitive. 26, 27, 91, 92

high order bits Some number of MSBs. 91

hit One hexadecimal digit. 10, 11, 91, 92, 93

ISA Instruction Set Architecture. 3, 4, 91

LaTeX Is a mark up language specially suited for scientific documents. 91

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". 91

low order bits Some number of LSBs. 91

LSB Least Significant Bit. 10, 12, 19, 37, 40, 42, 91, 93

machine language The instructions that are executed by a CPU that are expressed in the form of binary values. 1, 91

mnemonic 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 binary values of each machine instruction. 1, 91

MSB Most Significant Bit. 10, 12, 13, 16, 17, 19, 37, 38, 42, 91, 92

nybble Half of a *byte* is a *nybble* (sometimes spelled nibble.) Another word for *hit*. 10, 91

overflow The situation where the result of an addition or subtraction operation is approaching positive or negative infinity and exceeds the number of bits allotted to contain the result. This is typically caused by high-order truncation. 79, 91

program A ordered list of one or more instructions. 1, 91

quadword A binary value represented by 128 bits. 91

RAM Random Access Memory. 2, 91

register A unit of storage inside a CPU with the capacity of XLEN bits. 2, 91, 93

ROM Read Only Memory. 2, 91

RV32 Short for RISC-V 32. The number 32 refers to the XLEN. 45, 91

RV64 Short for RISC-V 64. The number 64 refers to the XLEN. 91

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. 18, 25, 91

thread An stream of instructions. When plural, it is used to refer to the ability of a CPU to execute multiple instruction streams at the same time. 3, 91

underflow The situation where the result of an addition or subtraction operation is approaching zero and exceeds the number of bits allotted to contain the result. This is typically caused by low-order truncation. 79, 91

XLEN The number of bits a RISC-V x integer register (such as x0). For RV32 XLEN=32, RV64 XLEN=64 and so on. 42, 43, 91, 93

Index

S

RV32I Reference Card

add	rd, rs1, rs2	Add	rd ← rs1 + rs2, pc ← pc+4
addi	rd, rs1, imm	Add Immediate	rd ← rs1+sx(imm), pc ← pc+4
and	rd, rs1, rs2	And	rd ← rs1 & rs2, pc ← pc+4
andi	rd, rs1, imm	And Immediate	rd ← rs1 & sx(imm), pc ← pc+4
auipc	t0, 3	Add Upper Immediate to PC	rd ← pc + zr(imm), pc ← pc+4
beq	rs1, rs2, imm	Branch Equal	pc ← (rs1==rs2) ? pc+sx(imm[12:1]<<1) : pc+4
bge	rs1, rs2, imm	Branch Greater or Equal	pc ← (rs1>=rs2) ? pc+sx(imm[12:1]<<1) : pc+4
bgeu	rs1, rs2, imm	Branch Greater or Equal Unsigned	pc ← (rs1>=rs2) ? pc+sx(imm[12:1]<<1) : pc+4
blt	rs1, rs2, imm	Branch Less Than	pc ← (rs1<rs2) ? pc+sx(imm[12:1]<<1) : pc+4
bltu	rs1, rs2, imm	Branch Less Than Unsigned	pc ← (rs1<rs2) ? pc+sx(imm[12:1]<<1) : pc+4
bne	rs1, rs2, imm	Branch Not Equal	pc ← (rs1!=rs2) ? pc+sx(imm[12:1]<<1) : pc+4
jal	rd, imm	Jump And Link	rd ← pc+4, pc ← pc+sx(imm<<1)
jalr	rd, rs1, imm	Jump And Link Register	rd ← pc+4, pc ← (rs1+sx(imm))&~1
lb	rd, imm(rs1)	Load Byte	rd ← sx(m8(rs1+sx(imm))), pc ← pc+4
lbu	rd, imm(rs1)	Load Byte Unsigned	rd ← zx(m8(rs1+sx(imm))), pc ← pc+4
lh	rd, imm(rs1)	Load Halfword	rd ← sx(m16(rs1+sx(imm))), pc ← pc+4
lhu	rd, imm(rs1)	Load Halfword Unsigned	rd ← zx(m16(rs1+sx(imm))), pc ← pc+4
lui	t0, 3	Load Upper Immediate	rd ← zr(imm), pc ← pc+4
lw	rd, imm(rs1)	Load Word	rd ← sx(m32(rs1+sx(imm))), pc ← pc+4
or	rd, rs1, rs2	Or	rd ← rs1 | rs2, pc ← pc+4
ori	rd, rs1, imm	Or Immediate	rd ← rs1 | sx(imm), pc ← pc+4
sb	rs2, imm(rs1)	Store Byte	m8(rs1+sx(imm)) ← rs2[7:0], pc ← pc+4
sh	rs2, imm(rs1)	Store Halfword	m16(rs1+sx(imm)) ← rs2[15:0], pc ← pc+4
sll	rd, rs1, rs2	Shift Left Logical	rd ← rs1 << rs2, pc ← pc+4
slli	rd, rs1, shamt	Shift Left Logical Immediate	rd ← rs1 << shamt, pc ← pc+4
slt	rd, rs1, rs2	Set Less Than	rd ← rs1 < rs2) ? 1 : 0, pc ← pc+4
slti	rd, rs1, imm	Set Less Than Immediate	rd ← (rs1 < sx(imm)) ? 1 : 0, pc ← pc+4
sltiu	rd, rs1, imm	Set Less Than Immediate Unsigned	rd ← (rs1 < sx(imm)) ? 1 : 0, pc ← pc+4
sltu	rd, rs1, rs2	Set Less Than Unsigned	rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4
sra	rd, rs1, rs2	Shift Right Arithmetic	rd ← rs1 >> rs2, pc ← pc+4
srai	rd, rs1, shamt	Shift Right Arithmetic Immediate	rd ← rs1 >> shamt, pc ← pc+4
srl	rd, rs1, rs2	Shift Right Logical	rd ← rs1 >> rs2, pc ← pc+4
srli	rd, rs1, shamt	Shift Right Logical Immediate	rd ← rs1 >> shamt, pc ← pc+4
sub	rd, rs1, rs2	Subtract	rd ← rs1 - rs2, pc ← pc+4
sw	rs2, imm(rs1)	Store Word	m16(rs1+sx(imm)) ← rs2[31:0], pc ← pc+4
xor	rd, rs1, rs2	Exclusive Or	rd ← rs1 ^ rs2, pc ← pc+4
xori	rd, rs1, imm	Exclusive Or Immediate	rd ← rs1 ^ sx(imm), pc ← pc+4