RISC-V Assembly Language Programming

(Draft v0.16-0-g9de9145)

John Winans
jwinans@niu.edu

April 18, 2022

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

Contents

Preface

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

The closest published work on this topic appear to 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 on the Internet that are suitable for those who 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 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 assembly language from data sheets and texts, that are still sitting on my shelves today, 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

All of these manuals 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 curriculum material I created 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 processes data represented using numeric values (digits), most commonly expressed in binary (ones and zeros) form.

This text focuses on digital computing.

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

1.1.1 Storage Systems

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

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

1.1.1.1 Volatile Storage

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

One type of volatile storage is provided inside the CPU itself in small blocks called registers. These 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 each assigned a 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 will focus on volatile storage.

1.1.2 CPU

The CPU is a collection of registers and circuitry designed to manipulate the register data and to exchange data and instructions with the main memory. The instructions that are read from the main memory tell the CPU to perform various mathematical and logical operations on the data in its registers and where to save the results of those operations.

➽ 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).

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 of 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 concerned with the peripherals of a computer system other than in sections where instructions are discussed with the purpose of addressing 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

The catalog of rules that describes the details of the instructions and features that a given CPU provides is called an 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 continuous repeats of a 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 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

In order to *fetch* an instruction from the main memory the CPU will update the address in the `pc` register and then request that the main memory return the value of the data stored at that address. [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 means inspecting the portions of the instruction that dictate which registers are involved and what that, if anything, ALU 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.

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.

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

> If the value in x8 is currently less than the value in x24 then proceed to the instruction at the next main memory address, otherwise branch to an instruction at a different address.

This type of instruction can therefore result in one of two different actions pending the result 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 or false and *binary* refers to the use of a base-2 numeric system to represent numbers.

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

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

2.1 Boolean Functions

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

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

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

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

2.1.1 NOT

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

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

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

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

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

A	B	A⊕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 is 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 128_{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:

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

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 exponent (the one furthest to the left for any given base) as the most significant digit and the place value with the lowest exponent 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 binary numbers for this pattern are 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.

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

2.2.1 Converting Between Bases

2.2.1.1 From Binary to Decimal

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

The four-bit mapping between binary and hex makes this task as straight forward as using a look-up table to translate each hit (Hex digIT) 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 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.

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

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

Adding two unsigned binary numbers using 16 full adders:

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

Note that the carry "into" the LSB is zero.

2.2.3 Signed Numbers

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

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

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

```
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:

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

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

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

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

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

```
       1    <== carries
     000100 <== 4 = 0 + 0 + 0 + 4 + 0 + 0
    +000101 <== 5 = 0 + 0 + 0 + 4 + 0 + 1
    -------
     001001 <== 9 = 0 + 0 + 8 + 0 + 0 + 1
```

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

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

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

```
   -128 64 32 16  8  4  2  1 <== place value
   1  1  1  1  1  1  1  1    <== 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 **must** be discarded.*

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, negating 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  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 <== 1 (with a truncation)
```

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 method will work for both signed and unsigned numbers!

➽ 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.

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

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

```
   -128 64 32 16  8  4  2  1
      1  1  1  1  1  1  0  0 <== -4  (minuend)
    - 0  0  0  0  1  0  0  0 <== 8   (subtrahend)
    ------------------------


  1   1  1  1  1  1  1  1  1 <== carries
      1  1  1  1  1  1  0  0 <== -4
    + 1  1  1  1  0  1  1  1 <== one's complement of 8
    ------------------------
  1   1  1  1  1  0  1  0  0 <== -12
```

2.2.5 Truncation

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

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

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

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

```
     -8  4  2  1
      1  0  0  0 <== -8  (minuend)
    - 0  0  0  1 <==  1  (subtrahend)
    ------------


  1           1 <== carries
      1  0  0  0 <== -8
    + 1  1  1  0 <== one's complement of 1
      ----------
  1   0  1  1  1 <== this SHOULD be -9 but with truncation it is 7
```

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

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

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

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

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

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

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

2.2.5.1 Unsigned Overflow

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

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

Figure 2.2 illustrates an unsigned overflow during addition:

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

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

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

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

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

Figure 2.3 illustrates an unsigned overflow during subtraction:

```
   0 0 0 0 0 0 1 1 <== 3 (minuend)
 - 0 0 0 0 0 1 0 0 <== 4 (subtrahend)
 -----------------


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

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

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

2.2.5.2 Signed Overflow

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2.7: $-2 + 10 = 8$

Just like an unsigned number can wrap around as a result of successive additions, a signed number can so the same thing. The only difference is that signed numbers won't wrap from the maximum value back to zero, instead it will wrap from the most positive to the most negative value as shown in Figure 2.8.

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

Figure 2.8: $127 + 1 = -128$

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

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
        11111111111111111111 <== -1
11111111111111111111111111111 <== -1
```

As do these:

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

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

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

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

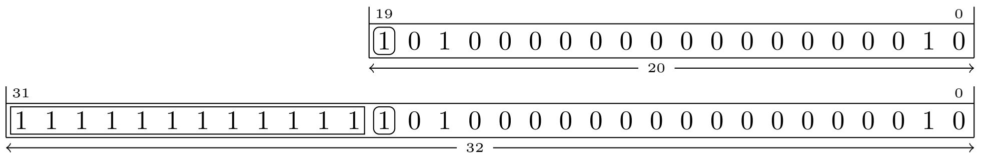


Figure 2.9: Sign-extending a negative integer from 20 bits to 32 bits.

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

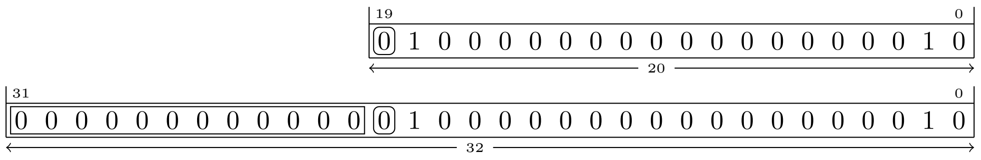


Figure 2.10: Sign-extending a positive integer from 20 bits to 32 bits.

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

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

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

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

➠ Fix Me:
Remove the sign-bit boxes from this figure?

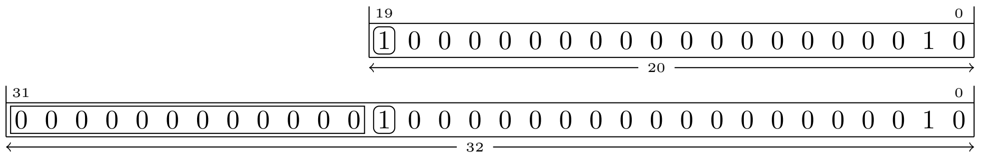


Figure 2.11: Zero-extending an unsigned integer from 20 bits to 32 bits.

2.4 Shifting

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

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

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

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

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

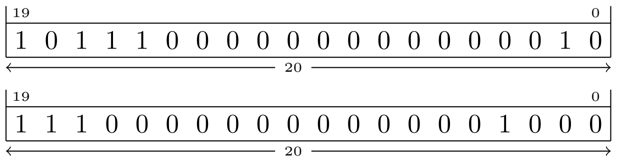
➠ Fix Me:
Add some examples showing the rounding of positive and negative values.

2.4.1 Logical Shifting

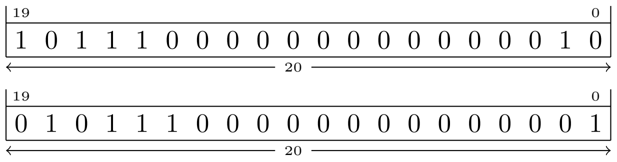
Shifting *logically* to the left or right is a matter of re-aligning the bits in a register and truncating the result.

To shift left two positions:

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



To shift right one position:



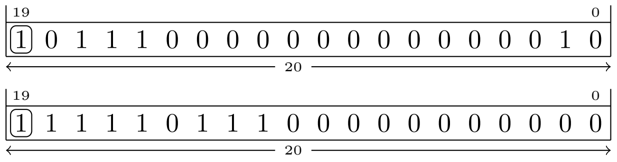
Note that the vacated bit positions are always filled with zero.

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.[6]

[6] 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.

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 showing 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 double-space 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 gap) 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 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?

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

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

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

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

2.5.2.1 Big-Endian

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

- The 8-bit value read from address `0x00002658` would be `0x76`.
- The 8-bit value read from address `0x00002659` would be `0x61`.
- The 8-bit value read from address `0x0000265a` would be `0x6c`.
- The 8-bit value read from address `0x0000265b` would be `0x3d`.
- The 16-bit value read from address `0x00002658` would be `0x7661`.
- The 16-bit value read from address `0x0000265a` would be `0x6c3d`.
- The 32-bit value read from address `0x00002658` would be `0x76616c3d`.

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

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

More examples:

- An 8-bit value read from address `0x00002624` would be `0x23`.
- An 8-bit value read from address `0x00002625` would be `0x24`.
- An 8-bit value read from address `0x00002626` would be `0x81`.
- An 8-bit value read from address `0x00002627` would be `0x00`.
- A 16-bit value read from address `0x00002624` would be `0x2324`.
- A 16-bit value read from address `0x00002626` would be `0x8100`.
- A 32-bit value read from address `0x00002624` would be `0x23248100`.

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

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

2.5.2.2 Little-Endian

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

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

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

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

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

More examples:

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

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

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

In the RISC-V ISA it is noted that

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]

2.5.3 Arrays and Character Strings

While Endianness defines 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 a plurality 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 e of any element $x[n]$ is:

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

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

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

Using this definition, and the memory dump shown in Listing 2.1, and the 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 x are:

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

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

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

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:

```
76 61 6c 3d 00
```

or

```
"val="
```

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 it.

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 will be printed as well as whatever random data bytes in memory follow it until a byte whose value is zero is encountered 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 C) 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.[9]

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 performance of a system.[10]

2.5.6 Instruction Alignment

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

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

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

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

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

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

Chapter 3

The Elements of a Assembly Language Program

3.1 Assembly Language Statements

Introduce the assembly language grammar.

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

3.2 Memory Layout

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

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

3.3 A Sample Program Source Listing

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

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

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

This program listing illustrates a number of things:

- Listings are identified by the name of the file within which they are stored. This listing is from a file named: `zero4regs.S`.
- The assembly language programs discussed in this text will be saved in files that end with: `.S` (Alternately you can use `.sx` on systems that don't understand the difference between upper and lowercase letters.[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 double-space gap in the middle of the line is a reference to make it easier to visually navigate across the line without being forced to count the values from the far left when seeking the value of, say, x5.

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

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

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

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

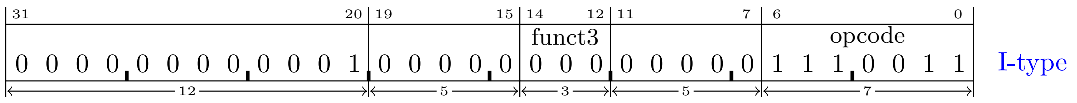
It is a good idea to learn how to stop before learning how to go!

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



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.

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

_start:
    ebreak
```

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

Listing 4.2: `ebreak/ebreak.out`

`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

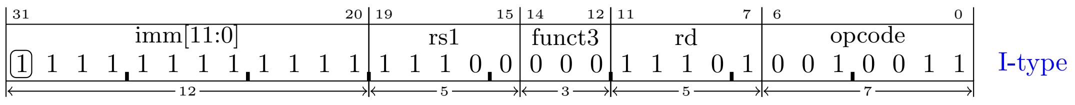
The detailed description of how the `addi` instruction is executed is that it:

➽ 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



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.

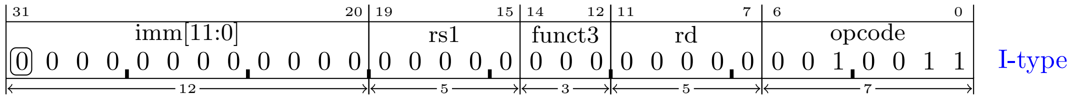
4.2.1 No Operation

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]

An instruction that accomplishes nothing is called a `nop` (sometimes 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



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 instructions as can be seen by comparing it to `objdump` Listing 4.4, and `rvddt` Listing 4.5 output.

Listing 4.3: `nop/nop.S`

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

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

_start:
    addi    x0, x0, 0   # these two instructions assemble into the same thing!
    nop

    ebreak
```

Listing 4.4: `nop/nop.lst`

Using `addi` to perform a `nop`

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

Listing 4.5: `nop/nop.out`

Using `addi` to perform a `nop`

```
$ rvddt -f nop.bin
sp initialized to top of memory: 0x0000fff0
Loading 'nop.bin' to 0x0
This is rvddt.  Enter ? for help.
ddt> d 0 16
 00000000: 13 00 00 00 13 00 00 00  73 00 10 00 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
```

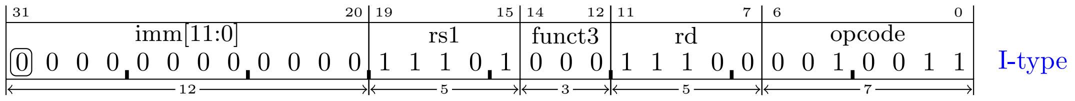
[2]This can happen during the evolution of one portion of code that reduces in size but has to continue to fit into a system without altering any other code... or sometimes you just need to waste a small amount of time in a device driver.

```
   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
```

4.2.2 Copying the Contents of One Register to Another

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

addi t3, t4, 0



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

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

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

```
    .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.

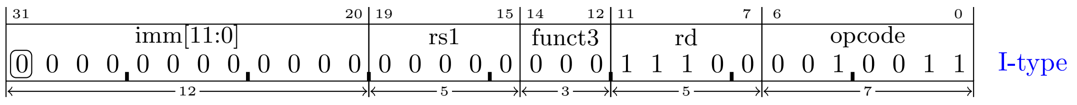
```
mv:     file format elf32-littleriscv
Disassembly of section .text:
00000000 <_start>:
   0:   000e8e13            mv  t3,t4
   4:   000e8e13            mv  t3,t4
   8:   00100073            ebreak
```

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



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

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

_start:
    mv      t3, x0        # t3 = 0

    ebreak
```

Listing 4.9 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.)

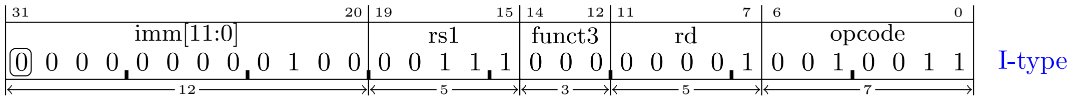
Listing 4.9: mvzero/mv.out
Setting t3 to zero.

```
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



```
    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)
```

4.3 todo

Ideas for the order of introducing instructions.

4.4 Other Instructions With Immediate Operands

```
    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
```

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 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 Jumps

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

```
    jal
    jalr
```

4.10 Pseudoinstructions

```
    li   rd,constant
                     lui      rd,(constant + 0x00000800) >> 12
                     addi     rd,rd,(constant & 0x00000fff)

    la   rd,label
                     auipc    rd,((label-.) + 0x00000800) >> 12
                     addi     rd,rd,((label-(.-4)) & 0x00000fff)

    l{b|h|w} rd,label
                     auipc    rd,((label-.) + 0x00000800) >> 12
                     l{b|h|w} rd,((label-(.-4)) & 0x00000fff)(rd)

    s{b|h|w} rd,label,rt          # rt used as a temp reg for the operation (default=x6)
                     auipc    rt,((label-.) + 0x00000800) >> 12
                     s{b|h|w} rd,((label-(.-4)) & 0x00000fff)(rt)

    call label       auipc    x1,((label-.) + 0x00000800) >> 12
                     jalr     x1,((label-(.-4)) & 0x00000fff)(x1)

    tail label,rt                 # rt used as a temp reg for the operation (default=x6)
                     auipc    rt,((label-.) + 0x00000800) >> 12
                     jalr     x0,((label-(.-4)) & 0x00000fff)(rt)

    mv   rd,rs       addi     rd,rs,0

    j    label       jal      x0,label
    jal  label       jal      x1,label
    jr   rs          jalr     x0,0(rs)
    jalr rs          jalr     x1,0(rs)
    ret              jalr     x0,0(x1)
```

4.10.1 The `li` Pseudoinstruction

Note that the `li` pseudoinstruction includes an (effectively) conditional addition of 1 to the immediate operand in the `lui` instruction. This is because the immediate operand in the `addi` instruction is sign-extended before it is added to `rd`. If the immediate operand to the `addi` has its most-significant-bit set to 1 then it will have the effect of subtracting 1 from the operand in the `lui` instruction.

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

```
    li  x5,0x12345800
```

A naive (incorrect) solution might be:

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

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

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

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

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

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

➽ Fix Me:
Add a ribbon diagram of this?

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

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

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

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

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

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

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

➽ Fix Me:
Find a proper citation for this.

4.10.2 The `la` Pseudoinstruction

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

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

```
00010040     la     x10,var1
00010048 ...                # note that the la pseudoinstruction expands into 8 bytes
...

         var1:
00010900     .word  999       # a 32-bit integer constant stored in memory at address var1
```

The `la` instruction in this example will expand into:

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

Note that `auipc` will shift the immediate operand to the left 12 bits and then add that to the `pc` register (see Figure 5.3.1.)

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

Therefore the expanded pseudoinstruction example will become:

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

After performing the subtractions, it will reduce to this:

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

Continuing to reduce the math operations we get:

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

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

4.11 Relocation

Because expressions that refer to constants and address labels are common in assembly language programs, a shorthand notation is available for calculating the pairs of values that are used in the implementation of things like the `li` and `la` pseudoinstructions (that have to be written to compensate for the sign-extension that will take place in the immediate operand that appears in instructions like `addi` and `jalr`.)

4.11.1 Absolute Addresses

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

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

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

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

4.11.2 PC-Relative Addresses

The following can be used for PC-relative addresses:

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

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

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

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

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

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

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

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

4.12 Relaxation

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

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

See: `https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md`

Chapter 5

RV32 Machine Instructions

5.1 Conventions and Terminology

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

5.1.1 XLEN

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

5.1.2 sx(val)

Sign extend *val* to the left.

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

For more on sign-extension see section 2.3.

5.1.3 zx(val)

Zero extend *val* to the left.

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

For more on zero-extension see Figure 2.3.

5.1.4 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.

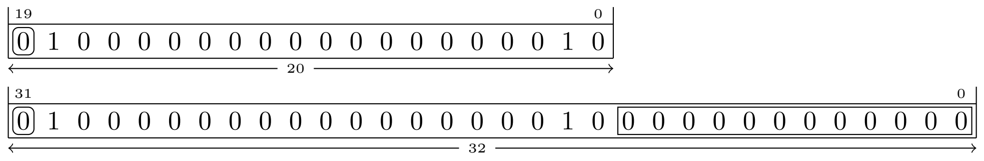


Figure 5.1: Zero-extending an integer to the right from 20 bits to 32 bits.

5.1.5 Sign Extended Left and Zero Extend Right

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

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

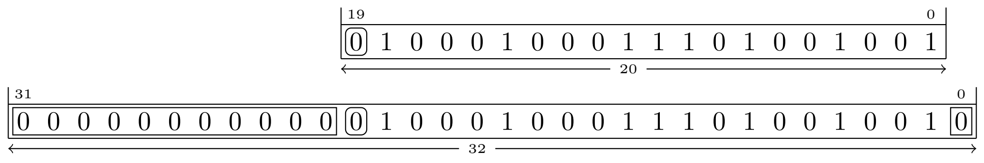


Figure 5.2: Sign-extending a positive 20-bit number 11 bits to the left and one bit to the right.

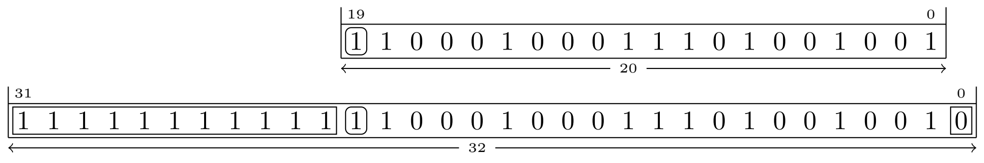


Figure 5.3: Sign-extending a negative 20-bit number 11 bits to the left and one bit to the right.

5.1.6 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(0x42)` refers to the memory location at address 42_{16} that currently contains the 8-bit value $\texttt{fc}_{16}$.

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

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

`m8(addr)` $\leftarrow$ `source`

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

`dest` $\leftarrow$ `m8(addr)`

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

```
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.

5.1.7 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(0x42)` refers to the memory location at address 42_{16} that currently contains $65fc_{16}$. See also section 5.1.6.

5.1.8 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(0x42)` refers to the memory location at address 42_{16} that currently contains $662065fc_{16}$. See also section 5.1.6.

5.1.9 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(0x42)` refers to the memory location at address 42_{16} that currently contains $656c6c69662065fc_{16}$. See also section 5.1.6.

5.1.10 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(0x42)` refers to the memory location at address 42_{16} that currently contains $72206874697720646 56c6c69662065fc_{16}$. See also section 5.1.6.

5.1.11 .+offset

The address of the current instruction plus a numeric offset.

5.1.12 .-offset

The address of the current instruction minus a numeric offset.

5.1.13 pcrel_13

An address that is within $[-4096..4095]$ of the current instruction location. These addresses are typically expressed in assembly source code by using labels. See section 5.3.6 for examples.

5.1.14 pcrel_21

An address that is within $[-1048576..1048575]$ of the current instruction location. These addresses are typically expressed in assembly source code by using labels. See section 5.3.2 for an example.

5.1.15 pc

The current value of the program counter.

5.1.16 rd

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

5.1.17 rs1

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

5.1.18 rs2

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

5.1.19 imm

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

5.1.20 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.

5.2 Addressing Modes

immediate, register, base-displacement, pc-relative

Fix Me: *Write this section.*

5.3 Instruction Encoding Formats

This document concerns itself with the RISC-V instruction formats shown in Figure 5.5.

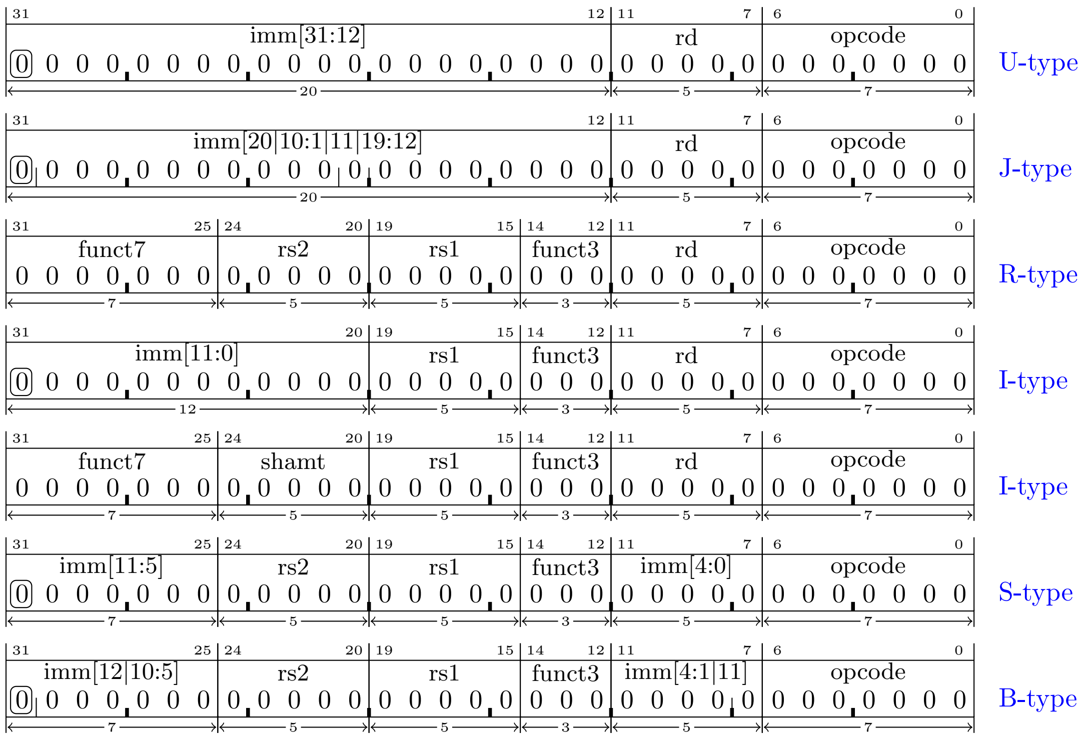


Figure 5.5: RISC-V instruction formats.

The method/format of the instructions has been designed with an eye on the ease of future manufacture of the machine that will execute them. It is easier to build a machine if it does not have to accommodate many different ways to perform the same task. The result is that a machine can be built with fewer gates, consumes less power, and can run faster than if it were built when a priority is on how a user might prefer to decode the same instructions from a hex dump.

Observe that all instructions have their opcode in bits 0-6 and when they include an `rd` register it will be specified in bits 7-11, an `rs1` register in bits 15-19, an `rs2` register in bits 20-24, and so on. This has a seemingly strange impact on the placement of any immediate operands.

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

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

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

5.3.1 U Type

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

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

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

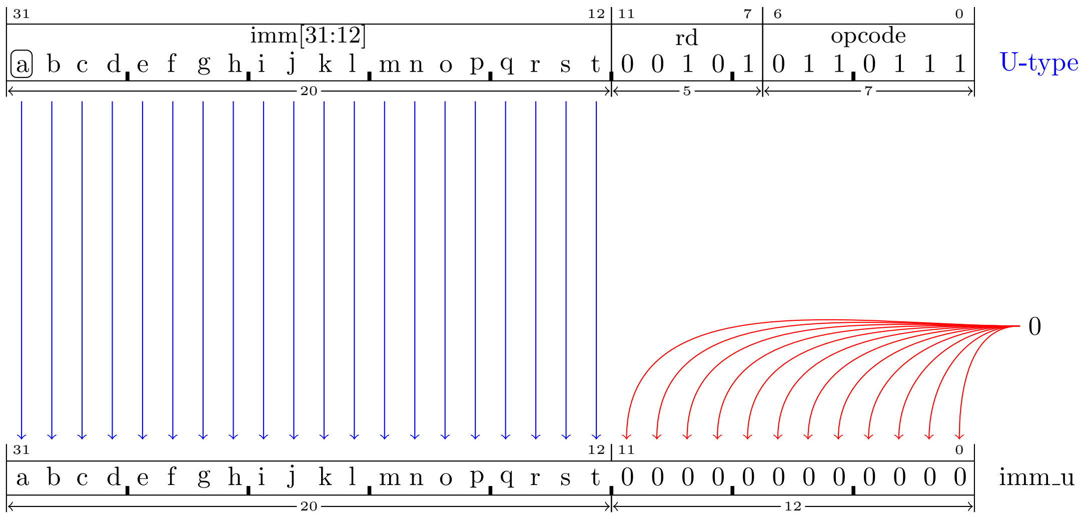

31
12
11
7
6
0
imm[31:12]
rd
opcode
a b c d e f g h i j k l m n o p q r s t 0 0 1 0 1 0 1 1 0 1 1 1
U-type
20
5
7
0
31
12
11
0
a b c d e f g h i j k l m n o p q r s t 0 0 0 0 0 0 0 0 0 0 0 0
imm_u
20
12

Figure 5.6: Decoding a U-type instruction.

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. Leaving the imm bits on the left, in the "upper bits" of the `imm_u` value suggests a rationale for the name of this format.

- `lui   rd,imm`

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

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

- `auipc rd,imm`

Add the address of the instruction to the `imm_u` value as shown Figure 5.6 and store the result in register `rd`.

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

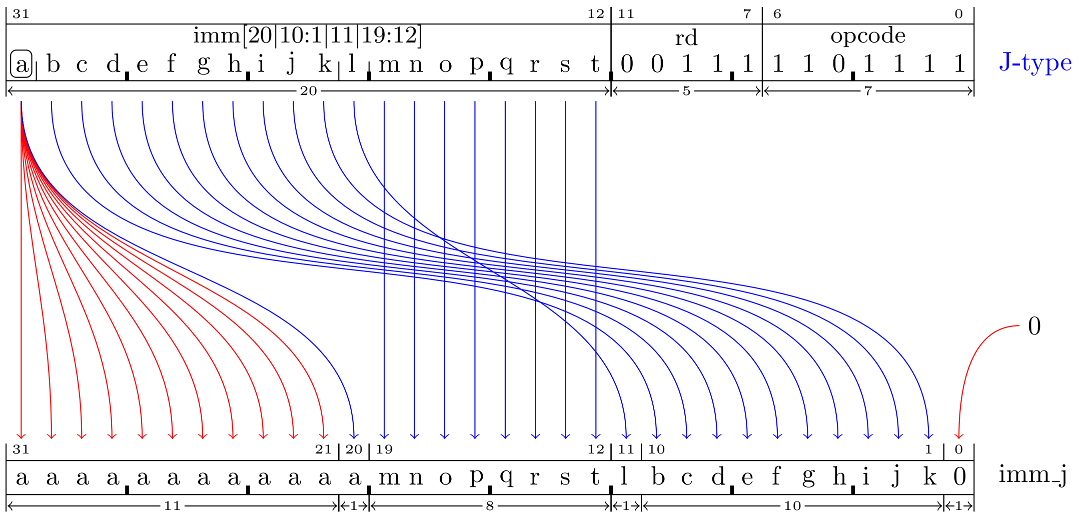
If XLEN=64 then the `imm_u` value in this example will be converted to the same two's complement integer value by extending the sign-bit further to the left.

5.3.2 J Type

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

Note that the `imm_j` value is a 21-bit value in the range of [−1048576..1048575] representing a `pc`-relative offset to the target address.

If XLEN=32 then the *imm* value will extracted from the instruction and converted as shown in Figure 5.7 to form the `imm_j` value.



31
12
11
7
6
0
imm[20|10:1|11|19:12]
rd
opcode
a b c d e f g h i j k l m n o p q r s t 0 0 1 1 1 1 1 0 1 1 1 1
J-type
20
5
7
0
31
21
20
19
12
11
10
1
0
a a a a a a a a a a a a m n o p q r s t l b c d e f g h i j k 0
imm_j
11
1
8
1
10
1

Figure 5.7: Decoding a J-type instruction.

The J-type format is used by the Jump And Link instruction that calculates the target address by adding `imm_j` 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 expressing a wider range of target addresses than the 20-bit imm value alone.

- `jal rd,pcrel_21`

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

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

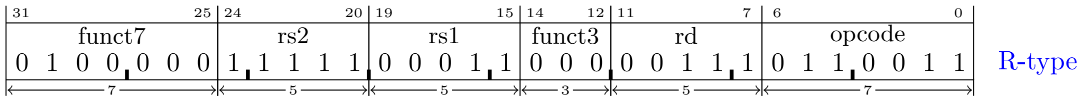
```
00000010: 000002ef  jal    x5,0x10      # jump to self (address 0x10)
00000014: 008002ef  jal    x5,0x1c      # jump to address 0x1c
00000018: 00100073  ebreak
0000001c: 00100073  ebreak
```

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

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

See also section 5.3.6.

5.3.3 R Type



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

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

- `add    rd,rs1,rs2`

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

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

- `and    rd,rs1,rs2`

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

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

- `or     rd,rs1,rs2`

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

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

- `sll    rd,rs1,rs2`

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

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

- `slt    rd,rs1,rs2`

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

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

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

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

- `sltu rd,rs1,rs2`

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

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

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

- `sra rd,rs1,rs2`

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

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

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

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

- `srl rd,rs1,rs2`

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

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

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

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

- `sub rd,rs1,rs2`

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

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

- `xor rd,rs1,rs2`

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

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

5.3.4 I Type

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

If XLEN=32 then the 12-bit *imm* value example will extracted from the instruction and converted as shown in Figure 5.8 to form the `imm_i` value.

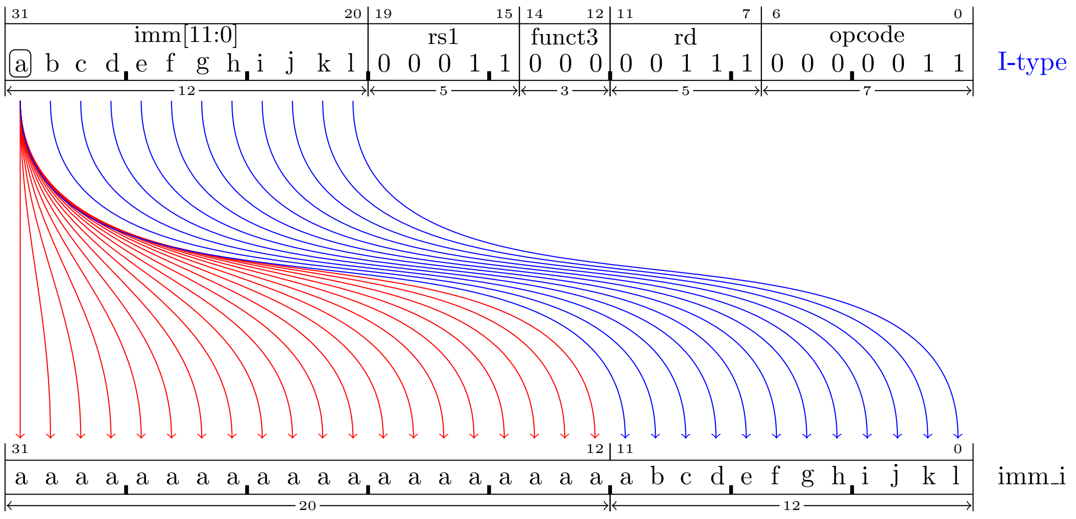


Figure 5.8: Decoding an I-type Instruction.

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

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

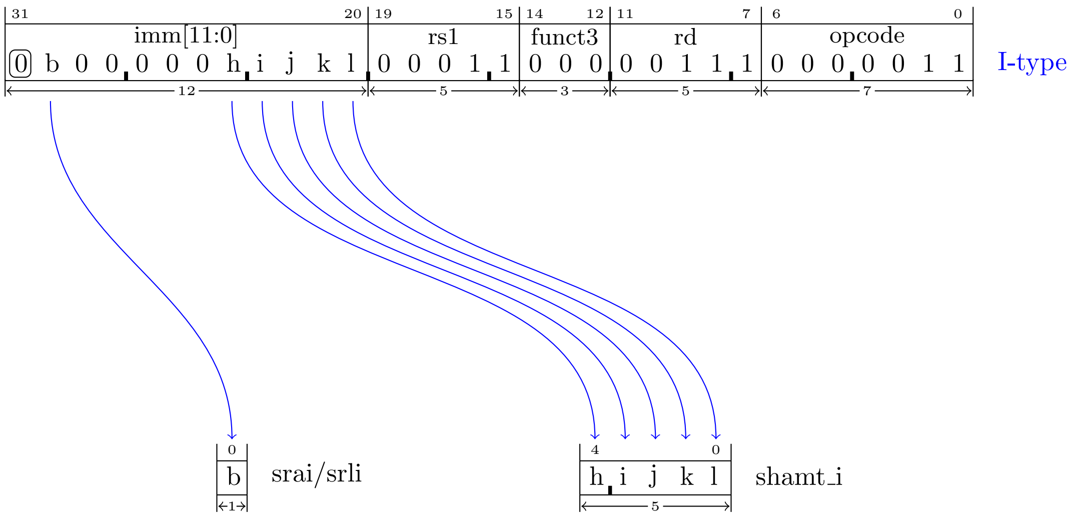


Figure 5.9: Decoding an I-type Shift Instruction.

- `addi rd,rs1,imm`

  Set register `rd` to `rs1 + imm_i`.
- `andi rd,rs1,imm`

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

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

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

Figure 5.10: An Example Memory Dump.

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

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

- `jalr  rd,imm(rs1)`

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

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

- `lb    rd,imm(rs1)`

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

  For example, given the memory contents shown in Figure 5.10, if register `x13` = `0x00002650` then the instruction `lb x12,1(x13)` will set `x12` to the value `0xffffff80`.

- `lbu   rd,imm(rs1)`

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

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

- `lh    rd,imm(rs1)`

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

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

  If register `x13` = `0x00002650` then the instruction `lh x12,-8(x13)` will set `x12` to the value `0xffff87b7`.

- `lhu   rd,imm(rs1)`

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

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

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

- `lw    rd,imm(rs1)`

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

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

- **ori rd,rs1,imm**

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

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

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

- **slli rd,rs1,imm**

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

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

- **slti rd,rs1,imm**

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

- **sltiu rd,rs1,imm**

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

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

- **srai rd,rs1,imm**

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

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

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

- **srli rd,rs1,imm**

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

  For example, if x17 = 0x87654321 then the instruction srli x12,x17,4 will set x12 to the value 0x08765432.

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

- **xori rd,rs1,imm**

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

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

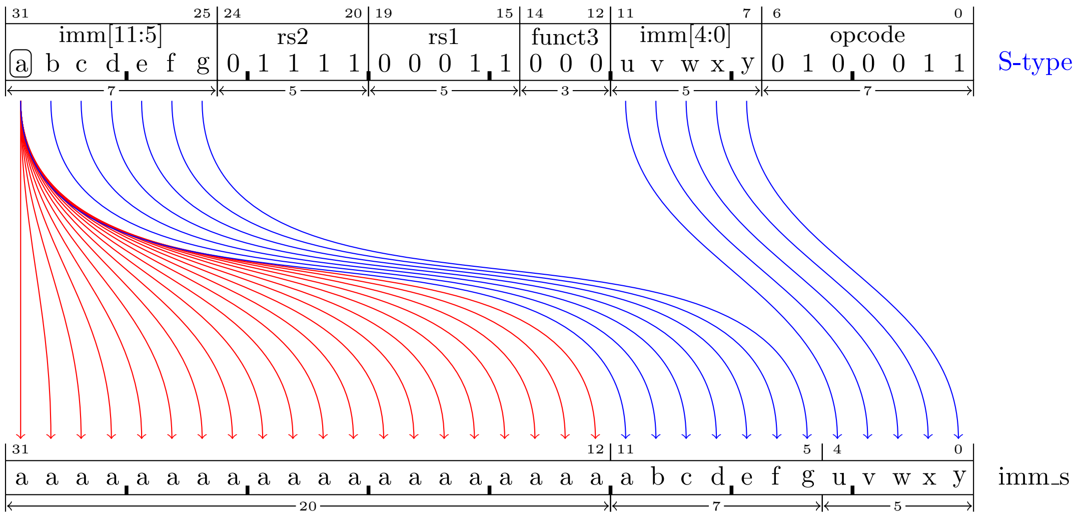
  Recall that imm is sign-extended. Therefore if x17 = 0x55551111 then xori x12,x17,0x800 will set x12 to the value 0xaaaae911.

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

5.3.5 S Type

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

If XLEN=32 then the 12-bit *imm* value example will extracted from the instruction and converted as shown Figure 5.11 to form the `imm_s` value.



31 25 24 20 19 15 14 12 11 7 6 0
imm[11:5]
rs2
rs1
funct3
imm[4:0]
opcode
a b c d e f g 0 1 1 1 1 0 0 0 1 1 0 0 0 u v w x y 0 1 0 0 0 1 1
7 5 5 3 5 7
S-type
31 12 11 5 4 0
a a a a a a a a a a a a a a a a a a a a a b c d e f g u v w x y
imm_s
20 7 5

Figure 5.11: Decoding an S-type Instruction.

- `sb rs2,imm(rs1)`

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

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

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

- `sh rs2,imm(rs1)`

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

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

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

- `sw rs2,imm(rs1)`
  Store the 32-bit value in `rs2` into the memory at the address given by the sum of `rs1` and `imm_s`.
  For example, given the memory contents shown in Figure 5.10, if registers `x13` = `0x00002650` and `x12` = `0x12345678` then the instruction `sw x12,0(x13)` will change the memory word at address `0x00002650` from `0x00008067` to `0x12345678` resulting in:

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

5.3.6 B Type

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

If XLEN=32 then the 12-bit *imm* value example will extracted from the instruction and converted as shown in Figure 5.12 to form the `imm_b` value.

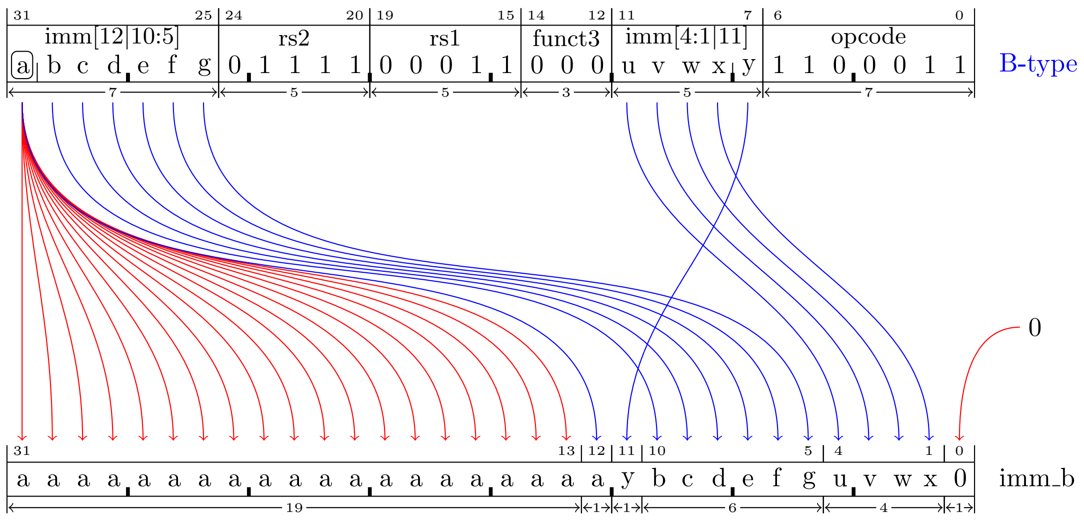


Figure 5.12: Decoding a B-type Instruction.

Note that `imm_b` is expressed in the instruction as a target address that is converted to a 13-bit value in the range of [−4096..4095] representing a `pc`-relative offset to the target address. For example, consider the branch instructions in the following code:

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

The instruction at address `0x0` has a target address of zero and `imm_b` is zero because the offset from the "current instruction" to the target is zero.[4]

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

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

The instruction at address `0x8` has a target address of zero and `imm_b` is `0xfffffff8` (-8) because `0x8 + 0xfffffff8 = 0x0`.

- `beq   rs1,rs2,pcrel_13`

  If `rs1` is equal to `rs2` then add `imm_b` to the `pc` register.
- `bge   rs1,rs2,pcrel_13`

  If the signed value in `rs1` is greater than or equal to the signed value in `rs2` then add `imm_b` to the `pc` register.
- `bgeu  rs1,rs2,pcrel_13`

  If the unsigned value in `rs1` is greater than or equal to the unsigned value in `rs2` then add `imm_b` to the `pc` register.
- `blt   rs1,rs2,pcrel_13`

  If the signed value in `rs1` is less than the signed value in `rs2` then add `imm_b` to the `pc` register.
- `bltu  rs1,rs2,pcrel_13`

  If the unsigned value in `rs1` is less than the unsigned value in `rs2` then add `imm_b` to the `pc` register.
- `bne   rs1,rs2,pcrel_13`

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

5.4 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.

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

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 **??**.

Appendix A

Installing a RISC-V Toolchain

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

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

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

A.1 The GNU Toolchain

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

➽ 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!

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

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

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

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

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

To download, compile and install the toolchain:

➽ Fix Me:
Discuss the choice of ilp32 as well as what the other variations would do.

1850
```
mkdir -p ~/projects/riscv
cd ~/projects/riscv
git clone https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain
INS_DIR=~/projects/riscv/install/rv32i
./configure --prefix=$INS_DIR \
    --with-multilib-generator="rv32i-ilp32--;rv32imafd-ilp32--;rv32ima-ilp32--"
make
```

1851 After building the toolchain, make it available by putting it into your PATH by adding the following
1852 to the end of your `.bashrc` file:

1853 1854 1855
```
export PATH=$PATH:$INS_DIR
```

1856 For this `PATH` change to take place, start a new terminal or paste the same `export` command into
1857 your existing terminal.

1858 A.2 rvddt

1859 Download and install the rvddt simulator by executing the following commands. Building the rvddt
1860 example programs will verify that the GNU toolchain has been built and installed properly.

1861
```
cd ~/projects/riscv
git clone https://github.com/johnwinans/rvddt.git
cd rvddt/src
make world
cd ../examples
make world
```

1862 After building rvddt, make it available by putting it into your PATH by adding the following to the
1863 end of your `.bashrc` file:

1864 1865 1866
```
export PATH=$PATH:~/projects/riscv/rvddt/src
```

1867 For this `PATH` change to take place, start a new terminal or paste the same `export` command into
1868 your existing terminal.

1869 Test the rvddt build by executing one of the examples:

1870
```
winans@ux410:~/projects/riscv/rvddt/examples$ rvddt -f counter/counter.bin
sp initialized to top of memory: 0x0000fff0
Loading 'counter/counter.bin' to 0x0
This is rvddt.  Enter ? for help.
ddt> ti 0 1000
00000000: 00300293  addi    x5, x0, 3     # x5 = 0x00000003 = 0x00000000 + 0x00000003
00000004: 00000313  addi    x6, x0, 0     # x6 = 0x00000000 = 0x00000000 + 0x00000000
00000008: 00130313  addi    x6, x6, 1     # x6 = 0x00000001 = 0x00000000 + 0x00000001
0000000c: fe534ee3  blt     x6, x5, -4    # pc = (0x1 < 0x3) ? 0x8 : 0x10
00000008: 00130313  addi    x6, x6, 1     # x6 = 0x00000002 = 0x00000001 + 0x00000001
0000000c: fe534ee3  blt     x6, x5, -4    # pc = (0x2 < 0x3) ? 0x8 : 0x10
00000008: 00130313  addi    x6, x6, 1     # x6 = 0x00000003 = 0x00000002 + 0x00000001
0000000c: fe534ee3  blt     x6, x5, -4    # pc = (0x3 < 0x3) ? 0x8 : 0x10
00000010: ebreak
ddt> x
winans@ux410:~/projects/riscv/rvddt/examples$
```

1871 ## A.3 qemu

1872 You can download and install the RV32 qemu simulator by executing the following commands.

1873 At the time of this writing (2021-06) I use release v5.0.0. Release v5.2.0 has issues that confuse GDB 1874 when printing the registers and v6.0.0 has different CPU types that I have had trouble with when 1875 executing privileged instructions.

1876
```
INS_DIR=~/projects/riscv/install/rv32i
cd ~/projects/riscv
git clone git@github.com:qemu/qemu.git
cd qemu
git checkout v5.0.0
./configure --target-list=riscv32-softmmu --prefix=${INS_DIR}
make -j4
make install
```

Appendix B

Floating Point Numbers

B.1 IEEE-754 Floating Point Number Representation

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

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

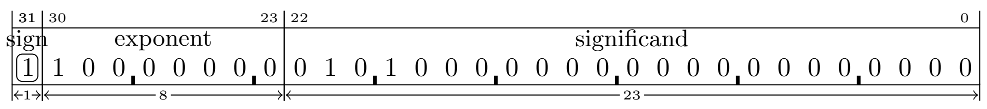
```
... 128 64 32 16 8 4 2 1
```

- 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.' part because *all* normalized floating point numbers will start that way. Thus we can save memory when storing normalized values by inserting a '1.' to the left of significand.



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

- IEEE-754 formats:

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

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

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

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

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

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

- 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.

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

- 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.

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

B.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.

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

B.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 B.1: `powersoftwo.c`
Precise Powers of Two

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

union floatbin
{
    unsigned int    i;
    float           f;
};
int main()
{
    union floatbin  x;
    union floatbin  y;
    int             i;
    x.f = 1.0;
    while (x.f > 1.0/1024.0)
    {
        y.f = -x.f;
        printf("%25.10f = %08x     %25.10f = %08x\n", x.f, x.i, y.f, y.i);
        x.f = x.f/2.0;
    }
}
```

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

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

B.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 B.4 can be perfectly represented in IEEE format. However, a problem arises in the 11Th loop iteration. It is due to the fact that the binary number can not be represented accurately in IEEE format. Its least significant bits were truncated in a best-effort attempt at rounding the value off in order to fit the value into the bits provided. This is an example of *low order truncation*. Once this happens, the value of `x.f` is no longer as precise as it could be given more bits in which to save its value.

Listing B.3: `cleandecimal.c`
Print Clean Decimal Numbers

```
#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 = 10;
    while (x.f <= 10000000000000.0)
    {
        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 B.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
```

B.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.[2]

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.

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

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

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

union floatbin
{
    unsigned int    i;
    float           f;
```

[2]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.

```
};
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 B.6: `erroraccumulation.out`
Output from `erroraccumulation.c`

```
0.1000000015 = 3dccccccd                -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
```

B.1.2 Reducing Error Accumulation

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

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

```
#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 B.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 C

The ASCII Character Set

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

C.1 NAME

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

C.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	

C.2.1 Tables

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

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

C.3 NOTES

C.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 Standards Institute (USASI) in 1968.

C.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 D

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. iv, 3, 4, 16, 25, 27, 32, 59, 82

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

[3] D. Patterson and J. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 4 2017. ISBN: 978-0128122754. iv, 27

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

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

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

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

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

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

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

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

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

[13] "Definition of subtrahend." www.mathsisfun.com/definitions/subtrahend.html. Accessed: 2018-06-02. 17

[14] 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. 22

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

[16] 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.

[17] 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.

[18] 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.

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

Glossary

address A numeric value used to uniquely identify each byte of main memory. 2, 77

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. 26, 77

ASCII American Standard Code for Information Interchange. See Appendix C. 21, 77

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. 23, 30, 31, 77, 78

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, 77, 78, 79

bit One binary digit. 3, 6, 10, 77, 78, 79

byte A binary value represented by 8 bits. 2, 6, 77, 78, 79

CPU Central Processing Unit. 1, 2, 77

doubleword A binary value represented by 64 bits. 77

exception An error encountered by the CPU while executing an instruction that can not be completed. 27, 77

fullword A binary value represented by 32 bits. 6, 77

halfword A binary value represented by 16 bits. 6, 22, 77

hart Hardware Thread. 3, 77

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

high order bits Some number of MSBs. 77

hit One hexadecimal digit. 10, 12, 77, 78, 79

ISA Instruction Set Architecture. 3, 4, 77

LaTeX Is a mark up language specially suited for scientific documents. 77

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, the big-endian number written as "1234" would be written in little endian form as "4321". 23, 77

low order bits Some number of LSBs. 77

LSB Least Significant Bit. 10, 12, 22, 44, 47, 54, 56, 77, 79

machine language The instructions that are executed by a CPU that are expressed in the form of binary values. 1, 77

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, 77

MSB Most Significant Bit. 10, 12, 13, 19, 22, 44, 45, 77, 78

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

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. 64, 77

place value the numerical value that a digit has as a result of its *position* within a number. For example, the digit 2 in the decimal number 123 is in the ten's place and its place value is 20. 9, 11, 23, 24, 77

program A ordered list of one or more instructions. 1, 77

quadword A binary value represented by 128 bits. 77

RAM Random Access Memory. 2, 77

register A unit of storage inside a CPU with the capacity of XLEN bits. 2, 77, 79

ROM Read Only Memory. 2, 77

RV32 Short for RISC-V 32. The number 32 refers to the XLEN. 77

RV64 Short for RISC-V 64. The number 64 refers to the XLEN. 77

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. 21, 29, 77

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, 77

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. 64, 77

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

Index

RV32I Reference Card

Usage Template	Type	Description	Detailed Description
add rd, rs1, rs2	R	Add	`rd ← rs1 + rs2, pc ← pc+4`
addi rd, rs1, imm	I	Add Immediate	`rd ← rs1 + imm_i, pc ← pc+4`
and rd, rs1, rs2	R	And	`rd ← rs1 & rs2, pc ← pc+4`
andi rd, rs1, imm	I	And Immediate	`rd ← rs1 & imm_i, pc ← pc+4`
auipc rd, imm	U	Add Upper Immediate to PC	`rd ← pc + imm_u, pc ← pc+4`
beq rs1, rs2, pcrel_13	B	Branch Equal	`pc ← pc + ((rs1==rs2) ? imm_b : 4)`
bge rs1, rs2, pcrel_13	B	Branch Greater or Equal	`pc ← pc + ((rs1>=rs2) ? imm_b : 4)`
bgeu rs1, rs2, pcrel_13	B	Branch Greater or Equal Unsigned	`pc ← pc + ((rs1>=rs2) ? imm_b : 4)`
blt rs1, rs2, pcrel_13	B	Branch Less Than	`pc ← pc + ((rs1<rs2) ? imm_b : 4)`
bltu rs1, rs2, pcrel_13	B	Branch Less Than Unsigned	`pc ← pc + ((rs1<rs2) ? imm_b : 4)`
bne rs1, rs2, pcrel_13	B	Branch Not Equal	`pc ← pc + ((rs1!=rs2) ? imm_b : 4)`
jal rd, pcrel_21	J	Jump And Link	`rd ← pc+4, pc ← pc+imm_j`
jalr rd, imm(rs1)	I	Jump And Link Register	`rd ← pc+4, pc ← (rs1+imm_i)&~1`
lb rd, imm(rs1)	I	Load Byte	`rd ← sx(m8(rs1+imm_i)), pc ← pc+4`
lbu rd, imm(rs1)	I	Load Byte Unsigned	`rd ← zx(m8(rs1+imm_i)), pc ← pc+4`
lh rd, imm(rs1)	I	Load Halfword	`rd ← sx(m16(rs1+imm_i)), pc ← pc+4`
lhu rd, imm(rs1)	I	Load Halfword Unsigned	`rd ← zx(m16(rs1+imm_i)), pc ← pc+4`
lui rd, imm	U	Load Upper Immediate	`rd ← imm_u, pc ← pc+4`
lw rd, imm(rs1)	I	Load Word	`rd ← sx(m32(rs1+imm_i)), pc ← pc+4`
or rd, rs1, rs2	R	Or	`rd ← rs1 | rs2, pc ← pc+4`
ori rd, rs1, imm	I	Or Immediate	`rd ← rs1 | imm_i, pc ← pc+4`
sb rs2, imm(rs1)	S	Store Byte	`m8(rs1+imm_s) ← rs2[7:0], pc ← pc+4`
sh rs2, imm(rs1)	S	Store Halfword	`m16(rs1+imm_s) ← rs2[15:0], pc ← pc+4`
sll rd, rs1, rs2	R	Shift Left Logical	`rd ← rs1 << (rs2%XLEN), pc ← pc+4`
slli rd, rs1, shamt	I	Shift Left Logical Immediate	`rd ← rs1 << shamt_i, pc ← pc+4`
slt rd, rs1, rs2	R	Set Less Than	`rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4`
slti rd, rs1, imm	I	Set Less Than Immediate	`rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4`
sltiu rd, rs1, imm	I	Set Less Than Immediate Unsigned	`rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4`
sltu rd, rs1, rs2	R	Set Less Than Unsigned	`rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4`
sra rd, rs1, rs2	R	Shift Right Arithmetic	`rd ← rs1 >> (rs2%XLEN), pc ← pc+4`
srai rd, rs1, shamt	I	Shift Right Arithmetic Immediate	`rd ← rs1 >> shamt_i, pc ← pc+4`
srl rd, rs1, rs2	R	Shift Right Logical	`rd ← rs1 >> (rs2%XLEN), pc ← pc+4`
srli rd, rs1, shamt	I	Shift Right Logical Immediate	`rd ← rs1 >> shamt_i, pc ← pc+4`
sub rd, rs1, rs2	R	Subtract	`rd ← rs1 - rs2, pc ← pc+4`
sw rs2, imm(rs1)	S	Store Word	`m32(rs1+imm_s) ← rs2[31:0], pc ← pc+4`
xor rd, rs1, rs2	R	Exclusive Or	`rd ← rs1 ^ rs2, pc ← pc+4`
xori rd, rs1, imm	I	Exclusive Or Immediate	`rd ← rs1 ^ imm_i, pc ← pc+4`

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

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