RISC-V
Assembly Language Programming

(Draft v0.1-0-g2323d75)

John Winans
jwinans@niu.edu

April 29, 2018

Contents

Preface

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

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

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

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

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

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

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

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

Chapter 1

Number Systems

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

1.1 Integers

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

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

Figure Figure 1.1 shows an abridged table of the decimal, binary and hexadecimal values from 0 to 129.

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

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

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

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

And the same for the hexadecimal value:

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

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

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

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 1.1: Counting in decimal, binary and hexadecimal.

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

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

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

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

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

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

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

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

1.1.1 Converting Between Bases

1.1.1.1 From Binary to Decimal

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

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

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

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

1.1.1.2 From Binary to Hexadecimal

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

Grouping the bits into sets of four and summing:

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

After the summing, convert each decimal value to hex. The decimal values from 0–9 are the same values in hex. Because we don't have any more numerals to represent the values from 10-15, we use the first 6 letters (See the right–most column of Figure 1.1.) 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
```

1.1.1.3 From Hexadecimal to Binary

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

For each hit (Hex digIT), translate it to its unique four–bit pattern. Perform this task either by memorizing each of the 16 patterns or by converting each hit to decimal first and then converting each four–bit binary value to decimal using the place–value summing method discussed in subsubsection 1.1.1.1.

For example:

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

1.1.1.4 From Decimal to Binary

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

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

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

   0          2048      (too big)
   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.

1.1.1.5 From Decimal to Hex

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

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

1.1.2 Addition of Binary Numbers

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

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

For example:

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

1.1.3 Signed Numbers

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

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

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

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

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

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

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

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

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

1.1.3.1 Converting between Positive and Negative

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

For example, inverting the number *four*:

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

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

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

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

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

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

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

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

1.1.4 Subtraction of Binary Numbers

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

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.

To calculate $-4 - 8 = -12$

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


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

1.1.5 Truncation and Overflow

Disscuss the details of truncation and overflow here.

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

1.1.6 Logical/Boolean Functions

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

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

➠ Fix Me:
Need to define 1 as true and 0 as false somewhere.

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

1.1.6.1 NOT

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

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

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

```
~ 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`

1.1.6.2 AND

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

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

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

```
  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`

1.1.6.3 OR

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

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

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

```
  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`

1.1.6.4 XOR

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

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

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

For example:

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

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

1.2 IEEE-754 Floating Point Number Representation

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

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

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

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

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

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

- We use scientific notation as in 2.7×10^{-47} to express either small fractions or large numbers when we are not concerned every last digit needed to represent the entire, exact, value of a number.
- The format of a number in scientific notation is $mantissa \times base^{exponent}$
- In binary we have $mantissa \times 2^{exponent}$
- IEEE–754 format requires binary numbers to be *normalized* to $1.significand \times 2^{exponent}$ where the *significand* is the portion of the *mantissa* that is to the right of the binary–point.
  - The unnormalized binary value of -2.625 is 10.101
  - The normalized value of -2.625 is 1.0101×2^1
- We need not store the '1.' because *all* normalized floating point numbers will start that way. Thus we can save memory when storing normalized values by adding 1 to the significand.

31	30 … 23	22 … 0
1	1 0 0 0 0 0 0 0	0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
sign	exponent	significand

- $-((1 + \frac{1}{4} + \frac{1}{16}) \times 2^{128-127}) = -((1 + \frac{1}{4} + \frac{1}{16}) \times 2^1) = -(2 + \frac{1}{2} + \frac{1}{8}) = -(2 + .5 + .125) = -2.625$
- IEEE754 formats:

	IEEE754 32–bit	IEEE754 64–bit
sign	1 bit	1 bit
exponent	8 bits (excess–127)	11 bits (excess-1023)
mantissa	23 bits	52 bits
max exponent	127	1023
min exponent	-126	-1022

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

➽ Fix Me:
Need to add the standard lecture numberline diagram showing where the over/under–flow areas are and why.

1.2.1 Floating Point Number Accuracy

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

1.2.1.1 Powers Of Two

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

Listing 1.1: `powersoftwo.c`
Precise Powers of Two

```c
#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 1.2: `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
```

1.2.1.2 Clean Decimal Numbers

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

Note how the decimal numbers are not accurately represented as they get larger. The decimal number on line 10 of Listing 1.4 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 1.3: `cleandecimal.c`
Print Clean Decimal Numbers

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

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

    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 1.4: `cleandecimal.out`
Output from `cleandecimal.c`

```
1              10.0000000000 = 41200000              -10.0000000000 = c1200000
2             100.0000000000 = 42c80000             -100.0000000000 = c2c80000
3            1000.0000000000 = 447a0000            -1000.0000000000 = c47a0000
4           10000.0000000000 = 461c4000           -10000.0000000000 = c61c4000
5          100000.0000000000 = 47c35000          -100000.0000000000 = c7c35000
6         1000000.0000000000 = 49742400         -1000000.0000000000 = c9742400
7        10000000.0000000000 = 4b189680        -10000000.0000000000 = cb189680
8       100000000.0000000000 = 4cbebc20       -100000000.0000000000 = ccbebc20
9      1000000000.0000000000 = 4e6e6b28      -1000000000.0000000000 = ce6e6b28
10    10000000000.0000000000 = 501502f9     -10000000000.0000000000 = d01502f9
11    99999997952.0000000000 = 51ba43b7     -99999997952.0000000000 = d1ba43b7
12   999999995904.0000000000 = 5368d4a5    -999999995904.0000000000 = d368d4a5
13  9999999827968.0000000000 = 551184e7   -9999999827968.0000000000 = d51184e7
```

1.2.1.3 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.

[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.

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 1.5: `erroraccumulation.c`
Accumulation of Error

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

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

    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 1.6: `erroraccumulation.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
0.7000000477 = 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
```

1.2.2 Reducing Error Accumulation

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

Listing 1.7: `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 1.8: `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
```

Chapter 2

The RISC-V GNU Toolchain

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

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

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

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

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

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

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

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

All other commands should be executed as a regular user. This will eliminate the possibility of clobbering system files that should not be touched when tinkering with the toolchain applicaitons.

To download, compile and "install" the toolchain:

```
# riscv toolchain:
#
# https://riscv.org/software-tools/risc-v-gnu-compiler-toolchain/

git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain
./configure --prefix=/usr/local/riscv --with-arch=rv32im --with-abi=ilp32
make
make install
```

Need to discuss augmenting the PATH environment variable.

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

Discuss rv32im and note that the details are found in chapter 3.

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

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

Source, object, and binary files

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

text, data, bss, stack

Labels and scope.

Forward & backward references to throw–away labels.

The entry address of an application.

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

Pre-processing conditional assembly using #if.

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

Linker scripts.

Makefiles

objdump

nm

hexdump -C

Chapter 3

RV32 Machine Instructions

3.1 Introduction

Fix Me:
Discuss what the IMAFD, G and other ISA extensions mean as well as the 32, 64 and 128–bit versions.

3.2 Conventions and Terminology

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

3.2.1 XLEN

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

3.2.2 sx(val)

Sign extend *val* to the left.

This is used to convert a signed integer value expressed using some number of bits to a larger number of bits by adding more bits to the left. In doing so, the sign will be preserved. In this case *val* represents the least MSBs of the value. For more on binary numbers see chapter 1.

Figure 3.1 illustrates extending the negative sign bit of *val* to the left by replicating it. When *val* is negative, its MSB (bit 19 in this example) will be set to 1. Extending this value to the left will set all the new bits to the left of it to 1 as well.

Figure 3.2 illustrates extending the positive sign bit of *val* to the left by replicating it. When *val* is positive, its MSB will be set to 0. Extending this value to the left will set all the new bits to the left of it to 0 as well.

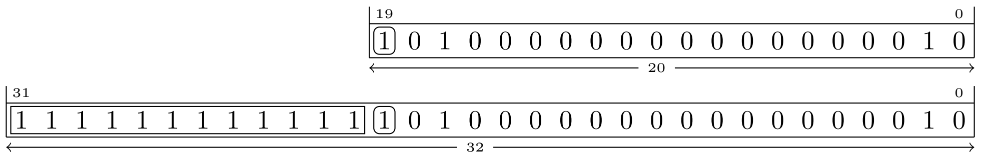


Figure 3.1: Sign–extending a negative integer from 20 bits to 32 bits.

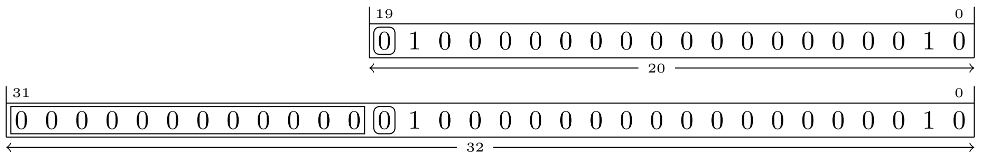


Figure 3.2: Sign–extending a positive integer from 20 bits to 32 bits.

3.2.3 zx(val)

Zero extend *val* to the left.

This is used to convert an unsigned integer value expressed using some number of bits to a larger number of bits by adding more bits to the left. In doing so, the new bits added will all be set to zero. As is the case with `sx(val)`, *val* represents the LSBs of the final value. Figure 3.3 illustrates zero–extending a 20–bit *val* to the left to form a 32–bit fullword.

For more on binary numbers see chapter 1.

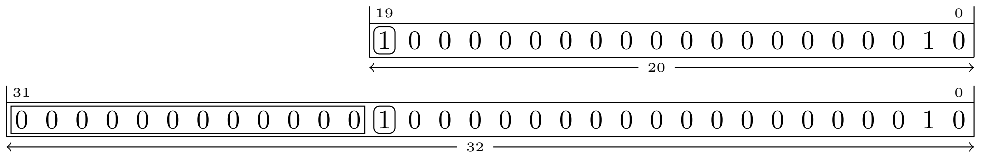


Figure 3.3: Zero–extending an unsigned integer from 20 bits to 32 bits.

3.2.4 zr(val)

Zero extend *val* to the right.

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

Figure 3.4 illustrates converting a 20–bit *val* to a 32–bit fullword.

3.2.5 Sign Entended Left and Zero Extend Right

Some instructions such as the J–type (see subsection 3.4.2) include immediate operands that are extended in both directions.

Figure 3.5 and Figure 3.6 illustrates zero–extending a 20–bit negative number one bit to the right and

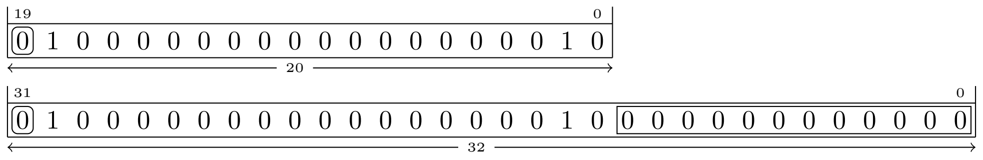


Figure 3.4: Zero–extending an integer to the right from 20 bits to 32 bits.

sign–extending it 11 bits to the left:

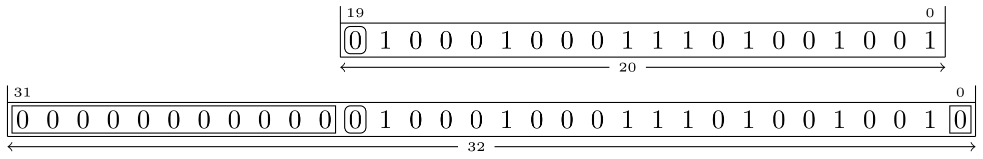


Figure 3.5: Sign–extending a positive 20–bit number 11 bits to the left and one bit to the right.

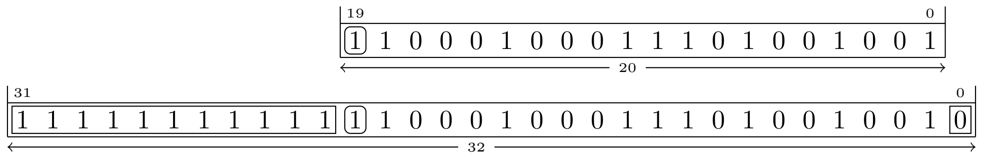


Figure 3.6: Sign–extending a negative 20–bit number 11 bits to the left and one bit to the right.

3.2.6 m8(addr)

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

Given the contents of the memory dump shown in Figure 3.7, `m8(42)` refers to the memory location at address `42`$_{16}$ that currently contains the 8–bit value `fc`$_{16}$.

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

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

`m8(addr)` ← `source`

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

`dest` ← `m8(addr)`

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

3.2.7 m16(addr)

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

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

Figure 3.7: Sample memory contents.

Given the contents of the memory dump shown in Figure 3.7, `m16(42)` refers to the memory location at address $\texttt{42}_{16}$ that currently contains $\texttt{65fc}_{16}$. See also subsection 3.2.6.

3.2.8 m32(addr)

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

Given the contents of the memory dump shown in Figure 3.7, `m32(42)` refers to the memory location at address $\texttt{42}_{16}$ that currently contains $\texttt{662065fc}_{16}$. See also subsection 3.2.6.

3.2.9 m64(addr)

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

Given the contents of the memory dump shown in Figure 3.7, `m64(42)` refers to the memory location at address $\texttt{42}_{16}$ that currently contains $\texttt{656c6c69662065fc}_{16}$. See also subsection 3.2.6.

3.2.10 m128(addr)

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

Given the contents of the memory dump shown in Figure 3.7, `m128(42)` refers to the memory location at address $\texttt{42}_{16}$ that currently contains $\texttt{7220687469772064656c6c69662065fc}_{16}$. See also subsection 3.2.6.

3.2.11 .+offset

The address of the current instruction plus a numeric offset.

3.2.12 .-offset

The address of the current instruction minus a numeric offset.

3.2.13 pc

The current value of the program counter.

3.2.14 rd

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

3.2.15 rs1

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

3.2.16 rs2

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

3.2.17 imm

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

3.2.18 rsN[h:l]

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

3.3 Addressing Modes

immediate, register, base-displacement, pc-relative

➽ Fix Me:
Write this section.

3.4 Instruction Encoding Formats

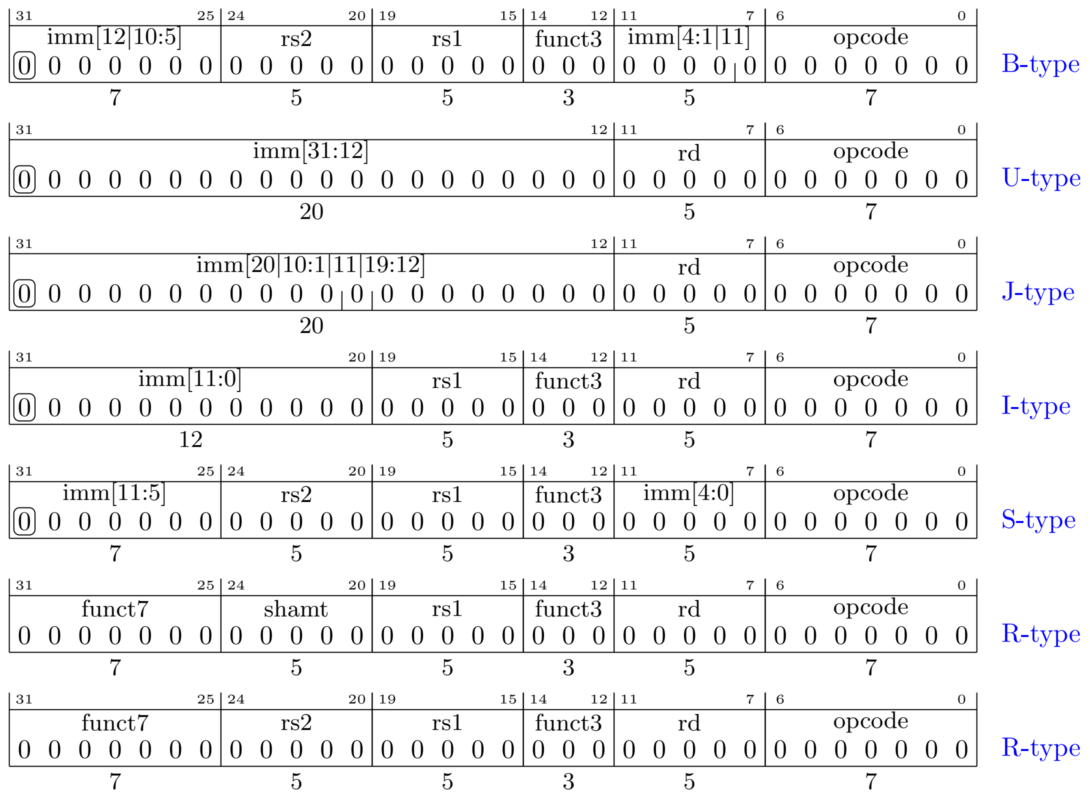
This document concerns itself with the following RISC-V instruction formats.

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

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

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

Figure 3.8 Shows the RISC-V instruction formats.

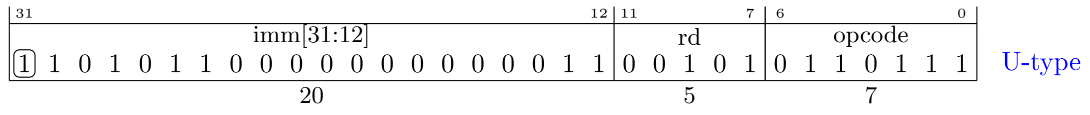


imm[12|10:5] rs2 rs1 funct3 imm[4:1|11] opcode B-type
imm[31:12] rd opcode U-type
imm[20|10:1|11|19:12] rd opcode J-type
imm[11:0] rs1 funct3 rd opcode I-type
imm[11:5] rs2 rs1 funct3 imm[4:0] opcode S-type
funct7 shamt rs1 funct3 rd opcode R-type
funct7 rs2 rs1 funct3 rd opcode R-type

Figure 3.8: RISC-V instruction formats.

3.4.1 U Type

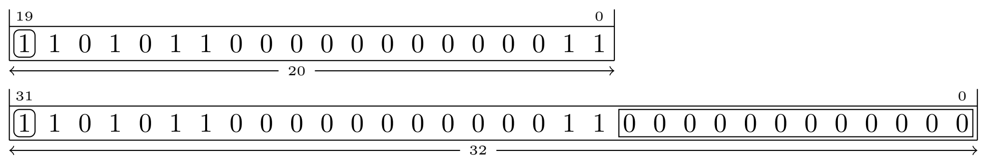
The U–Type format is used for instructions that use a 20–bit immediate operand and a destination register.



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

The imm field contains a 20–bit value that will be converted into XLEN bits by using the *imm* operand for bits 31:12 and then sign-extending it to the left[1] and zero–extending the LSBs as discussed in subsection 3.2.4.

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



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

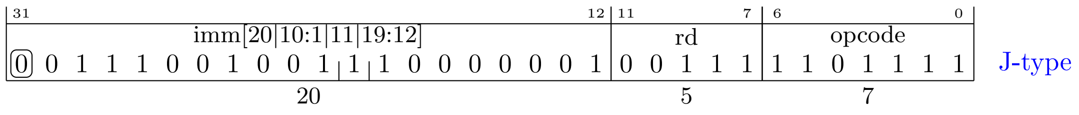
[1]When XLEN is larger than 32.

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

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

3.4.2 J Type

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



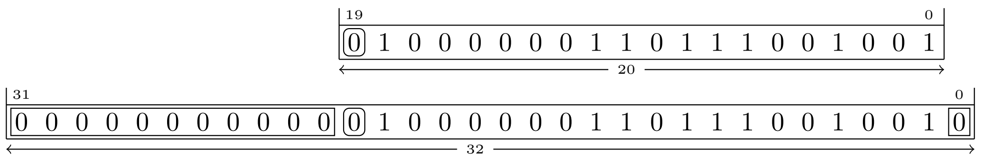
The rd field contains an x register number to be set to a value that depends on the instruction.

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

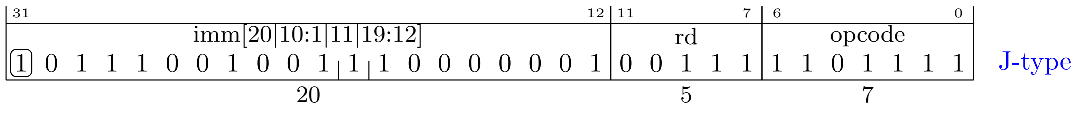
The example above shows that the bit positions in the *imm* field description. We see that the 20 *imm* bits are re–ordered according to: [20|10:1|11|19:12]. This means that the MSB of the *imm* field is to be placed into bit 20 of the immediate integer value ultimately used by the instruction when it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

After the *imm* bits are re-positioned into bits 20:1 of the immediate value being constructed, a zero–bit will be added to the LSB and the value in bit–position 20 will be replicated to sign–extend the value to XLEN bits as discussed in subsection 3.2.5.

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

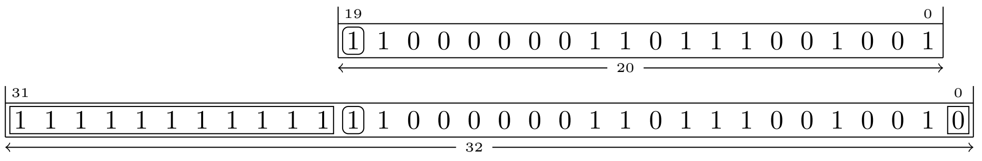


A J–type example with a negative imm field:



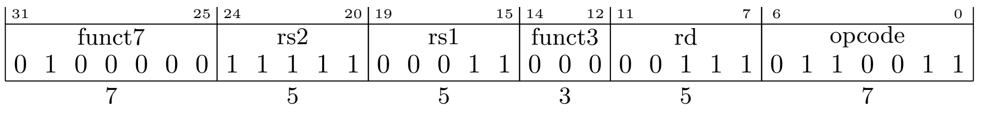
If XLEN=32 then the *imm* field in this example will be converted as shown below.

[2]The reason that the J–type bits are reordered like this is because it simplifies the implementation of hardware as discussed in section 3.4.



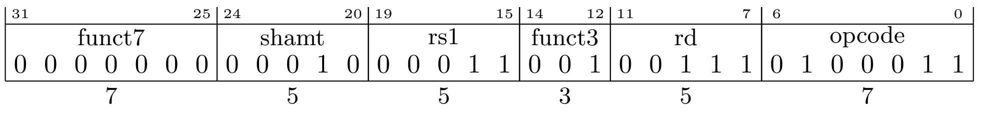
The J–type format is used by the Jump And Link instruction that calculates a target address by adding a signed immediate value to the current program counter. Since no instruction can be placed at an odd address the 20–bit imm value is zero–extended to the right to represent a 21-bit signed offset capable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

3.4.3 R Type



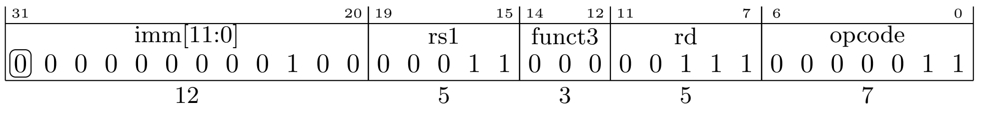
R-type

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



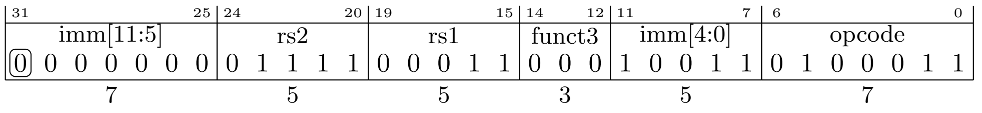
R-type

3.4.4 I Type



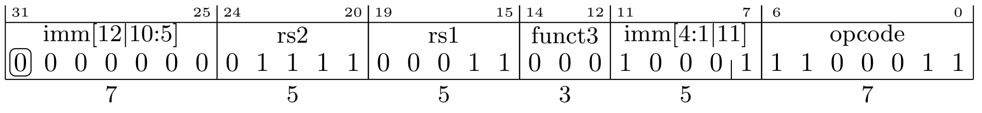
I-type

3.4.5 S Type



S-type

3.4.6 B Type



B-type

3.4.7 CPU Registers

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

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

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

3.5 memory

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

RV32I
Instruction!LUI
Instruction!AUIPC

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

3.6 RV32I Base Instruction Set

RV32I refers to the basic 32-bit integer instructions.

3.6.1 LUI rd, imm

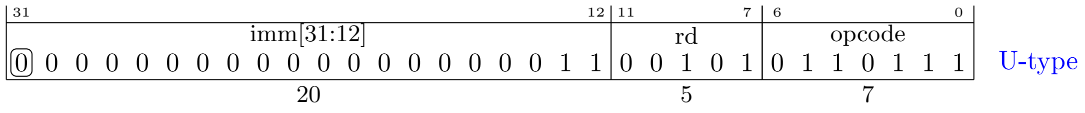
Load Upper Immediate.

`rd` ← `zr(imm)`

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

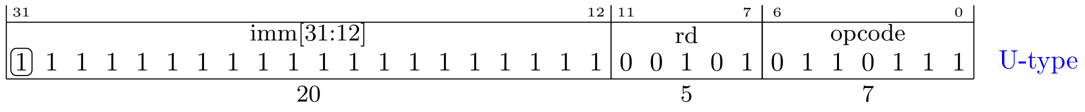
Instruction Format and Example:

LUI t0, 3
LUI t0, 3



t0 = 0x00000300

LUI t0, -1
LUI t0, 3



t0 = 0xfffff000

3.6.2 AUIPC rd, imm

Add Upper Immediate to PC.

`rd` ← `pc + zr(imm)`

Create a signed 32–bit value by zero–extending imm[31:12] to the right (see subsection 3.2.4) and add this value to the `pc` register, placing the result into `rd`.

Instruction!JAL

When XLEN is 64 or 128, the value placed into `rd` will be sign–extended to the left.

Encoding:

AUIPC t0, 3
AUIPC t0, 3

31 … 12	11 … 7	6 … 0	
imm[31:12]	rd	opcode	
$\boxed{0}$ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1	0 0 1 0 1	0 1 1 0 1 1 1	U-type
20	5	7	

State of registers before execution:

pc = 0x11114444

Decoding the instruction:

op-code = 0110111
rd = x5
imm = 0x00003000

After execution:

pc = 0x11114444
x5 = 0x11117444

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

3.6.3 JAL rd, imm

Jump and link.

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

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

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

Encoding:

JAL x7, .+16

31 … 12	11 … 7	6 … 0	
imm[20|10:1|11|19:12]	rd	opcode	
$\boxed{0}$ 0 0 0 0 0 0 1 0 0 0 | 0 | 0 0 0 0 0 0 0 0	0 0 1 1 1	1 1 0 1 1 1 1	J-type
20	5	7	

Instruction!JALR

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

State of registers before execution:

pc = 0x11114444

State of registers after execution:

pc = 0x11114454 x7 = 0x11114448

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

JAL x7, .-16

31 … 12	11 … 7	6 … 0	
imm[20|10:1|11|19:12]	rd	opcode	
$\boxed{1}$ 1 1 1 1 1 1 1 0 0 0 | 1 | 1 1 1 1 1 1 1 1	0 0 1 1 1	1 1 0 1 1 1 1	J-type
20	5	7	

imm demultiplexed value = $11111111111111111000_2 \ll 1 = -16_{10}$

State of registers before execution:

pc = 0x11114444

State of registers after execution:

pc = 0x11114434 x7 = 0x11114448

3.6.4 JALR rd, rs1, imm

Jump and link register.

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

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

Note that the branch target address is calculated by sign–extending the imm[11:0] bits from the instruction, adding it to the `rs1` register and *then* the LSB of the sum is to zero and the result is stored into the `pc` register. The discarding of the LSB allows the branch to refer to any even address.

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

Encoding:

JALR x1, x7, 4

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

Instruction!BEQ
Instruction!BNE

Before:

pc = 0x11114444
x7 = 0x44444444

After

pc = 0x5555888c
x1 = 0x11114448

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

JALR x1, x0, 5

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

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

pc = 0x11114444

After

pc = 0x00000004
x1 = 0x11114448

3.6.5 BEQ rs1, rs2, imm

Branch if equal.

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

Encoding:

BEQ x3, x15, 2064

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

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

3.6.6 BNE rs1, rs2, imm

Branch if Not Equal.

Instruction!BLT
Instruction!BGE

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

Encoding:

BNE x3, x15, 2064

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

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

3.6.7 BLT rs1, rs2, imm

Branch if Less Than.

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

Encoding:

BLT x3, x15, 2064

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

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

3.6.8 BGE rs1, rs2, imm

Branch if Greater or Equal.

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

Encoding:

BGE x3, x15, 2064

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

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

imm = 2064_{10}
funct3 = 101_2
rs1 = x3
rs2 = x15

Instruction!BLTU
Instruction!BGEU
Instruction!LB

3.6.9 BLTU rs1, rs2, imm

Branch if Less Than Unsigned.

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

Encoding:

BLTU x3, x15, 2064

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

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

3.6.10 BGEU rs1, rs2, imm

Branch if Greater or Equal Unsigned.

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

Encoding:

BGEU x3, x15, 2064

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

Fix Me:
use symbols in branch examples

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

3.6.11 LB rd, imm(rs1)

Load byte.

`rd ← sx(m8(rs1+sx(imm)))`

Instruction!LH
Instruction!LW

`pc` ← `pc+4`

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

Encoding:

LB x7, 4(x3)

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

3.6.12 LH rd, imm(rs1)

Load halfword.

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

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

Encoding:

LH x7, 4(x3)

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

3.6.13 LW rd, imm(rs1)

Load word.

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

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

Encoding:

LW x7, 4(x3)

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

3.6.14 LBU rd, imm(rs1)

Instruction!LBU
Instruction!LHU
Instruction!SB

Load byte unsigned.

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

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

Encoding:

LBU x7, 4(x3)

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

3.6.15 LHU rd, imm(rs1)

Load halfword unsigned.

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

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

Encoding:

LHU x7, 4(x3)

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

3.6.16 SB rs2, imm(rs1)

Store Byte.

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

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

Encoding:

SB x3, 19(x15)

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

Instruction!SH
Instruction!SW
Instruction!ADDI

3.6.17 SH rs2, imm(rs1)

Store Halfword.

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

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

Encoding:

SH x3, 19(x15)

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

3.6.18 SW rs2, imm(rs1)

Store Word

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

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

Encoding:

SW x3, 19(x15)

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

Show pos & neg imm examples.

3.6.19 ADDI rd, rs1, imm

Add Immediate

`rd` ← `rs1+sx(imm)`
`pc` ← `pc+4`

Encoding:

ADDI x1, x7, 4

Instruction!SLTI
Instruction!SLTIU

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

Before:

x7 = 0x11111111

After:

x1 = 0x11111115

3.6.20 SLTI rd, rs1, imm

Set LessThan Immediate

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

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

Encoding:

SLTI x1, x7, 4

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

Before:

x7 = 0x11111111

After:

x1 = 0x00000000

3.6.21 SLTIU rd, rs1, imm

Set LessThan Immediate Unsigned

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

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

Encoding:

SLTIU x1, x7, 4

Instruction!XORI
Instruction!ORI

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

Before:

x7 = 0x81111111

After:

x1 = 0x00000001

3.6.22 XORI rd, rs1, imm

Exclusive Or Immediate

`rd` ← `rs1 ^ sx(imm)`
`pc` ← `pc+4`

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

Encoding:

XORI x1, x7, 4

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

Before:

x7 = 0x81111111

After:

x1 = 0x81111115

3.6.23 ORI rd, rs1, imm

Or Immediate

`rd` ← `rs1 | sx(imm)`
`pc` ← `pc+4`

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

Encoding:

Instruction!ANDI
Instruction!SLLI

ORI x1, x7, 4

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

Before:

x7 = 0x81111111

After:

x1 = 0x81111115

3.6.24 ANDI rd, rs1, imm

And Immediate

`rd` ← `rs1 & sx(imm)`
`pc` ← `pc+4`

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

Encoding:

ANDI x1, x7, 4

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

Before:

x7 = 0x81111111

After:

x1 = 0x81111115

3.6.25 SLLI rd, rs1, shamt

Shift Left Logical Immediate

`rd` ← `rs1 << shamt`
`pc` ← `pc+4`

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

Encoding:

Instruction!SRLI
Instruction!SRAI

SLLI x7, x3, 2

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

x3 = 0x81111111

After:

x7 = 0x04444444

3.6.26 SRLI rd, rs1, shamt

Shift Right Logical Immediate

```
rd ← rs1 >> shamt
pc ← pc+4
```

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

Encoding:

SRLI x7, x3, 2

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

x3 = 0x81111111

After:

x7 = 0x20444444

3.6.27 SRAI rd, rs1, shamt

Shift Right Arithmetic Immediate

```
rd ← rs1 >> shamt
pc ← pc+4
```

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

Encoding:

SRAI x7, x3, 2

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

Instruction!ADD
Instruction!SUB

x3 = 0x81111111

After:

x7 = 0xe0444444

3.6.28 ADD rd, rs1, rs2

Add

`rd` ← `rs1 + rs2`
`pc` ← `pc+4`

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

Encoding:

ADD x7, x3, x31

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

x3 = 0x81111111 x31 = 0x22222222

After:

x7 = 0xa3333333

3.6.29 SUB rd, rs1, rs2

Subtract

`rd` ← `rs1 - rs2`
`pc` ← `pc+4`

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

Encoding:

SUB x7, x3, x31

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

x3 = 0x83333333 x31 = 0x01111111

Instruction!SLL
Instruction!SLT

After:

x7 = 0x82222222

3.6.30 SLL rd, rs1, rs2

Shift Left Logical

```
rd ← rs1 << rs2
pc ← pc+4
```

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

Encoding:

SLL x7, x3, x31

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

x3 = 0x83333333
x31 = 0x00000002

After:

x7 = 0x0ccccccc

3.6.31 SLT rd, rs1, rs2

Set Less Than

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

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

Encoding:

SLT x7, x3, x31

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

x3 = 0x83333333
x31 = 0x00000002

After:

x7 = 0x00000001

Instruction!SLTU
Instruction!XOR

3.6.32 SLTU rd, rs1, rs2

Set Less Than Unsigned

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

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

Encoding:

SLTU x7, x3, x31

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

x3 = 0x83333333
x31 = 0x00000002

After:

x7 = 0x00000000

3.6.33 XOR rd, rs1, rs2

Exclusive Or

`rd` ← `rs1 ^ rs2`
`pc` ← `pc+4`

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

Encoding:

XOR x7, x3, x31

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

x3 = 0x83333333
x31 = 0x1888ffff

After:

x7 = 0x9bbbcccc

3.6.34 SRL rd, rs1, rs2

Instruction!SRL
Instruction!SRA

Shift Right Logical

`rd` ← `rs1 >> rs2`
`pc` ← `pc+4`

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

Encoding:

SRL x7, x3, x31

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

x3 = 0x83333333
x31 = 0x00000010

After:

x7 = 0x00008333

3.6.35 SRA rd, rs1, rs2

Shift Right Arithmetic

`rd` ← `rs1 >> rs2`
`pc` ← `pc+4`

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

Encoding:

SLA x7, x3, x31

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

x3 = 0x83333333
x31 = 0x00000010

After:

x7 = 0xffff8333

3.6.36 OR rd, rs1, rs2

Instruction!OR
Instruction!AND

Or

`rd` ← `rs1 | rs2`
`pc` ← `pc+4`

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

Encoding:

OR x7, x3, x31

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

x3 = 0x83333333
x31 = 0x00000440

After:

x7 = 0x83333773

3.6.37 AND rd, rs1, rs2

And

`rd` ← `rs1 & rs2`
`pc` ← `pc+4`

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

Encoding:

AND x7, x3, x31

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

x3 = 0x83333333
x31 = 0x00000fe2

After:

x7 = 0x00000322

3.6.38 FENCE predecessor, successor

Instruction!FENCE
Instruction!FENCE.I
Instruction!ECALL

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

Operation:

`pc` ← `pc+4`

Encoding:

0 0 0 1 1 1 1 0 0 0 1 1 1 1 1 1 1

3.6.39 FENCE.I

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

Operation:

`pc` ← `pc+4`

Encoding:

0 0 0 1 1 1 1 0 0 1 0 0 0 0 0 0 0

3.6.40 ECALL

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

1 1 1 0 0 1 1 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

3.6.41 EBREAK

Instruction!EBREAK
Instruction!CSRRW
Instruction!CSRRS
Instruction!CSRRC

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

1 1 1 0 0 1 1 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

3.6.42 CSRRW rd, csr, rs1

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

1 1 1 0 0 1 1 0 0 0 1 1 0 0 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0

3.6.43 CSRRS rd, csr, rs1

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

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

1 1 1 0 0 1 1 0 0 0 1 1 0 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0

3.6.44 CSRRC rd, csr, rs1

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

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

1 1 1 0 0 1 1 0 0 0 1 1 0 1 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0

3.6.45 CSRRWI rd, csr, imm

Instruction!CSRRWI
Instruction!CSRRSI
Instruction!CSRRCI
RV32M
RV32A
RV32F
RV32D

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

1 1 1 0 0 1 1 0 0 0 1 1 1 0 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0

3.6.46 CSRRSI rd, csr, rs1

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

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

3.6.47 CSRRCI rd, csr, rs1

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

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

3.7 RV32M Standard Extension

32-bit integer multiply and divide instructions.

3.8 RV32A Standard Extension

32-bit atomic operations.

3.9 RV32F Standard Extension

32-bit IEEE floating point instructions.

3.10 RV32D Standard Extension

64-bit IEEE floating point instructions.

Appendix A

Attribution 4.0 International

Section 2. Scope

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.

Bibliography

[1] RISC-V Foundation, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*, 5 2017. Editors Andrew Waterman and Krste Asanović. vi, 25, 26, 27, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45

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

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

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

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

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

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

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

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

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

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

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

[13] D. Cohen, "IEN 137, On Holy Wars and a Plea for Peace," Apr. 1980. This note discusses the Big-Endian/Little-Endian byte/bit-order controversy, but did not settle it. A decade later, David V. James in "Multiplexed Buses: The Endian Wars Continue", *IEEE Micro*, **10**(3), 9–21 (1990) continued the discussion. 1

[14] R. M. Stallman and the GCC Developer Community, *Using the GNU Compiler Collection (For GCC version 7.3.0)*. Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA: GNU Press, 2017. 15

Index

Glossary

bit One binary digit. 2, 52

byte A binary value represented by 8 bits. 52

Doubleword A binary value represented by 64 bits. 52

Fullword A binary value represented by 32 bits. 52

Halfword A binary value represented by 16 bits. 52

High order bits Some number of MSBs. 52

hit One hex digit. 2, 3, 52

ISA Instruction Set Architecture. 52

LaTeX Is a mark up language specially suited for scientific documents. 52

Low order bits Some number of LSBs. 52

LSB Least Significant Bit. 4, 17, 20, 22, 52

MSB Most Significant Bit. 4, 5, 16, 17, 22, 52

overflow The situation where the result of an addition or subtraction operation is approaching positive or negative infinity and exceeds the number of bits alloted to contain the result. This is typically caused by high–order truncation. 5, 9, 52

Quadword A binary value represented by 128 bits. 52

RV32 Short for RISC-V 32. The number 32 refers to the XLEN. 25, 52

RV64 Short for RISC-V 64. The number 64 refers to the XLEN. 52

underflow The situation where the result of an addition or subtraction operation is approaching zero and exceeds the number of bits alloted to contain the result. This is typically caused by low–order truncation. 9, 52

XLEN The number of bits a RISC-V x integer register (such as x0). For RV32 XLEN=32, RV64 XLEN=64 etc. 21, 22, 52