APPENDIX **B**

Additional Topics in Binary Number Systems

B.1 INTRODUCTION

Chapter 1 introduced the concept of ***binary*** or base two numbers. The chapter showed how one could convert a decimal integer to binary through the ***addition method*** or the ***divide-by-two method***. However, numbers used in digital design may not always be whole numbers.

Consider a doctor who uses an in-ear digital thermometer that works in Celsius units to check whether a patient's body temperature is normal. We know that a human's normal body temperature is 37 degrees C (98.6 degrees F). If the thermometer's temperature sensor outputs integer values, then a readout of 37 C corresponds to an actual temperature anywhere between 36.5 C and 37.4 C, assuming the temperature sensor rounds its output to the nearest integer. Clearly, a more precise temperature readout is preferable to tell if a patient's temperature is abnormal. A readout of 37 C may mean that the patient has a normal body temperature or it may mean that the patient is close to having a fever. In order to be useful, the thermometer should output fractional components of the temperature so that the doctor can differentiate between 37.0 C and 37.9 C, for example.

This appendix discusses how *real* numbers (as opposed to just whole numbers) are represented in binary, and discusses methods that modern digital designers use to work with real numbers.

B.2 REAL NUMBER REPRESENTATION

Just as Chapter 1 looked closely at how integers are represented in decimal before moving on to binary numbers, understanding how real numbers are represented in decimal can illuminate how real numbers are represented in binary.

Chapter 1 showed that each digit in a number had a certain weight that was a power of 10. The ones place had a weight equal to $10^0 = 1$, the tens place had a weight equal to $10^1 = 10$, the hundreds place had a weight equal to $10^2 = 100$, and so on. If a decimal

number had an 8 in the hundred's place, a 6 in the ten's place, and a 0 in the one's place, the value of the number can be calculated by multiplying each digit by its weight and adding them together: $8*10^2 + 6*10^1 + 0*10^0 = 860$. This calculation is easy since people commonly manipulate decimal numbers.

The same concept of weights for each digit can be extended to the fractional components of the number. Consider the decimal number "923.501." The dot in the middle of the digits is the ***decimal point***. The decimal point separates the fractional component of the number from the whole part. While the weights of each digit in the whole part of the number are increasing powers of 10, the weights of the fractional digits are decreasing powers of 10, so the digits have fractional weights (e.g., $10^{-1}=0.1$ and $10^{-2}=0.01$). Therefore, the digits "923.501" represent $9*10^2 + 2*10^1 + 3*10^0 + 5*10^{-1} + 0*10^{-2} + 1*10^{-3}$, as shown in Figure B.1.

___	9	2	3 .	5	0	1	___
10^3	10^2	10^1	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-4}

Figure B.1 Representing real numbers in base 10.

Generally, the point used to separate the whole part of the number from the fractional part is called a ***radix point****, a term applicable to any base.*

We can represent real numbers in binary in a similar manner. Instead of a decimal point, real binary numbers feature a ***binary point***. Digits to the right of the binary point are weighted with negative powers of 2. For example, the binary number 10.1101 equals $1*2^1 + 0*2^0 + 1*2^{-1} + 1*2^{-2} + 0*2^{-3} + 1*2^{-4}$, or 2.8125 in decimal, as shown in Figure B.2.

___	1	0	. 1	1	0	1	___
2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}

Figure B.2 Representing real numbers in base 2.

You may be comfortable counting up by powers of two (1, 2, 4, 8, 16, 32, etc.). Counting down by powers of two may be difficult to memorize, but the numbers can be derived by dividing by 2: 1, 0.5, 0.25, and so on. Table B.1 illustrates this pattern.

Table B.1 Powers of two.

Power	Value
2^2	4
2^1	2
2^0	1
2^{-1}	0.5
2^{-2}	0.25
2^{-3}	0.125
2^{-4}	0.0625
2^{-5}	0.03125

The addition method used in Chapter 1 to convert decimal integers to binary is also a suitable method for converting real numbers, requiring no modifications other than needing to work with negative powers of two.

Example B.1 Converting real numbers from decimal to binary with the addition method

Convert the number 5.75 to binary using the addition method.

To perform this conversion, we follow the process described in Chapter 1. The conversion is detailed in Figure B.3.

	Binary	
Put 1 in highest place Place 8 too big, but 4 works, put 1 there.	1 0 0 . 0 0 4 2 1 0.5 0.25	*(current value: 4)*
1 in place 2 too big (4+2>5.75), put 0. 1 in place 1 is OK (4+1 < 5.75), put 1.	1 0 1 . 0 0 4 2 1 0.5 0.25	*(current value: 5)*
1 in place 0.5 is OK (5+0.5 < 5.75), put 1	1 0 1 . 1 0 4 2 1 0.5 0.25	*(current value: 5.5)*
1 in place 0.25 is OK (5.5+0.25 = 5.75) 5.75 reached, so done.	1 0 1 . 1 1 4 2 1 0.5 0.25	*(current value: 5.75)*

Figure B.3 Converting the decimal number 5.75 to binary using the addition method.

The alternative method in Chapter 1 for converting a decimal number to a binary number, namely the divide-by-2 method, can be adapted to work with decimal numbers. We first separate the whole part of the number from the fractional part, and perform the divide-by-2 method on the whole part by itself. Second, we take the fractional part of the number and *multiply* it by 2. After multiplying the fractional part by two, we append the digit in the one's place of the product after the binary point in the converted number. We continue multiplying the fractional part of the product and appending one's place digits until the fractional part of the product is 0.

For example, let's convert the decimal number 9.8125 to binary using the divide-by-2 method variant. First, we convert 9 to binary, which we know is `1001`. Next, we take the fractional part of the number, 0.8125, and multiply by 2: 0.8125*2 = 1.625. The one's digit is a 1, therefore we write a `1` after the binary point of the converted number: `1001.1`. Since the fractional part of the product is not 0, we continue multiplying the fractional part of the product by 2: 0.625*2 = 1.25. We append a `1` to the end of our converted number, giving us `1001.11`, and we continue multiplying by 2: 0.25*2 = 0.5. Now we append a `0` to the end of our converted number, yielding `1001.110`. We multiply by 2 again: 0.5*2 = 1.0. After appending the `1` to our converted number, we are left with `1001.1101`. Since the fractional part of the last product is 0, we are finished converting the number and thus obtain 9.8125_{10} = $\texttt{1001.1101}_2$.

A decimal real number can often require a very long sequence of bits after the binary point to represent the number in binary. In digital design, we are typically constrained to a finite number of bits available to store a number. As a result, the binary number may need to be truncated, and the binary number becomes an approximation.

▶ B.3 FIXED POINT ARITHMETIC

If we fix the binary point of a real number in a certain position in the number (e.g., after the 4th bit), we can add or subtract binary real numbers by treating the numbers as integers and adding or subtracting normally. The process is known as ***fixed point arithmetic***. In the resulting sum or difference, we maintain the binary point's position. For example, assume we are working with 8-bit numbers with half of the bits used to represent the fractional part of the number. Adding `1001.0010` (9.125) and `0011.1111` (3.9375) can be done simply by adding the two numbers as if they were integers. The sum, shown in Figure B.4, can be converted back to a real number by maintaining the binary point's position within the sum. Converting the sum to decimal verifies that the calculation was correct: $1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 + 0*2^{-1} + 0*2^{-2} + 0*2^{-3} + 1*2^{-4} = 8 + 4 + 1 + 0.0625 = 13.0625$.

```
    1 1 1   1 1
  1 0 0 1 . 0 0 1 0
+ 0 0 1 1 . 1 1 1 1
-----------------------
  1 1 0 1 . 0 0 0 1
```

Figure B.4 Adding two fixed point numbers.

Multiplying binary real numbers is also straightforward and does not require that the binary point be fixed. We first multiply the two numbers as if they were integers. Second, we place a binary point in the product such that the precision of the product is the sum of the precisions of the multiplicand and multiplier (the two numbers being multiplied), just like what is done when we multiply two decimal numbers together. Figure B.5 shows how we might multiply the binary numbers 01.10 (1.5) and 11.01 (3.25) using the partial product method described in Chapter 4. After calculating the product of the two numbers, we place a binary point in the appropriate location. Both the multiplier and multiplicand feature two bits of precision; therefore the product must have four bits of precision, and we insert a binary point to reflect this. Converting the product to decimal verifies that the calculation was correct: $0*2^3 + 1*2^2 + 0*2^1 + 0*2^0 + 1*2^{-1} + 1*2^{-2} + 1*2^{-3} + 0*2^{-4} = 4 + 0.5 + 0.25 + 0.125 = 4.875$.

```
        0 1.1 0
      × 1 1.0 1
-----------------
        0 1 1 0
      0 0 0 0
    0 1 1 0
+ 0 1 1 0
-----------------
0 1 0 0.1 1 1 0
```

Figure B.5 Multiplying two fixed point numbers.

The previous example was convenient in that we never had to add four `1`s together in a column when we summed up the partial products. To make the calculations simpler and to allow for the partial product summation to be implemented using full-adders, which can only add three `1`s at a time, we add the partial products incrementally instead of all at once. For example, let's multiply `1110.1` (14.5) by `0111.1` (7.5). As seen in Figure B.6, we begin by generating partial products as done earlier. However, we add partial products immediately into partial product sums, labeled *pps* in the figure. Eventually, we find that the product is `01101100.11`, which

```
    1 1 1 0.1     multiplicand
  × 0 1 1 1.1     multiplier
-----------------
    1 1 1 0 1     partial product 1 (pp1)
+ 1 1 1 0 1       pp2
-----------------
  1 0 1 0 1 1 1   pps1 = pp1 + pp2
+ 1 1 1 0 1       pp3
-----------------
  1 1 0 0 1 0 1 1 pps2 = pps1 + pp3
+ 1 1 1 0 1       pp4
-----------------
1 1 0 1 1 0 0 1 1 pps3 = pps2 + pp4
+ 0 0 0 0 0       pp5
-----------------
0 1 1 0 1 1 0 0.1 1 product = pps3 + pp5
```

Figure B.6 Multiplying two fixed point numbers using intermediate partial products.

corresponds to the correct answer of 108.75. You may want to try adding the five partial products together at once instead of using the intermediate partial product sums to see why this method is useful.

Before proceeding to binary real number division, we will introduce binary integer division, which was not discussed in previous chapters.

We can use the familiar process of long division to divide two binary integers. For example, consider the binary division of `101100` (44) by `10` (2). The full calculation is shown in Figure B.7. Notice how the procedure is exactly the same as decimal long division except that the numbers are now in binary.

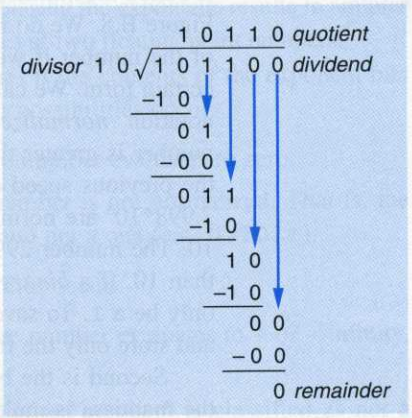

Figure B.7 Dividing two binary integers using long division.

Dividing binary real numbers, like multiplication, also does not require that the binary point be fixed. However, to simplify the calculation, we shift both the dividend and divisor's binary points right until the divisor no longer has a fractional part. For example, consider the division of 1.01_2 (1.25) by 0.1_2 (0.5). The divisor, 0.1_2, has one digit in its fractional part; therefore we shift the dividend and divisor's binary points right by one digit, changing our problem to 10.1_2 divided by 1_2. We now treat the numbers as integers (ignoring the binary point) and can divide them using the long division approach. Trivially, $101_2/1_2$ is 101_2. We then restore the binary point to where it was in the dividend, giving us the answer 10.1_2 or 2.5.

Why does shifting the binary point not change the answer? In general, shifting the radix point right by one digit is the same as multiplying the number by its base. For binary numbers, shifting the binary point right is equivalent to multiplying the number by 2. Dividing two numbers will give you the ratio of the two numbers to each other. Multiplying the two numbers by the same number (by means of shifting the binary point) will not affect that ratio, since doing so is equivalent to multiplying the ratio by 1.

Fixed point numbers are simple to work with, but are limited in the range of numbers that they can represent. For a fixed number of bits, increasing the precision of a number comes at the expense of the range of whole numbers that we can use, and vice versa. Fixed point numbers are suitable for a variety of applications, such as a digital thermometer, but more demanding applications need greater flexibility and range in their real number representations.

▶ B.4 FLOATING POINT REPRESENTATION

When working with decimal numbers, we often represent very large or very small numbers by using scientific notation. Rather than writing a googol as a 1 with a hundred 0s after it, we could write $1.0*10^{100}$. Instead of 299,792,458 m/s, we could write the speed of light as $3.0*10^8$ m/s, as $2.998*10^8$, or even $299.8*10^6$.

If such notation could be translated into binary, we would be able to store a much greater range of numbers than if the binary point were fixed. What features of this notation need to be captured in a binary representation?

First is the whole and fractional part of the number being multiplied by a power of 10, which is called the ***mantissa*** (or ***significand***), as shown in Figure B.8. We do not need to store the whole part of the number if we make sure the number is in a certain form. We call a number written in scientific notation ***normalized*** if the whole part of the number is greater than 0 but less than the base. In the previous speed of light examples, $3.0*10^8$ and $2.998*10^8$ are normalized since 3 and 2, respectively, are greater than zero but less than 10. The number $299.8*10^6$, on the other hand, is not normalized because 299 is greater than 10. If a *binary* real number is normalized, then the whole part of the mantissa can only be a `1`. To save bits, we can safely assume that the whole part of the mantissa is 1 and store only the fractional part.

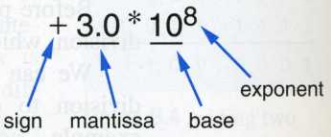

Figure B.8 Parts of a number in scientific notation.

Second is the base (sometimes referred to as the ***radix***) and the exponent by which the mantissa is multiplied, shown in Figure B.8. Calling 10 the base is no accident—the number is the same as the base of the entire number. In binary, the base is naturally 2. Knowing this, we do not need to store the 2. We can simply assume that 2 is the base and store the exponent.

Third, we must capture the sign of the number.

The IEEE 754-1985 Standard

The Institute of Electrical and Electronic Engineers (IEEE) 754-1985 standard specifies a way in which the three values described above can be represented in a 32-bit or a 64-bit binary number, referred to as single and double precision, respectively. Though there are other ways to represent real numbers, the IEEE standard is by far the most widely used. We refer to these numbers as ***floating point*** numbers.

The IEEE standard assigns a certain range of bits for each of the three values. For 32-bit numbers, the first—most significant—bit specifies the sign, followed by 8 bits for the exponent, and the remaining 23 bits are used for the mantissa. This arrangement is pictured in Figure B.9.

Figure B.9 Bit arrangement in a 32-bit floating point number.

The sign bit is set to 0 if the number is positive, and the bit is set to 1 if the number is negative. The mantissa bits are set to the fractional part of the mantissa in the original number. For example, if the mantissa is `1.1011`, we would store `1011` followed by 19 zeroes in bits 22 to 0. As part of the standard, we add 127 to the exponent we store in the exponent bits. Therefore, if a floating point number's exponent is 3, we would store 130 in

the exponent bits. If the exponent was –30, we would store 97 in the exponent bits. The adjusted number is called a ***biased*** exponent. Exponent bits containing all `0`s or all `1`s have special meanings and cannot be used. Under these conditions, the range of biased exponents we can write in the exponent bits is 1 to 254, meaning the range of unbiased exponents is –126 to 127. Why don't we simply store the exponent as a signed, two's complement number (discussed in Section 4.8)? Because it turns out that biasing the exponents results in simpler circuitry for comparing the magnitude (absolute value) of two floating point numbers.

The IEEE standard defines certain special values if the contents of the exponent bits are uniform. When the exponent bits are all `0`s, two possibilities occur:

1. If the mantissa bits are all `0`s, then the entire number evaluates to zero.
2. If the mantissa bits are nonzero, then the number is not normalized. That is, the whole part of the mantissa is a binary zero and not a one (e.g. `0.1011`).

When the exponent bits are all `1`s, two possibilites occur:

1. If the mantissa bits are all `0`s, then the entire number evaluates to + or – infinity, depending on the sign bit.
2. If the mantissa bits are nonzero, then the entire "number" is classified as not a number (NaN).

There are also specific classes of NaNs, beyond the scope of this appendix, that are used in computations involving NaNs.

With this information, we can convert decimal real numbers to floating point numbers. Assuming the decimal number to be converted is not a special value in floating point notation, Table B.2 describes how to perform the conversion.

Table B.2 Method for converting real decimal numbers to floating point

	Step	Description
1	*Convert the number from base 10 to base 2.*	Use the method described in Section B.2.
2	*Convert the number to normalized scientific notation.*	Initially multiply the number by 2^0. Adjust the binary point and exponent so that the whole part of the number is 1_2.
3	*Fill in the bit fields.*	Set the sign, *biased* exponent, and mantissa bits appropriately.

Example B.2 Converting decimal real numbers to floating point

Convert the following numbers from decimal to IEEE 754 32-bit floating point: 9.5, infinity, and $-52406.25 * 10^{-2}$.

Let's follow the procedure in Table B.2 to convert 9.5 to floating point. In Step 1, we convert 9.5 to binary. Using the subtraction method, we find that 9.5 is `1001.1` in binary. To convert the number to scientific notation per Step 2, we multiply the number by 2^0, giving `1001.1` $* 2^0$ (for readability purposes, we write the 2^0 part in base 10). To normalize the number, we must shift the

binary point left by three digits. In order not to change the value of the number after moving the binary point, we change the 2's exponent to 3. After Step 2, our number becomes `1.0011` * 2^3.

In Step 3, we put everything together into the properly formatted sequence of bits. The sign bit is set to 0, indicating a positive number. The exponent bits are set to 3 + 127 = 130 (we must bias the exponent) in binary, and the mantissa bits are set to 0011_2, which is the fractional part of the mantissa. Remember that the `1` to the left of the binary point is implied since the number is normalized. The properly encoded number is shown in Figure B.10.

Now let's convert infinity to a floating point number. Since infinity is a special value, we cannot employ the method we used to convert 9.5 to floating point. Rather, we fill in the three bit fields with special values indicating that the number is infinity. From the discussion of special values above, we know that the exponent bits should be all 1s and the mantissa bits should be all 0s. The sign bit should be 0 since infinity is positive. Therefore, the equivalent floating point number is `0 11111111 00000000000000000000000`.

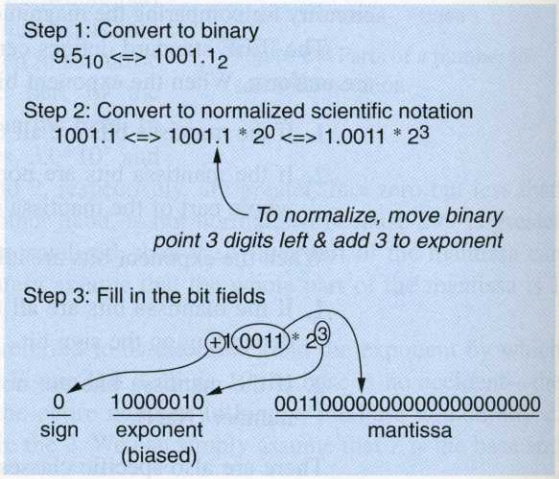

Figure B.10 Representing 9.5 as a 32-bit floating point number, most significant bit first.

Converting –52406.25 * 10^{-2} to floating point is straightforward using the method in Table B.2. For Step 1, we convert the number to binary. Recall that we represent the sign of the number using a single bit and not using two's complement representation, so we only need to convert 52406.25 * 10^{-2} to binary and set the sign bit to indicate that the number is negative. The number 52406.25 * 10^{-2} evaluates to 524.0625. Using the subtraction or divide-by-2 method we know that 524 is `1000001100` in binary. The fractional part, 0.0625, is conveniently 2^{-4}. Thus 524.0625 is `1000001100.0001` in binary. In Step 2, we write the number in scientific notation: `1000001100.0001` * 2^0. We must also normalize the number by shifting the binary point left by 9 digits and compensating for this shift in the exponent: `1.0000011000001` * 2^9. Finally, we combine the sign (1 since the original number is negative), biased exponent (9 + 127 = 136), and fractional part of the mantissa into a floating point number: `1 10001000 00000110000010000000000`.

Example B.3 Converting floating point numbers to decimal

Convert the number `11001011101010100000000000000000` from IEEE 754 32-bit floating point to decimal.

To perform this conversion, we first split the number into its sign, exponent, and mantissa parts: `1 10010111 01010100000000000000000`. We can immediately see from the sign bit that the number is negative.

Next, we convert the 8-bit exponent and 23-bit mantissa from binary to decimal. We find that 10010111 is 151. We unbias the exponent by subtracting 127 from 151, giving an unbiased exponent of 24. Recall that the mantissa in the pattern of bits represents the fractional part of the

mantissa and is stored without the leading 1 from the whole part of the mantissa (assuming the original number was normalized). Restoring the 1 and adding a binary point gives us the number 1.01010100000000000000000, which is the same number as 1.010101. By applying weights to each digit, we see that $1.010101 = 1*2^0 + 0*2^{-1} + 1*2^{-2} + 0*2^{-3} + 1*2^{-4} + 0*2^{-5} + 1*2^{-6} = 1.328125$.

With the original sign, exponent, and mantissa extracted, we can combine them into a single number: $-1.327125 * 2^{24}$. We can multiply the number out to –22,265,462.784, which is equivalent to $-2.2265462784 * 10^7$.

The format for double precision (64-bit) floating point numbers is similar, with three fields having a defined number of bits. The first, most significant bit represents the sign of the number. The next 11 bits hold the biased exponent, and the remaining 52 bits hold the fractional part of the mantissa. Additionally, we add 1023 to the exponent instead of 127 to form the biased exponent. This arrangement is pictured in Figure B.11.

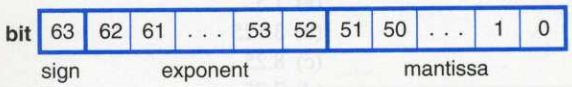

Figure B.11 Bit arrangement in a 64-bit floating point number.

Floating Point Arithmetic

Floating point arithmetic is beyond the scope of this text, but we will provide a brief overview of the concept.

Floating point addition and subtraction must be performed by first *aligning* the two floating point numbers so that their exponents are equal. For example, consider adding the two decimal numbers $2.52*10^2 + 1.44*10^4$. Since the exponents differ, we can change $2.52*10^2$ to $0.0252*10^4$. Adding $0.0252*10^4$ and $1.44*10^4$ gives us the answer $1.4652*10^4$. Similarly, we could have changed $1.44*10^4$ to $144*10^2$. Adding $144*10^2$ and $2.52*10^2$ gives us the sum $146.52*10^2$, which is the same number as our first set of calculations. An analogous situation occurs when we work with floating point numbers. Typically, hardware that performs floating point arithmetic, often referred to as a ***floating point unit***, will adjust the mantissa of the number with the smaller exponent before adding or subtracting the mantissas (with their implied 1s restored) together and preserving the common exponent. Notice that before the addition or subtraction is performed, the exponents of the two numbers are compared. This comparison is facilitated through the use of the sign bit and the biased exponent as opposed to representing the exponent in two's complement form.

Multiplication and division in floating point require no such alignments. Like in decimal multiplication and division of numbers in scientific notation, we multiply or divide the mantissas and add or subtract the two exponents, depending on the operation. When multiplying, we add exponents. For example, let's multiply $6.44*10^7$ by $5.0*10^{-3}$. Instead of trying to multiply 64,400,000 by 0.005, we multiply the two mantissas together and add the exponents. 6.44*5.0 is 32.2 and 7+(–3) is 4. Thus the answer is $32.2*10^4$. When dividing, we subtract the exponent of the divisor from the dividend's exponent. For example, let's divide $31.5*10^{-4}$ (dividend) by $2.0*10^{-12}$ (divisor). Dividing 31.5 by 2.0 gives us 15.75. Subtracting the divisor's exponent from the dividend's gives

$-4 - (-12)=8$. Thus the answer is $15.75*10^8$. Floating point division defines results for several boundary cases such as dividing by 0, which evaluates to positive or negative infinity, depending on the sign of the dividend. Dividing a nonzero number by infinity is defined as 0, otherwise dividing by infinity is NaN.

▶ B.5 EXERCISES

SECTION B.1: REAL NUMBER REPRESENTATION

B.1 Convert the following numbers from decimal to binary:
(a) 1.5
(b) 3.125
(c) 8.25
(d) 7.75

B.2 Convert the following numbers from decimal to binary:
(a) 9.375
(b) 2.4375
(c) 5.65625
(d) 15.5703125

SECTION B.3: FIXED POINT ARITHMETIC

B.3 Add the following two unsigned binary numbers using binary addition and convert the result to decimal:
(a) `10111.001 + 1010.110`
(b) `01101.100 + 10100.101`
(c) `10110.1 + 110.011`
(d) `1101.111 + 10011.0111`

SECTION B.4: FLOATING POINT REPRESENTATION

B.4 Convert the following decimal numbers to 32-bit floating point:
(a) –50,208
(b) $42.427523 * 10^3$
(c) $-24{,}551{,}152 * 10^{-4}$
(d) 0

B.5 Convert the following 32-bit floating point numbers to decimal:
(a) `01001100010110110101100001011000`
(b) `01001100010110110101001000000000`
(c) `01111111111000110000000000000000`
(d) `01001101000110101000101000000000`