ECOR 1041
Computation and Programming

Binary Representation of Numbers

Carleton University

References

- Binary Number System: https://www.mathsisfun.com/binary-number-system.html

- *The Python Tutorial: Floating Point Arithmetic: Issues and Limitations*
- https://docs.python.org/3/tutorial/floatingpoint.html

Lecture Objectives

- Review Modulus
- Understand how computers represent unsigned integers, signed integers, and floating-point numbers as binary numbers
- Understand some of the limitations of computation using floating-point numbers

Review Modulus

Section Objectives

- Review how modulus works for positive and negative integers
- Briefly discuss how modulus works for floating point numbers

Learning Outcomes

- Know how to calculate a % b when both are positive

- Know how to calculate a % b when one or both is negative

Modulus

- Modulus gives us the remainder when dividing one number, m, by another number, n: i.e. m % n
  - We will assume that m and n are integers (for now).
  - If the number, n, is **positive**, the possible remainders are 0, 1, 2, …, n - 1
  - If the number, n, is **negative**, the possible remainder are 0, -1, -2, …, n + 1

Modulus (continued)

For example:

- If we have m % 3, the remainder will be 0, 1, or 2.
- If we have m % -3, the remainder will be 0, -1, or -2.
- Mathematically, we want to find two numbers, x, and y, so that:
  - m = x * n + y, where y is the remainder (in the valid range)
  - once we satisfy this equation, x will be m // n, and y will be m % n

Modulus (continued)

- How to find modulus with an example: % 3 (always 0, 1, or 2):
- m positive:

m	0	1	2	3	4	5	6
m%3	0	1	2	0	1	2	0

- What if m is negative?:

m	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
m%3	0	1	2	0	1	2	0	1	2	0	1	2	0

- Note that if m % n is not 0, then -m % n is n - m % n
  - e.g. -5 % 3 = 3 - 5 % 3 = 3 - 2 = 1

Carleton University

Modulus (continued)

- What about: % -3 (always 0, -1, or -2):

m	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
m%-3	0	-2	-1	0	-2	-1	0	-2	-1	0	-2	-1	0

- Note that -m % -n = - (m % n)
  - e.g. -5 % -3 = - (5 % 3) = -2

Integer Division

- Calculate using regular division.
- If it is not a round number, then go to the next lowest number. This is equivalent to truncating for positive numbers, but **not** for negative.
- Example dividing by 3:

m	**-6**	**-5**	**-4**	**-3**	**-2**	**-1**	**0**	**1**	**2**	**3**	**4**	**5**	**6**
m/3	-2	-1.67	-1.33	-1	-0.67	-0.33	0	0.33	0.67	1	1.33	1.67	2
m//3	-2	-2	-2	-1	-1	-1	0	0	0	1	1	1	2

Integer Division (continue)

- Example dividing by -3:

m	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
m/-3	2	1.67	1.33	1	0.67	0.33	0	-0.33	-0.67	-1	-1.33	-1.67	-2
m//-3	2	1	1	1	0	0	0	-1	-1	-1	-2	-2	-2

Modulus (continued)

- What if m or n (or both) is a floating point number?
  - The answer will be a floating point number.
  - If either ends with .0, then do the calculation as above, and add .0 on the end of the answer (i.e. the answer is a float)
  - Otherwise, the same rules apply, you just round to the number of decimal places in the original numbers.
    - Do not worry about this ☺ (beyond the scope of the course).

Number Systems

Representing Unsigned Integers in Binary

Representing Signed Integers in Binary

Section Objective

- Understand how computers represent unsigned integers, signed integers, and floating-point numbers as binary numbers

Learning Outcomes (Vocabulary)

- Know the meaning of these words
  - Binary number system
  - Bit
  - Base 10, base 2
  - Signed-magnitude

Learning Outcomes: Needed for ECOR 1044

- Convert unsigned and signed-magnitude binary integers to decimal integers
- Convert decimal integers to unsigned and signed-magnitude binary integers
- Note: There are other (better) ways to represent negative numbers in binary, but those will be covered later in your program, if needed.

Number Systems

Decimal (Base 10) Number System

- How do we interpret the decimal integer 742?
- "7 hundreds, 4 tens and 2 ones"
- $742 \Rightarrow 7 \times 10^2 + 4 \times 10^1 + 2 \times 10^0$
  - Weights associated with the digit positions are powers of 10
- Base 10 uses 10 symbols to represent the decimal digits: 0 1 2 3 4 5 6 7 8 9

Binary (Base 2) Number System

- Decimal is convenient for humans (10 fingers/thumbs)
- Digital circuits in computers represent information using the binary (base 2) number system
- Base 2 has two binary digits (*bits*)
  - In hardware, a bit can be represented by a transistor switch that is either on or off
  - When we do binary math, bits are usually represented by the two symbols, 0 and 1

Number Systems: Subscript Notation

- To clearly specify which number system we are using, we can write the base as a subscript after the number
  - $\textit{digits}_{\textit{base}}$
- 11_{10} means 11 base 10 (decimal)
- 1101_2 means 1101 base 2 (binary)

Adding Binary Numbers

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 0$ carry 1; i.e, 10_2

Representing Unsigned Integers in Binary

Data Storage

- In a computer, data are stored in “words” containing a fixed number of binary digits
- If a number requires fewer bits than the word size, it is padded with leading 0’s
- 11010_2 is stored as:
  - 00011010 in an 8-bit word (a *byte*)
  - 0000000000011010 in a 16-bit word

Unsigned Binary Integers

- All bits contribute to the integer's magnitude
- The smallest integer has all bits equal to 0
  - $00000000_2 = 0_{10}$
- The largest integer has all bits equal to 1
  - If the integer has *k* bits, its decimal value is $2^k - 1$
  - 8-bit integer: $11111111_2 = 2^8 - 1 = 255_{10}$
  - 16-bit integer: $1111111111111111_2 = 2^{16} - 1 = 65{,}535_{10}$

Binary to Decimal Conversion

- To convert an unsigned integer from its binary representation to decimal, multiply each digit by its weight, then sum the results
  - Weights are powers of 2
- $d_3d_2d_1d_0$ (binary)
  $= d_3 \times 2^3 + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$ (decimal)

Binary to Decimal Conversion Example

- $1001011001_2 = ?_{10}$

- 1001011001_2

  $= 1 \times 2^9 + 0 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 +$
  $1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
  $= 2^9 + 2^6 + 2^4 + 2^3 + 2^0$
  $= 512_{10} + 64_{10} + 16_{10} + 8_{10} + 1_{10}$
  $= 601_{10}$

Carleton University

Decimal to Binary Conversion Example

- $598_{10} = ?_2$

- What is the largest power of two that is smaller or equal to the number?: $512 = 2^9$
- Thus, we will have a 1 in the 2^9 position in the binary number (i.e., 10^{th} digit from the right).
- Subtract 512 from the original number and we have 86.
- Repeat the above (see next slide).

Decimal to Binary Conversion Example

- Repeat the above:
  - $64 = 2^6$; $86 - 64 = 22$;
  - $16 = 2^4$; $22 - 16 = 6$;
  - $4 = 2^2$; $6 - 4 = 2$;
  - $2 = 2^1$; $2 - 2 = 0$
- Thus, we will have 1s at 2^9, 2^6, 2^4, 2^2, and 2^1 in our binary representation, and the other digits will be 0.
- So, $598_{10} = 1001010110_2$

Decimal to Binary Conversion Example: Alternate Solution

- $598_{10} = ?_2$

- Repeatedly divide by 2:
- 598/2 = 299 with remainder 0
- 299/2 = 149 with remainder 1
- 149/2 = 74 with remainder 1
- 74/2 = 37 with remainder 0
- 37/2 = 18 with remainder 1 (continued)

Decimal to Binary Conversion Example: Alternate Solution

- 18/2 = 9 with remainder 0
- 9/2 = 4 with remainder 1
- 4/2 = 2 with remainder 0
- 2/2 = 1 with remainder 0
- 1/2 = 0 with remainder 1

- Write out the remainders from bottom to top to get your binary number: $598_{10} = 1001010110_2$

Carleton University

Decimal to Binary Conversion Alternate Solution Explained using % 2 and // 2

- Divide-by-2 algorithm
  - *n* is a positive integer
  - Step 1: calculate the remainder when *n* is divided by 2
    - this is 0 or 1 (a bit)
  - Step 2: write the remainder to the left any previous remainders
  - Step 3: replace *n* by *n* // 2 (integer division)
- Repeat steps until *n* becomes 0

Decimal to Binary Conversion Alternate Solution Explained using % 2 and // 2

- Use mathematical notation to express the algorithm concisely
  - $n \leftarrow$ a positive integer
  - $s \leftarrow$ an empty sequence of bits
  - Repeat steps until n becomes 0
    - Step 1: $bit \leftarrow n \;\%\; 2$
    - Step 2: prepend *bit* to *s*
    - Step 3: $n \leftarrow n \;//\; 2$

Decimal to Binary Conversion Example Alternate Solution Explained using % 2 and // 2

- $23_{10} = ?_2$
  - $bit \leftarrow 23 \% 2 = 1$
  - s becomes 1
  - $n \leftarrow 23 // 2 = 11$
  - $bit \leftarrow 11 \% 2 = 1$
  - s becomes 11
  - $n \leftarrow 11 // 2 = 5$
  - $bit \leftarrow 5 \% 2 = 1$
  - s becomes 111
- $n \leftarrow 5 // 2 = 2$
- $bit \leftarrow 2 \% 2 = 0$
- s becomes 0111
- $n \leftarrow 2 // 2 = 1$
- $bit \leftarrow 1 \% 2 = 1$
- s becomes 10111
- $n \leftarrow 1 // 2 = 0$
- So, $23_{10} = 10111_2$

Representing Signed Integers in Binary

Signed Binary Integers

- How do we represent signed (positive and negative) integers in binary if we only have the symbols 0 and 1 (no minus sign)?

Signed Magnitude

- Use the most significant bit as the sign bit
  - 0 $\Rightarrow$ positive
  - 1 $\Rightarrow$ negative
- Use the remaining bits to represent the integer’s magnitude

Signed Magnitude Examples (8-bit Numbers)

- $00000000_2 \Rightarrow +0_{10}$
- $00000001_2 \Rightarrow +1_{10}$
- $01111111_2 \Rightarrow +127_{10}$
- $10000000_2 \Rightarrow -0_{10}$
- $10000001_2 \Rightarrow -1_{10}$
- $11111111_2 \Rightarrow -127_{10}$

Signed Magnitude: Summary

- A k-bit binary number, interpreted as a signed magnitude integer, can represent 2^k decimal integers ranging from $-(2^{k-1} - 1)$ to $2^{k-1} - 1$
- Example: a 16-bit binary number can represent signed magnitude integers ranging from -32,767 to 32,767
- Drawbacks
  - 2 representations for 0
  - Circuits to perform arithmetic are relatively complex

Signed Magnitude Conversion to/from Decimal

- The only difference between signed-magnitude and unsigned binary integers is the sign bit.

- To convert from signed-magnitude to/from decimal, start by ignoring the sign bit or negative sign and perform the conversion as for an unsigned integer to/from decimal.

- Then… (see next slide)

Signed Magnitude Conversion to/from Decimal

- Then:

- If converting a signed-magnitude binary integer to decimal, if the sign bit is one, add a negative sign to the decimal number. (Otherwise, the decimal number is positive.)

- If converting a decimal number to signed-magnitude, if the decimal number is negative add a one in the leftmost bit of the signed-magnitude binary number. (Otherwise, the leftmost bit is zero.)

Representing Fractions in Binary, Limitations of Floating-Point Arithmetic

Carleton University

Section Objective

- Understand some of the limitations of computation using floating-point numbers

Learning Outcomes (Vocabulary)

- Know the meaning of these words
  - Floating-point number
  - Binary fraction

Learning Outcomes

- Convert a binary fraction into a decimal fraction
- Convert a decimal fraction to a binary fraction
- Explain why an arithmetic operation on floating-point numbers may yield a result that is only approximately equal to the same operation performed on real numbers

What are Floating-Point Numbers?

- Most programming languages use floating point numbers to represent real numbers
- A floating-point number is a number that is representable in a floating-point format
- IEEE 754-2008 (the most commonly implemented standard for floating-point arithmetic) defines 5 basic formats for floating-point numbers

What are Floating Point Numbers?

- In the 64-bit binary format (which is used by Python), the floating-point representation of a non-zero, finite real number has the form:

$$(-1)^{s} \times 2^{e} \times m$$

  - s is 0 or 1 (denotes the number's sign)
  - Exponent e
  - Significand m has 53 binary digits of precision

What are Floating Point Numbers?

- The details of this format are beyond the scope of the course, but if you are interested, here is a double-precision (64-bit) floating point to binary (and vice versa) conversion tool: https://www.binaryconvert.com/result_double.html

Floating-Point Arithmetic

- Calculate 0.1 × 10 by repeated addition:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + \
... 0.1 + 0.1 + 0.1 + 0.1 + 0.1
0.9999999999999999
```

- To understand why the result is not 1.0, we need to learn how fractions are represented in binary

Decimal Fractions

- How do we interpret 0.538_{10}?
- "5 tenths, 3 hundredths, 8 thousandths”
- $0.538 = 5 \times 10^{-1} + 3 \times 10^{-2} + 8 \times 10^{-3}$

Binary Fractions

- Binary number can have a *binary point*
- Each digit to the right of the binary point is weighted by a negative power of 2
- 0.1011_2
  $= 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}$
  $= 0.5_{10} + 0.125_{10} + 0.0625_{10}$
  $= 0.6875_{10}$

Binary Fractions

- What is the binary representation of 0.1_{10}?
- $0.0001_2 = 0.0625_{10}$
- $0.00011_2 = 0.09375_{10}$
- $0.00011001_2 = 0.09765625_{10}$
- $0.000110011_2 = 0.099609375_{10}$

Increasing the precision (number of bits) gives us a better approximation of 0.1_{10}

Repeating Binary Fractions

- The decimal fraction $(1/10)_{10} = 1\times10^{-1} = 0.1_{10}$ has a finite decimal expansion, but in the binary representation, the sequence 0011 repeats indefinitely:
    - $0.1_{10} = 0.0001100110011001\ldots_2$
- We cannot represent a repeating binary fraction in a fixed number of bits, so we cannot represent 0.1_{10} exactly as a binary fraction

Floating-Point Approximations

- Python (and other programming languages) represents the literal value 0.1 by a 64-bit floating-point number that is a close approximation of 0.1_{10}, but is not equal to 0.1_{10}
- This introduces a slight error when floating-point arithmetic is performed (as seen in the earlier example)

Summary: Floating-Point Approximations

- If a decimal fraction has a finite decimal expansion, that does not mean it can be represented exactly as a binary fraction
- Most decimal fractions with finite expansions are stored as binary floating-point numbers that are close approximations of the decimal numbers
  - How close depends on the precision (the number of bits used to represent the number)

Repeating Decimals and Irrational Numbers

- The decimal expansions of some rational numbers (numbers than can be expressed as a fraction *p* / *q*) are repeating decimals (recurring decimals); i.e., become periodic; e.g., $(2/3)_{10} = 0.66666666\ldots$
- The decimal expansions of irrational numbers do not terminate or become periodic; e.g., $\pi = 3.1415926\ldots$
- These numbers cannot be represented exactly in binary, so close approximations are used instead

Recap of Learning Outcomes

Learning Outcomes

- Know how to calculate a % b when both are positive

- Know how to calculate a % b when one or both is negative

Learning Outcomes (Vocabulary)

- Know the meaning of these words
  - Binary number system
  - Bit
  - Base 10, base 2
  - Signed-magnitude

Learning Outcomes: Needed for ECOR 1044

- Convert unsigned and signed-magnitude binary integers to decimal integers
- Convert decimal integers to unsigned and signed-magnitude binary integers
- Note: There are other (better) ways to represent negative numbers in binary, but those will be covered in a later course.

Learning Outcomes (Vocabulary)

- Know the meaning of these words
  - Floating-point number
  - Binary fraction

Learning Outcomes

- Convert a binary fraction into a decimal fraction
- Convert a decimal fraction to a binary fraction
- Explain why an arithmetic operation on floating-point numbers may yield a result that is only approximately equal to the same operation performed on real numbers