CS 61C Summer 2022

Number Representation

Discussion 2

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and if false, correct the statement to make it true:

1.1

Depending on the context, the same sets of bits may represent different things.

True. The same bits can be interpreted in many different ways with the exact same bits! It can be from an unsigned number to a signed number or even a program. It is all dependent on your interpretation.

1.2 It is possible to get an overflow error in Two's Complement when adding numbers of opposite signs.

False. Overflow errors only occur when the correct result of the addition falls outside the range of $[-(2^{n-1}), 2^{n-1} - 1]$. Adding numbers of opposite signs will not result in numbers outside of this range.

1.3 If you interpret a N bit Two's complement number as an unsigned number, negative numbers would be smaller than positive numbers.

False. In Two's Complement, the MSB is always 1 for a negative number. This means ALL Two's Complement negative numbers will be larger than the positive numbers.

1.4 If you interpret an N bit Bias notation number as an unsigned number (assume there are negative numbers for the given bias), negative numbers would be smaller than positive numbers.

True. In bias notation, we add a bias to the unsigned interpretation to create the value. This means that negative numbers will stay smaller than positive numbers. This is unlike Two's Complement (see description above).

1.5 We can represent fractions and decimals in our given number representation formats (unsigned, biased, and Two's Complement).

False. Our current representation formats has a major limitation; we can only represent and do arithmetic with integers. To successfully represent fractional values as well as numbers with extremely high magnitude beyond our current boundaries, we need another representation format.

2 Unsigned Integers

2.1 If we have an n-digit unsigned numeral $d_{n-1}d_{n-2}\dots d_0$ in *radix* (or *base*) r, then the value of that numeral is $\sum_{i=0}^{n-1} r^i d_i$, which is just fancy notation to say that instead of a 10's or 100's place we have an r's or r^2's place. For the three radices binary, decimal, and hex, we just let r be 2, 10, and 16, respectively.

Let's try this by hand. Our preferred tool for writing large numbers is the IEC prefixing system, which is similar to scientific notation but with powers of 2 rather than 10:

Ki (Kibi) = 2^{10}	Gi (Gibi) = 2^{30}	Pi (Pebi) = 2^{50}	Zi (Zebi) = 2^{70}
Mi (Mebi) = 2^{20}	Ti (Tebi) = 2^{40}	Ei (Exbi) = 2^{60}	Yi (Yobi) = 2^{80}

For example, we would write 2^{81} as $2 * 2^{80} = 2$ Yi.

(a) Convert the following numbers from their initial radix into the other two common radices:

1. 0b10010011 = 147 = 0x93
2. 63 = 0b0011 1111 = 0x3F
3. 0b00100100 = 36 = 0x24
4. 0 = 0b0 = 0x0
5. 39 = 0b0010 0111 = 0x27
6. 437 = 0b0001 1011 0101 = 0x1B5
7. 0x0123 = 0b0000 0001 0010 0011 = 291

(b) Convert the following numbers from hex to binary:

1. 0xD3AD = 0b1101 0011 1010 1101 = 54189
2. 0xB33F = 0b1011 0011 0011 1111 = 45887
3. 0x7EC4 = 0b0111 1110 1100 0100 = 32452

(c) Write the following numbers using IEC prefixes:

- $2^{16} = 64$ Ki
- $2^{27} = 128$ Mi
- $2^{43} = 8$ Ti
- $2^{36} = 64$ Gi
- $2^{34} = 16$ Gi
- $2^{61} = 2$ Ei
- $2^{47} = 128$ Ti
- $2^{59} = 512$ Pi

(d) Write the following numbers as powers of 2:

- 2 Ki $= 2^{11}$
- 512 Ki $= 2^{19}$
- 16 Mi $= 2^{24}$
- 256 Pi $= 2^{58}$
- 64 Gi $= 2^{36}$
- 128 Ei $= 2^{67}$

;

3 Signed Integers

3.1 Unsigned binary numbers work for natural numbers, but many calculations use negative numbers as well. To deal with this, a number of different schemes have been used to represent signed numbers, but we will focus on two's complement, as it is the standard solution for representing signed integers.

- We can write the value of an n-digit two's complement number as $\sum_{i=0}^{n-2} 2^i d_i - 2^{n-1} d_{n-1}$.
- Negative numbers will have a 1 as their most significant bit (MSB). Plugging in $d_{n-1} = 1$ to the formula above gets us $\sum_{i=0}^{n-2} 2^i d_i - 2^{n-1}$.
- Meanwhile, positive numbers will have a 0 as their MSB. Plugging in $d_{n-1} = 0$ gets us $\sum_{i=0}^{n-2} 2^i d_i$, which is very similar to unsigned numbers.4
- A neat trick for flipping the sign of a two's complement number: flip all the bits and add 1.
- Addition is exactly the same as with an unsigned number.
- Only one 0, and it's located at 0b0.

For questions (a) through (c), assume an 8-bit integer and answer each one for the case of an unsigned number, biased number with a bias of -127, and two's complement number. Indicate if it cannot be answered with a specific representation.

(a) What is the largest integer? What is the result of adding one to that number?

1. Unsigned? 255, 0
2. Biased? 128, -127
3. Two's Complement? 127, -128

(b) How would you represent the numbers 0, 1, and -1?

1. Unsigned? 0b0000 0000, 0b0000 0001, not possible
2. Biased? 0b0111 1111, 0b1000 0000, 0b0111 1110
3. Two's Complement? 0b0000 0000, 0b0000 0001, 0b1111 1111

(c) How would you represent 17 and -17?

1. Unsigned? 0b0001 0001, not possible
2. Biased? 0b1001 0000, 0b0110 1110
3. Two's Complement? 0b0001 0001, 0b1110 1111

(d) What is the largest integer that can be represented by *any* encoding scheme that only uses 8 bits?

There is no such integer. For example, an arbitrary 8-bit mapping could choose to represent the numbers from 1 to 256 instead of 0 to 255.

(e) Prove that the two's complement inversion trick is valid (i.e. that x and $\overline{x}+1$ sum to 0).

Note that for any x we have $x + \overline{x} = 0b1\dots1$. Adding 0b1 to $0b1\dots1$ will cause the value to overflow, meaning that $0b1\dots1 + 0b1 = 0b0 = 0$. Therefore, $x + \overline{x} + 1 = 0$

A straightforward hand calculation shows that $0b1\dots1 + 0b1 = 0$.

(f) Explain where each of the three radices shines and why it is preferred over other bases in a given context.

Decimal is the preferred radix for human hand calculations, likely related to the fact that humans have 10 fingers.

Binary numerals are particularly useful for computers. Binary signals are less likely to be garbled than higher radix signals, as there is more "distance" (voltage or current) between valid signals. Additionally, binary signals are quite convenient to design circuits, as we'll see later in the course.

Hexadecimal numbers are a convenient shorthand for displaying binary numbers, owing to the fact that one hex digit corresponds exactly to four binary digits.

4 Arithmetic and Counting

4.1 Addition and subtraction of binary/hex numbers can be done in a similar fashion as with decimal digits by working right to left and carrying over extra digits to the next place. However, sometimes this may result in an overflow if the number of bits can no longer represent the true sum. Overflow occurs if and only if two numbers with the same sign are added and the result has the opposite sign.

(a) Compute the decimal result of the following arithmetic expressions involving 6-bit Two's Complement numbers as they would be calculated on a computer. Do any of these result in an overflow? Are all these operations possible?

1. $0b011001 - 0b000111$

   $0b010010 = 18$, No overflow.

2. $0b100011 + 0b111010$

Adding together we get 0b1011101, however since we are working with 6-bit numbers we truncate the first digit to get 0b011101 = 29. Since we added two negative numbers and ended up with a positive number, this results in an overflow.

3. 0x3B + 0x06

   Converting to binary, we get 0b111011 + 0b000110 = (after truncating as the problem states we're working with 6-bit numbers) 0b000001 = 1. Despite the extra truncated bit, this is not an overflow as -5 + 6 indeed equals 1!

4. 0xFF − 0xAA

   Trick question! This is not possible, as these hex numbers would need 8 bits to represent and we are working with 6 bit numbers.

5. 0b000100 − 0b001000

   The 2's complement of 0b001000 is 0b110111 + 1 = 0b111000. We add that to 0b000100 to get 0b111100.

   We can logically fact check this by converting everything to decimals: 0b000100 is 4 and 0b001000 is 8, so the subtraction should result in -4, which is 0b111100.

(b) What is the least number of bits needed to represent the following ranges using any number representation scheme.

1. 0 to 256

   In general n bits can be used to represent at most 2^n distinct things. As such 8 bits can represent $2^8 = 256$ numbers. However, this range actually contains 257 numbers so we need 9 bits.

2. -7 to 56

   Range of 64 numbers which can be represented through 6 bits as $2^6 = 64$

3. 64 to 127 and -64 to -127

   We are representing 128 numbers in total which requires 7 bits.

4. Address every byte of a 12 TiB chunk of memory

   Since a TiB is 2^{40} and the factor of 12 needs 4 bits, in total we can represent using 44 bits as 2^{43} bytes < 12 TiB < 2^{44} bytes

5 ASCII

5.1 The ASCII character encoding standard, developed in 1963, is an internationally agreed upon mapping of bitstrings to character symbols, allowing for the representation of characters and strings. That means that the same bitstring can have many different interpretations and meanings, all depending on its context.

(a) Given 151_6, what is the binary representation of this number? What is the corresponding ASCII value according to the ASCII table?

The binary representation is 0b01000011, the corresponding ASCII value is 'C'.

(b) For the given hexadecimal values, which of the following can be represented in C? If the value cannot, briefly justify your reasoning.

1. 0xAA524C00

   (a) Integer

   (b) Memory Address

   (c) String

   (d) Character

   The hexadecimal value can represent an integer as well as a memory address (in this case, an address in 32b architecture). However, there is no ASCII character for 0xAA, so the value cannot represent a string. Along with that, ASCII characters are 8 bits long, so the value cannot represent a char either.

2. 0x4465616462656500

   (a) Integer

   (b) Memory Address

   (c) String

   (d) Character

   The hexadecimal value cannot represent a 32 bit integer, but it can represent a memory address (now in 64b architecture). The value can represent a string ("Deadbee"), as the hexadecimal value ends with the null terminator 0x00. Finally, the value cannot represent an 8 bit character.