Integers

So far, we've seen how to convert numbers between bases.
How do we represent particular kinds of data in a certain (32-bit) architecture?
We will consider

- **integers**
- **floating point**
- **characters**

What do we know about integers?

- **They can be represented in binary using the digits (bits) 0 or 1**
- **The value N can be represented using K bits.**
  - **K = ceil (lg N)**
- **Suppose we have 32 bits.**
- **Then our numbers will be of the form**

$$N = \sum_{i=0}^{i=31} (b_i * 2^i) \qquad \text{where } b_i = 0 \text{ or } 1$$

- **This is called unsigned binary representation.**

Integers: unsigned binary

How many possible values are there for 32-bit unsigned binary representation?

- **There are 2^{32} different representations, so there are about 4 billion possible values**
- **Why 4 billion?**
- **Why do we say possible values?**

Minimum value: 0

Maximum value: $2^{32} - 1$ (approximately 4 billion)

- **In general, for N bits, the max value is $2^N - 1$**
- **Why -1, since there are 2^N representations?**
  - **Think of array of size 2^N. What is the largest index?**
  - **What is 2^N?**
  - **1 followed by N 0's (N+1 bits).**
  - **(2^2 = 100, 2^3 = 1000, 2^4 = 10000, etc.)**

Why do we use unsigned values? Anything which is only positive or zero.

- **Memory address**
- **Array index**
- **ASCII character**

What if we need to have negative values?

- **Need different representation.**

Integers: signed magnitude

Signed magnitude

- **Reserve one particular bit for the sign.**
- **Sign bit: If msb is 0, then positive value, if msb is 1, then negative value.**

Converting base 10 to N-bit signed magnitude (SM):

1. **Ignoring sign, convert base 10 value to binary.**
2. **If less than (N-1) bits, pad rest of bits to (N-1) with 0.**
3. **If negative, set msb to 1.**

Converting N-bit SM to base 10:

1. **Convert lower (N-1) bits to base 10.**
2. **If msb is 1, put minus sign in front of number.**

Example:

- **Convert 3 in base 10 to 4-bit SM:**
  - **binary value: 11**
  - **using 3 bits: 011**
  - **with sign bit: 0011**
- **Convert -3 in base 10 to 4-bit SM:**
  - **binary value: 11**
  - **using 3 bits: 011**
  - **with sign bit: 1011**
- **What about 15?**
  - **binary value: 1111**
  - **However, this is already 4 bits, and the sign bit needs to be 0 for positive value.**

Range of values for SM

- **How many possible values (i.e., representations)?**
  - **2^N for N bits**
- **Maximum value**
  - **2^{N-1} - 1 for (N-1) bits**
- **Minimum value**
  - **$-(2^{N-1} - 1)$ for (N-1) bits**

Total values, including 0:

$2 * (2^{N-1} - 1) + 1 = 2^N - 1$

What happened to the extra value?

There are actually 2 zeroes!

positive zero: N 0's

negative zero: 1 followed by (N-1) 0's

So, total number of representations is 2^N, but total number of values is 1 less.

Disadvantage of SM: 2 zeroes adds complexity to hardware.

Addition

It would be nice if we could use the same hardware for signed or unsigned addition.

However, that is not the case for signed magnitude.

Consider adding -1 and -1 in 4-bit SM

1001 + 1001 = 0010 in 4 bits, since the carry from the msb is lost.

This gives a result of +2, which is incorrect.

One way around this is to ignore the sign bit in doing the addition,

then use the sign bit for the result.

However, that doesn't work if the sign bits are different.

The sign bit of the result will depend on the relative magnitudes.

The magnitude of the result will be the difference in the magnitudes.

For example, (-5) + 3 is -2, but 5 + (-3) is +2.

Negation

Negating a value means giving it the opposite sign.

This is easy with SM, since we simply flip the sign bit.

negative value: sign bit 1 --> 0

positive value: sign bit 0 --> 1

Integers: 1's complement

Another way to represent negative values:

Negate (flip) each bit.

Use the operator ~ to represent flipping each bit.

~B is B with all of its bits flipped.

This is called 1's complement (1C).

Example: 11_{ten} is 01011 in 5-bit unsigned binary. Call this number B.

B = 01011

-B = 10100

Converting from base 10 to N-bit 1C

1. Ignoring sign, convert value to unsigned N-bit value.

2. If sign is negative, negate (flip) all bits.

1C to base 10

1. If sign bit (msb) is 1, flip all bits.

2. Convert N-bit unsigned value to base 10, using negative value if sign bit was 1.

Range of values

Number of positive values (including +0): 2^{N-1}

Number of negative values (including -0): 2^{N-1}

Total number of different values: $2^N - 1$

Negation

To negate a number, flip all bits.

Problems

2 zeroes, as in signed magnitude

addition is complicated, but easier than in signed magnitude

Since it has the same problems as SM, why look at 1C?

Because it leads to a better representation.

Integers: 2's complement

2's complement (2C) may not be intuitively obvious, but it has nice properties

Negation

1. **flip all bits**
2. **add 1 (ignoring any carry out of the msb)**

Example: 11_{ten} is 01011 in 5-bit unsigned binary. Call this number B.

1. flip bits:	10100	~B
2. add 1:	+ 1	
	10101	-B

What do we get if we add this to the original number?

01011	B
+ 10101	+ -B
00000	0

If we negate B (-B) using 2C and negate again (--B) we get the original value B back.

1. flip bits:	01010	~(-B)
2. add 1:	+ 1	
	01011	B

Other ways to do N-bit 2C negation:

1. **Find the rightmost 1 bit, then flip all the higher bits (bits to the left).**

   Example: 01011

   Rightmost 1 bit is b_0

   flip bits to the left: 10101

Why does this work?

Consider N-bit value value: $B = b_{N-1}b_{N-2} \ldots 1\ 0 \ldots 0$

where b_k is 1 and all b_i are 0 for $i < k$

If we create the negation -B by flipping the bits to the left of b_k, then

$-B = \sim b_{N-1}\sim b_{N-2} \ldots 1\ 0 \ldots 0$

What happens when we add these 2 numbers?

For each b_i, $i < k$, the sum of the bits is 0 + 0 = 0

For bit k, the sum of the bits is 1 + 1 = 10, so the result is 0 with carry 1
Now, for i > k, each bit is either 0 or 1 in B and the opposite in -B
So, the result is 1 + 0 + carry bit 1 for a result of 0 and a carry 1
For the leftmost bit position N-1, the result is 0, and the final carry is ignored.
Therefore, the result consists of all 0's, which must be true for B + (-B)

2. Subtract B from value: 1 followed by N 0's.

Example: 01011

subtract from 100000: 100000

- 01011

10101

Odometer (Ferris Bueller) view: Ferris Bueller's Day Off

Consider a 4-digit decimal odometer.

What is the largest milage it can show?

9999

What happens if we go 1 more mile?

0000

Now, think of starting at 0000 and going backward, like Ferris:

Go back 1 mile, and the odometer reads 9999, 2 miles 9998, and so forth.

N-bit 2's complement is really like a binary odometer:

Go forward from 00 . . . 0 for positive numbers.

Go backward for negative numbers.

What's really going on here mathematically?

What is the value of 1 followed by N 0's? 2^N

So, N-bit 2C is really based on arithmetic modulo 2^N

This is why 2C has nice properties for doing arithmetic.

Advantages of 2C representation:

Only 1 zero

Add using same hardware as unsigned binary

Subtract by taking complement and adding.

Why is there only 1 zero?

Consider the 5-bit positive zero 00000

1. flip bits: 11111

2. add 1: + 1

00000

This is negative 0, but it has the same representation :)

Range of values:

Maximum positive number: $2^{N-1} - 1$

Minimum negative number: $-(2^{N-1})$

Total number of values (including 0): 2^N

Converting from base 10 to N-bit 2C

1. Convert to N bits UB
2. If number has minus sign, negate:
   - flip bits
   - add 1

Converting from N-bit 2C to base 10

1. If msb is 1 (negative number), negate
2. Convert result (or original value) as UB to base 10
3. If number was negative, insert minus sign

Conclusion: 2C rules!

Most computer representations of signed integers use 2C.

Integers: excess/bias

One disadvantage of 2C:

Can't sort values just using the bit representation.

Would look like negative numbers were greater than positive numbers.

Another idea:

Consider the unsigned values for a 3-bit representation

representation	value	excess-4
000	0	-4
001	1	-3
010	2	-2
011	3	-1
100	4	0
101	5	1
110	6	2
111	7	3

Represent negative values, but keep the values in representation order

First half of the representations for negative, second half for positive

This is called excess, or biased, representation.

Each value is shifted by a constant amount; in this case the bias is 4.

Since there are 3 bits, the bias value is $2^{(3-1)} = 4$

In general, for an N-bit representation, we will have half of the values negative and half of the values non-negative if the bias is 2^{N-1}.

However, the bias could be any number $< 2^N - 2$ if the range of values includes both positive and negative numbers.

Range of values: N-bit excess K

Number of values: 2^N

Minimum negative value: -K

Maximum positive value: $2^N - 1 - K$

Converting base 10 to excess K

Add K to the number

Convert to unsigned binary

Converting excess K to base 10

Convert unsigned binary value to base 10

Subtract K

Disadvantage: Can't use unsigned binary hardware to do addition.

Excess notation is used as part of floating point representation.

Integers: BCD

Since people think in decimal, it may be convenient to use a decimal-like representation.
Binary coded decimal (BCD) uses unsigned binary to represent each decimal digit.
How many bits are needed?

4 bits per decimal digit: ceil (lg 10) = 4

Since 4 bits can represent 16 values, 6 values are unused for every digit.
Advantages

Decimal input/output doesn't require complicated conversion.
Fewer problems with repeating fractions.

Example:

Represent 1739_{ten} in BCD:

decimal	1	7	3	9
BCD	0001	0111	0011	1001

Some early computers used BCD representation, and many calculators still use BCD.
Since calculators can be much slower than computers, they can use hardware which operates on 4 bits at a time.
Negative values are represented using 10's complement (similar to 2C).

Integers: Sign extension

We may need to change the number of bits used to represent a number.
For example, C has int, short int, and long int to represent integers.
The language doesn't specify the exact size of each form, but
sizeof (short int) <= sizeof (int) <= sizeof (long int)
How do we cast from one kind of int to a larger number of bits?
ints are represented in N-bit 2C
To convert a number from N bits to N + K bits in 2C notation,
copy (extend) the sign bit from bit N-1 to bit N + K -1
If the value is positive, all the higher-order bits will become 0;
if the value is negative, all the higher-order bits will become 1.
All of the other bits (location N-2, . . . , 0) remain unchanged.
Example:
3_{ten} is 0011 in 4 bits
In 8 bits, it becomes 00000011 (the zero sign bit is extended to the left).
-3_{ten} is 1101 in 4 bits
In 8 bits, it becomes 11111101 (the one sign bit is extended to the left).
Casting to a shorter representation
May lose value
32-bit to 16-bit: upper 16 bits are chopped off
Unsigned binary
No sign bit, so extend with 0's