CAB203
Tutorial 2

Bits

1. Count:

   (a) How many different 8-bit strings are there?

   Solution: In general, for n-bit string there are 2^n possible strings. So in this case there are $2^8 = 256$ possible strings.

   (b) How many gigabits are there if you have one bit for every possible 32-bit string?

   Solution: There are 2^{32} possible bit strings, and so there are 2^{32} bits. There are 2^{30} bits in a gigabit, so we have $2^{32-30} = 2^2 = 4$ gigabits.
   This may sound like a convoluted question, but it is actually quite relevant. We typically use bit strings as *addresses* for data, and you need a unique address for each piece of data. So if you have addresses which are 32 bits wide, you can address 2^{32} possible pieces of data.

   (c) How many bits are required if you need 18 unique bit strings?

   Solution: Since there are 2^n possible unique strings for n-bit strings, we need to find the smallest n such that $2^n \geq 18$.
   First way: By inspection we see that $2^4 = 16$ is not enough, and $2^5 = 32$ is enough. So $n = 5$ is required.
   We could also use logarithms:

   $$\begin{aligned} 2^n &\geq 18 \\ \log_2 2^n &\geq \log_2 18 \\ n &\geq 4.1699 \end{aligned}$$

   We need an integer number of bits, so we take the ceiling (i.e. round up) to get $n = 5$.

2. Calculate the following expressions using bitwise operations:

   (a) 0011 & 1010

   Solution: We just go through each pair of bits at a time and apply &

   $$0010$$

   (b) 0011 | 1010

   Solution:

   $$1011$$

(c) 0011 ˆ 1010

Solution:

$$1001$$

(d) ∼1010

Solution:

$$0101$$

3. In the following, explain how to use bitwise operators to manipulate individual bits.

Solution: As a general princple, we can first have a look at how applying the various bit operations with a masking bit works. For example:

$$\begin{aligned} x \,|\, 0 &= x \\ x \,|\, 1 &= 1 \end{aligned}$$

Hence if we form $\overline{x} \,|\, \overline{m}$ for mask $\overline{m}$, wherever there is a 0 in $\overline{m}$ the corresponding bit of $\overline{x}$ will come through in the output. Wherever there is a 1 in $\overline{m}$ we will always get a 1 in the outcome.

Similarly,

$$\begin{aligned} x \,\&\, 0 &= 0 \\ x \,\&\, 1 &= x \end{aligned}$$

Following a similar logic to above, we can see how we can turn individual bits off by setting the corresponding bit in $\overline{m}$ to 0.

For flipping bits, we have

$$\begin{aligned} x \,\hat{}\, 0 &= x \\ x \,\hat{}\, 1 &= \sim x \end{aligned}$$

This allows us to flip individual bits by turning on the mask bits where we want to flip.

(a) In $\overline{x} = 1010$, turn on bit 2

Solution: We first form the mask. You can create the mask manually, or do the counting with a left shift:

$$\overline{m} = 1 << 2 = 0100$$

Then we apply the bit-wise |

$$\overline{x} \,|\, \overline{m} = 1110$$

(b) In $\overline{x} = 1100$, turn off all but bits 1 and 2

Solution: We first form the mask:

$$\overline{m} = (1 << 1) \,|\, (1 << 2) = 0110$$

Then we apply the bit-wise &

$$\overline{x}\,\&\,\overline{m} = 0100$$

(c) In $\overline{x} = 1110$, flip bit 3

Solution: We first form the mask:

$$\overline{m} = 1 << 3 = 1000$$

Then we apply the bit-wise ˆ

$$\overline{x}\ \hat{}\ \overline{m} = 0110$$

(d) In $\overline{x} = 0110$, turn off bit 1

Solution: We first form the mask:

$$\overline{m} = 1 << 1 = 0010$$

Then we apply the bit-wise & with $\sim\overline{m}$

$$\overline{x}\,\&\ \sim\overline{m} = 0100$$

Python bit manipulation

1. Define a Python function that takes two integers, x and j, and returns a value which is x, but with bit j set (set to 1).

Solution:

```
def setbit(x,j):
   return x | (1 << j)
```

2. Define a Python function that takes two integers, x and j, and returns a value which is x, but with bit j cleared (set to 0).

Solution:

```
def clearbit(x,j):
   return x & ~(1 << j)
```

3. Define a Python function that takes two integers, x and j, and returns a value which is x, but with bit j flipped.

Solution:

```
def flipbit(x,j):
   return x ^ (1 << j)
```

4. Define a Python function that takes two integers, x and j, and returns True if the jth bit of x is 1, otherwise False.

Solution:

```
def testbit(x,j):
   return x & (1 << j) != 0
```

Character and text representations

1. Find the ASCII representations of the following using the ASCII chart from the Lecture 2 slides
   (a) "A"

   Solution: Looking at the chart, we find "A" in row 1, column 4. We first look at the top of the column to find the first three bits, 100. Then looking at the left of the column, we find the last four bits, 0001. Together the bit string is 1000001.

   (b) "1"

   Solution: Following the same solution above, we find "1" in row 1, column 3. The bit string is 0110001.

   (c) ";"

   Solution: Following the same solution above, we find "1" in row 11, column 3. The bit string is 0111011.

2. Give the C-string for "CAB"

   Solution: As above, we find the bit strings for "C", "A", and "B", which are 1000011, 1000001, and 1000010. We need to add a leading 0 to these because C uses 8-bit characters. Adding the NUL character (00000000) on the end, we find the string to be:

   01000011 01000001 01000010 00000000

3. Put these string in lexicographic order: 1010, 11, 00110, 110011, 00001

Solution: Let's start by sorting the strings into two groups: those that start with 0, and those that start with 1. The ones starting with 0 will go first. So we have

$$00110, 00001 \quad 1010, 11, 110011$$

Looking at the first two, 00001 comes before 00110. The last three are already in order. So we end up with.

$$00001, 00110, 1010, 11, 110011$$

4. *(Stretch question)* The modern system for character encodings is *Unicode.* Unicode is actually a family of encodings for a common set of characters. The most common encoding is UTF-8, which is a *variable length encoding.* Some characters require only 8 bits to encode, and others require 16, 24 or 32 bits. How do you think this is accomplished? In particular, when looking at, say, 8 bytes. How do you know where the characters start and end if they can have different lengths?

Solution:

There are many possible variations, but the most important part is that you have some 8-bit patterns which cannot be a single character, and these are used to signal a multi-byte character.

In UTF-8 we partition the set of possible 8-bit strings into three sets. The first set of 8-bit strings start with 0 on the left. These bit strings are for characters encoded into 8 bits only. So if you see a byte starting with 0, it is a single character (and actually just ASCII!). Bytes starting with 11 are lead bytes, which are the first byte in a multi-byte character. Bytes starting with 10 are bytes which continue on after a lead byte. So if you see

11?? ???? 10?? ???? 0??? ????

then you have a two byte character followed by a single byte character. Likewise,

11?? ???? 10?? ???? 10?? ???? 10?? ???? 0??? ????

is a four byte character followed by a 1 byte character.

This method has the advantage that if you lose your place in a byte stream, you can just look for a byte starting with 0 or 11, and then you know you are at the beginning of a character. This is called *self-synchronisation.*

Integer representations

1. Convert these binary numbers to base-10

   (a) 111

   Solution: The 1 in the first postion has a multiplier of $2^0 = 1$. The second has a mulitplier of $2^1 = 2$, and the last has a multiplier of $2^2 = 4$, so we get

   $$1 + 2 + 4 = 7$$

   (b) 1010

Solution: Working as above we get

$$2^3 + 0 + 2^1 + 0 = 8 + 2 = 10$$

2. Add these binary numbers

   (a) 101 + 011

Solution: The addition procedure is exactly analogous to what we usually use for base 10. We work column by column, and carry a 1 if necessary. In base 10 we carry a 1 if the sum of a column is 10 or more, but in base 2 we carry a 1 if the sum of a column is 2 or more.
Below we have the columns, with the carry bits written above.

$$\begin{array}{cccc} 1 & 1 & 1 & \\ & 1 & 0 & 1 \\ + & 0 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 \end{array}$$

As a quick check, we see that in base 10 this is 5 + 3 = 8, which is what we have.

   (b) 1110 + 1010

Solution: Here is the solution written in columns, with the carry bits on top.

$$\begin{array}{ccccc} 1 & 1 & 1 & 0 & \\ & 1 & 1 & 1 & 0 \\ + & 1 & 0 & 1 & 0 \\ \hline 1 & 1 & 0 & 0 & 0 \end{array}$$

Note the column with three 1's. So we carry a 1, and there is a 1 below as well.
This comes out to $14 + 10 = 24$.

3. Convert these binary strings to hexidecimal

   (a) 11001011

Solution: We start by breaking the string up into two 4-bit strings, 1100 and 1011. 1100 is 12, which is numeral C. We can find this by looking in the chart in the lecture slides or just by counting from $A = 10$. The second string 1011 is $11 = B$. Together the answer is CB.

   (b) 00001001

Solution: Following the same method, we have $0000 \to 0$ and $1001 \to 9$. So the hexidecimal string is 09.

4. Convert these hexidecimal strings to binary

   (a) AE

Solution: We first convert each numeral to a 4-bit string. A is 10, which is 1010 and E is 14 which is 1110. Putting them together we get 10101110

(b) 10

Solution: We convert the same way. 1 has 4-bit string 0001. 0 has bit string 0000. Putting them together we get 00010000.

5. Assuming 4-bit numbers with 2's complement encoding, decode the following:

(a) 1100

Solution: For 4-bit numbers, the 2's complement encoding allows us to encode numbers $-2^3 = -8$ through to $2^3 - 1 = 7$. The string 1100 is binary for 12, which is not in the range, so it must instead be $12 - 16 = -4$.

(b) 0101

Solution: Here 0101 is binary for 5. This is in the range for 4-bit 2's complement encoding, so that's all there is to do!

6. *(Stretch question)* To negate a number x in 2's complement, you need to find the binary representation for $2^n - x$. There is a shortcut to doing this, which is used internally in CPUs. Supposing that $\overline{z}$ is the binary representation for x,

(a) NOT each bit in $\overline{z}$

(b) add 1 to the result.

Show that this procedure produces the binary representation for $2^n - x$. *Hint: look at a binary representation* $x = \sum_{j=0}^{n-1} z_j 2^j$ *and the version after applying the procedure:* $1 + \sum_{j=0}^{n-1}(1 - z_j)2^j$*. And yes, for a bit* b*,* $\sim b$ *is the same as* $1 - b$*.*

Solution: Let's start with the version after the applying the procedure.

$$1 + \sum_{j=0}^{n-1}(1 - z_j)2^j.$$

We can break the sum up into two parts:

$$1 + \sum_{j=0}^{n-1} 2^j - \sum_{j=0}^{n-1} z_j 2^j.$$

We recognise the last summation as x. The first summation is the sum of all powers of two less than n. We might recognise that this is a geometric series, in which case we can apply the geometric series formula and find out that

$$\sum_{j=0}^{n-1} 2^j = \frac{2^n - 1}{2 - 1} = 2^n - 1.$$

Adding the extra 1, we get $2^n - x$.

If you don't know about geometric sums, you could instead think about what the bit string for the first summation looks like, which is $11\dots1$. If we add the extra 1, then it's pretty easy to see that we will end up carrying 1 in every place, and the sum will look like $100\ldots0$ with n zeros. This is the binary representation of 2^n.

If the first two ideas don't work for you, then just look at the pattern. 1 is the binary representation of $1 = 2^1 - 1$. 11 is the binary representation of $3 = 2^2 - 1$, etc. This pattern keeps going.