Computer Science 281
Binary and Hexadecimal Review

1 The Binary Number System

Computers store everything, both instructions and data, by using many, many transistors, each of which can be in one of two states: off or on. We represent the former transistor state with a 0 and the latter state with a 1. The number system that uses only these two digits is called *binary* (base 2) and each binary digit (0 or 1) is called a *bit*. Every instruction we give a computer and every number, character, and object is ultimately represented in binary notation.

We can represent numbers in any base we like. We are accustomed to base 10, but the rules are the same for any base. Let's say we are using the base n number system with digits having values $0, 1, \ldots, n-1$. Then in any number in this base, the i^{th} (i starting at 1) digit d from the right has the value $d \cdot n^{i-1}$. The decimal number 7104, for example, is equal to $7 \cdot 10^3 + 1 \cdot 10^2 + 0 \cdot 10^1 + 4 \cdot 10^0$.

In binary, the place values are $2^0, 2^1, 2^2, \ldots$. So the binary number `10110` has the decimal value $1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 16 + 0 + 4 + 2 + 0 = 22$.

In general, you can use the following algorithm to convert a binary number to decimal:

1. Write the place values over the binary digits.
2. Multiply each place value by the digit in that position, and add the results together.

How many different numbers can be represented with n binary digits? How might we figure this out? First, let's look at an example with 4 digits. The smallest 4 digit binary number is `0000` and the largest is `1111`, which is $8 + 4 + 2 + 1 = 15$ in decimal. Therefore, there are 15 + 1 = 16 possible bit patterns with 4 digits. In general, there are 2^n different bit patterns available with n binary digits because there are 2 possible digits per position and there are n positions.

1.1 Converting from Decimal to Binary

The easiest way to convert a decimal number to binary is to use what is known as the division method:

1. Divide the number by its base (in this case 2).
2. Write down the remainder to the left of the digits that have already been found.
3. If the quotient is 0, then stop. Otherwise, replace the number with the quotient and continue with step 1.

Example 1 Convert the decimal number 102 to binary.

We use the division method:

division	quotient	remainder
102/2	51	0
51/2	25	1
25/2	12	1
12/2	6	0
6/2	3	0
3/2	1	1
1/2	0	1

Therefore, $102_{10} = \texttt{1100110}_2$.

Subscripts are used to make clear the bases we are using. They are not necessary if the bases are clear in the current context.

To check the calculation, we can convert the answer back to decimal:

$$\texttt{1100110}_2 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 64 + 32 + 4 + 2 = 102.$$

■

1.2 Adding Binary Numbers

Addition in binary is simple. You only need to remember that $\texttt{0} + \texttt{1} = \texttt{1} + \texttt{0} = \texttt{1}$, $\texttt{1} + \texttt{1} = \texttt{10}$, and $\texttt{1} + \texttt{1} + \texttt{1} = \texttt{11}$.

Example 2 Add the binary numbers `101` and `10`.

	Binary				*Decimal*	
		1	0	1		5
+			1	0	+	2
		1	1	1		7

Example 3 Add the binary numbers `101101` and `1110`.

		Binary						*Decimal*	
			1	1					← carries
		1	0	1	1	0	1	45	
+				1	1	1	0	+ 14	
	1	1	1	0	1	1		59	

2 Representing Negative Integers: Two's Complement Notation

2.1 Finite Precision

On any real computer, the number of bits allocated to the representation of an integer is limited. On most machines this limit is 32 bits. The actual number of bits is irrelevant to understanding

Binary	Decimal	Binary	Decimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

Figure 1: Unsigned 4 bit binary numbers.

binary arithmetic, but you do need to recognize that computer arithmetic has a finite precision. This implies two important things:

1. There are some numbers that can not be represented in a computer. This includes integers that require more bits than the computer uses to represent an integer and real numbers that are non-terminating in binary notation. (We will examine the latter problem later in the semester.)
2. If an integer is represented with n bits in a computer, then *every* integer is represented with n bits. For example, if an integer is 32 bits long then 3_{10} is not represented by `11`, but rather `00000000000000000000000000000011`.

We will use 4 bit precision as an example in this section. There are $2^4 = 16$ different bit patterns using 4 bits, as shown in Figure 1.

2.2 Two's Complement

Most computers use a system called two's complement to represent the integers. The basic idea with any such system is this:

1. Choose some number of bits (precision) to represent the integers. This is usually either 32 or 64 on real computers.
2. Half of the bit patterns represent positive integers;
   half of the bit patterns represent negative integers.

With 4 bit precision, the two's complement numbers look like this:

Two's Complement	Decimal	Two's Complement	Decimal
0000	0	1000	-8
0001	1	1001	-7
0010	2	1010	-6
0011	3	1011	-5
0100	4	1100	-4
0101	5	1101	-3
0110	6	1110	-2
0111	7	1111	-1

To take the two's complement of a binary number, you can use the following algorithm:

1. Take the *one's complement* of the binary number by changing every `1` to a `0` and every `0` to a `1`.
2. Add `1` to the number formed in step 1. Ignore the carry out of the most significant bit if there is one. The result is the two's complement.

Example 4 Find the 4 bit two's complement representation of -6_{10}.

First, we find that 6_{10} = `0110`. Then we take the one's complement of `0110`, which is `1001`. Finally, we add `1`, giving `1001 + 1 = 1010`. Therefore, in two's complement notation, $-6_{10} =$ `1010`. ■

On your own, find the 4 bit two's complement representations of the integers 1 to 7 and check your answers against the table above.

There are few things you should notice about two's complement that hold no matter how many bits we use:

1. An n bit finite precision two's complement integer is negative if, and only if, its leftmost (most significant) bit is a `1`. For this reason, the leftmost bit is called the *sign bit*.
2. For n bits of precision, the largest positive n bit finite precision integer is $2^{n-1} - 1$, and the most negative n bit finite precision integer is $-(2^{n-1})$. There is one more negative number than positive.
3. If you add any n bit two's complement number and its negative together, treating each as if it were an unsigned binary number, the result will be 2^n. Therefore, the negative of a two's complement number x is $2^n - x$ (which is another way to compute the two's complement). This fact is the origin of the name "two's complement".
4. For any integer x, $-(-x) = x$.
5. The most negative integer in a two's complement number system is an oddity because if you take the two's complement of it, you get itself back. For this reason, it is not used in most two's complement systems.

2.2.1 Two's Complement Shortcut

There is an easier way to find the two's complement of an integer. Simply complement each bit to the left of the rightmost (lowest order) 1.

Example 5 Find the 7 bit two's complement representation of -17_{10}.

The 7 bit binary equivalent of 17 is `0010001`. If we complement each bit to the left of the rightmost 1, we get `1101111`. Therefore, in two's complement notation, $-17_{10} =$ `1101111`. ■

2.3 Two's Complement Addition and Subtraction

The mechanics of two's complement addition are just like unsigned binary addition **except:** If the answer has more bits than the precision, truncate the most significant (left) bit.

Discarding the carry out of the most significant bit is a normal part of the process.

Example 6 Add `0110` and `1110` using 4 bit two's complement arithmetic.

Binary	*Decimal*
`0110`	6
`+ 1110`	+ -2
`10100`	4

Therefore, `0110` + `1110` = `0100`.

The subtraction $A - B$ is implemented as the addition $A + (-B)$ where $-B$ is the two's complement of B.

Example 7 Subtract `0110` from `1110` using 4 bit two's complement arithmetic.

Binary			*Decimal*
`1110`		`1110`	14
`- 0110`	$\Rightarrow$	`+ 1010`	- 6
		`11000`	8

Therefore, `1110` - `0110` = `1000`.

2.3.1 Overflow

Consider the following example:

Example 8 Add `0111` and `0100` using 4 bit two's complement arithmetic.

Binary	*Decimal*
`0111`	7
`+ 0100`	+ 4
`1011`	-5 !!

What happened here? The value of the answer was too large to represent accurately with 4 bit precision. The result is called *overflow*. Fortunately, overflow is very easy to detect. ■

The Overflow Rule:

> Overflow in finite precision addition occurs only when two numbers of the same sign are added and the result has the opposite sign. Overflow in finite precision addition can never occur when numbers of opposite signs are added. This is because the answer in this case cannot be larger than either operand.
>
> Overflow in finite precision subtraction can only occur when subtracting numbers of the opposite sign. To determine whether overflow occurs in finite precision subtraction, convert the problem to an addition problem and check for addition overflow.

Do not confuse overflow with discarding the most significant carry bit, which is a normal part of the process.

3 The Hexadecimal Number System

The problem with the binary number system is that relatively small numbers use a large number of bits, which human beings have trouble reading. For example,

$$8192_{10} = 10000000000000 \neq 1000000000000 = 4096_{10},$$

but it is hard for us to tell that by just looking. The hexadecimal number system (base 16) has two things going for it:

1. Numbers represented in hexadecimal are far easier for humans to compare (and write) than are binary numbers.
2. The conversion between binary and hexadecimal is very easy, much easier than between binary and decimal.

Therefore, hexadecimal is often used as a shorthand for binary strings. Since hexadecimal is base 16, we need 16 digits. The hexadecimal digits are the decimal digits, plus the digits A, B, C, D, E, and F which denote the values 10, 11, 12, 13, 14, and 15, respectively. As you should expect by now, the place values in hexadecimal are $16^0, 16^1, 16^2$, etc.. For example, here is a table of the first 34 hexadecimal numbers and their decimal equivalents:

dec	hex	dec	hex	dec	hex	dec	hex	dec	hex	dec	hex	dec	hex
0	0	5	5	10	A	15	F	20	14	25	19	30	1E
1	1	6	6	11	B	16	10	21	15	26	1A	31	1F
2	2	7	7	12	C	17	11	22	16	27	1B	32	20
3	3	8	8	13	D	18	12	23	17	28	1C	33	21
4	4	9	9	14	E	19	13	24	18	29	1D	34	22

You can easily convert a hexadecimal number to decimal by multiplying each digit by its place value and adding, just as we did with binary.

Example 9 Convert $\texttt{BE3}_{16}$ to decimal.

The hexadecimal number BE3 is equivalent to

$$11 \cdot 16^2 + 14 \cdot 16^1 + 3 \cdot 16^0 = 11 \cdot 256 + 14 \cdot 16 + 3 \cdot 1 = 2816 + 224 + 3 = 3043$$

in decimal. ■

3.1 Converting from Decimal to Hexadecimal

To convert from decimal to hexadecimal, you can use the division method again, but with base 16 instead of 2.

Example 10 Convert 4798_{10} to hexadecimal.

We use the division method:

division	quotient	remainder	
4798/16	299	14	(E)
299/16	18	11	(B)
18/16	1	2	
1/16	0	1	

Therefore, $4798_{10} = \texttt{12BE}_{16}$. To check you work, make sure that

$$1 \cdot 16^3 + 2 \cdot 16^2 + 11 \cdot 16^1 + 14 \cdot 16^0 = 1 \cdot 4096 + 2 \cdot 256 + 11 \cdot 16 + 14 \cdot 1 = 4798.$$

■

3.2 Converting between Binary and Hexadecimal

Converting between binary and hexadecimal is easy because each hexadecimal digit, which has a value between 0 and 15, is equivalent to 4 bits, as can be seen in Figure 1. Therefore, to convert from hexadecimal to binary, simply replace each hexadecimal digit with its 4 bit binary equivalent.

Example 11 Convert $\texttt{12BE}_{16}$ to binary.

We simply replace each digit by its corresponding bit pattern:

```
  1     2     B     E
0001  0010  1011  1110
```

Therefore, $\texttt{12BE}_{16} = \texttt{1001010111110}_2$. ■

Converting from binary to hexadecimal is almost as easy. Starting on the right, group the binary digits in groups of 4, then replace each group with its corresponding hexadecimal digit.

Example 12 Convert `11100101010001100110010000111 0` to hexadecimal.

Group the bits in groups of 4, starting at the right, and then replace each group with its hexadecimal equivalent:

```
 11  1100  0101  0001  1001  1001  0000  1110
  3     C     5     1     9     9     0     E
```

Therefore, $\texttt{11100101010001100110010000111 0} = \texttt{3C51990E}_{16}$. ■