Computer system unit 2 Assessment 2

1. Twos compliment

Two's complement is the most popular method of signifying negative integers in computer science. It is also an operation of negation (converting positive to negative numbers or vice versa) in computers which represent negative numbers using two's complement. Its use is ubiquitous today because it doesn't require the addition and subtraction circuitry to examine the signs of the operands to determine whether to add or subtract, making it both simpler to implement and capable of easily handling higher precision arithmetic. As well, 0 has only a single representation, obviating the subtleties associated with negative zero.

In the two's complement representation, the most significant bit of a signed binary value indicates the sign. If the sign bit is zero, the value is non-negative binary number. If the most significant (leftmost) bit is 1, the value is negative: the bits contain a two's complement version of the value. To obtain the value of a negative number, all the bits are inverted then 1 is added to the result. If all bits are one the value is negative one; if the sign bit is on but the rest of the bits are off the value is the most-negative number. The most negative number cannot be represented as a positive number with the same number of bits.

A signed 8-bit binary numeral can represent every integer in the range −128 to +127. If the sign bit is 0, then the largest value that can be stored in the remaining seven bits is $2^7 - 1$, or 127.

Using two's complement to represent negative numbers allows only one representation of zero, and to have effective addition and subtraction *while* still having the most significant bit as the sign bit.

2.Hexadecimal

Hexadecimal numbers are used for the benefit of human programmers, as they are easier to handle than long strings of binary 1s and 0s - with less chance of making an error. Hexadecimal numbers might be used in printouts of a machine code program which a programmer needs to check or amend.

A common use of hexadecimal numerals is found in HTML and CSS. They use hexadecimal notation (hex triplets) to specify colours on web pages; there is just the #

symbol, not a separate symbol for "hexadecimal". Twenty-four-bit color is represented in the format #RRGGBB, where RR specifies the value of the Red component of the color, GG the Green component and BB the Blue component. For example, a shade of red that is 238,9,63 in decimal is coded as #EE093F. This syntax is borrowed from the X Window System.
In URLs, special characters can be coded hexadecimally, with a percent sign used to introduce each byte; e.g., http://en.wikipedia.org/wiki/Main%20Page
The canonical written form of numeric IPv6 addresses represents each group of 16 bits as a separate hexadecimal number, to ease reading and transcription of the 128-bit addresses.

When working with computers we often need to deal with binary data. It is much easier to handle numbers in hexadecimal than in binary (just think of lots of '0's and '1's) and whilst we are more familiar with the base 10 system, it is much easier to map binary to hexadecimal than to decimal since each hexadecimal digit maps to a whole number of bits (4_{10}).
Consider converting 1111_2 to base 10. Since each position in a binary (base 2) number can only be either a 1 or 0, its value may be easily determined by its position from the right:

- $0001_2 = 1_{10}$
- $0010_2 = 2_{10}$
- $0100_2 = 4_{10}$
- $1000_2 = 8_{10}$

Therefore:

$$1111_2 = 8_{10} + 4_{10} + 2_{10} + 1_{10}$$
$$= 15_{10}$$

This is a very simple example which still requires the addition of 4 numbers; whereas, with some practice, 1111_2 can be mapped directly to F_{16} in one step. When the binary number is very much greater, conversion to decimal becomes very much more tedious; however, when mapping to hexadecimal, it is simple to divide the binary number up in blocks of 4 positions and map each block of 4 bits to a single position hexadecimal number. For example a tedious conversion to decimal:

$$01011110101101010010_2 = 262144_{10} + 65536_{10} + 32768_{10} + 16384_{10} + 8192_{10} + 2048_{10} + 512_{10} + 256_{10} + 64_{10} + 16_{10} + 2_{10}$$
$$= 387922_{10}$$

0Compared to the conversion to hexadecimal:

$$01011110101101010010_2 = 0101\ \ 1110\ \ 1011\ \ 0101\ \ 0010_2$$
$$= 5\ \ E\ \ B\ \ 5\ \ 2_{16}$$
$$= 5EB52_{16}$$

Conversion from hexadecimal back to binary is just as direct.

3a. Numeric

When an input is numeric, it is important that it has to be valid for example if a number was to be entered it cannot be in text form, as calculation will not work and the output will be un-valid.

Numeric data may have character-oriented or binary representations. In the former, the number "27" would be represented by two pairs of (typically) 6 or 8 bits, one set representing the character "2" and the other set the character "7". One example of this kind of representation is known as "ASCII" characters. ASCII is the abbreviation for the "American Standard Code for Information Interchange". Specific information about character representations is documented here for various formats

Binary Data uses sets of N bits (N is typically 16, 18, 32, 36, 48, or 60 bits - or more!) to represent numbers of a wide range of sizes. Binary representations may be pure integer numbers or floating point numbers.

3b ASCII

ASCII stands for the American Standard Code for Information Interchange, and is pronounced with a hard 'c' sound, as *ask-ee*. As a standard, ASCII was first adopted in 1963 and quickly became widely used throughout the computer world.

ASCII is a way of defining a set of characters, which can be displayed by a computer on a screen, as well as some control characters, which have special functions. Basic ASCII uses seven bits to define each letter, meaning it can have up to 128 specific identifiers, two to the seventh power. This size was chosen based on the common basic block of computing, the byte, which consists of eight bits. The eighth bit was often set aside for error-checking functions, leaving seven remaining for a character set.

Thirty-three codes in ASCII are used to represent things other than specific characters. The first 32 (0-31) represent things ranging from a chime sound, to a line feed command, to the start of a header. The final code, 127, represents a backspace. Beyond the first 31 bits are the printable characters. Bits 48-57 represent the numeric digits. Bits 65-90 are the capital letters, while bits 97-122 are the lower-case letters. The rest of the bits are symbols of punctuation, mathematical symbols, and other symbols such as the pipe and tilde.

ASCII began in theory as a simpler character set, using six rather than seven bits. Ultimately it was decided that the addition of lower-case letters, punctuation, and control characters would greatly enhance its usefulness. Not long after its adoption, much discussion arose about possible replacements and adaptations of ASCII to incorporate non-English and even non-Roman characters. As early as 1972 an ISO standard (646) was created in an attempt to allow a greater range of characters. A number of problems existed with ISO-646, however, leaving it by the wayside.

The current leading contender for replacing ASCII is the Unicode character set. This standard allows for essentially unlimited characters to be mapped by using collections of bytes to represent a character, rather than a single byte. The first byte of all Unicode standards remains dedicated to the ASCII character set, however, to preserve backward compatibility.

ASCII is now most often heard in the phrase *ASCII art*. This describes the use of the basic character set to create visual approximations of images

3c Bit Masks

A bit mask makes use of the fact that binary numbers are made up of 1's and 0's, each digit in a binary number being equivalent to one bit. This makes binary numbers ideal for use as “switches” to enable or disable certain facilities.

Binary numbers are always read from right to left, and when used as bit masks the same is true. The rightmost digit is always bit 0, so taking 1010 as an example bit 0 is off, bit 1 is on, bit 2 is off and bit 3 is on.
A good example of bit mask usage is when setting the log levels for a particular service. For example, if you enable all the logging options for the SMTP service you will end up with the SMTPLog variable set to a value of 4382. Converting 4382 to its binary equivalent gives 1000100011110, that is bits 1, 2, 3, 4, 8 and 12 are all set.

Bit masks are always converted to their decimal equivalent prior to entering them.

3d Bit Map

A bit map defines a display space and the colon for each pixel or "bit" in the display space. A Graphics Interchange Format and a JPEG are examples of graphic image file types that contain bit maps.
A bit map does not need to contain a bit of colour-coded information for each pixel on every row. It only needs to contain information indicating a new colour as the display scans along a row. Thus, an image with much solid colour will tend to require a small bit map.
Because a bit map uses a fixed or raster graphics method of specifying an image, a user cannot immediately rescale the image without losing definition. A vector graphic image, however, is designed to be quickly rescaled. Typically, an image is created using vector graphics and then, when the artist is satisfied with the image, it is converted to (or saved as) a raster graphic file or bit map.

4. Error in data representation

Input conversion errors arise because the internal representation of a floating-point value often cannot exactly represent the value being converted. This occurs whenever the floating-point representation has fewer bits than are needed to hold the value being converted. The most common source for this form of error is probably base conversion. For example, the value 0.1 has an infinitely repeating representation as a base two fraction. If floating-point numbers on our target system use binary representation we have to cut that infinitely repeating binary fraction off to fit it in a finite sized floating-point representation, and the resulting value is not exactly 0.1. But the finite representation affects more than infinite representations. Even if an input value can be exactly represented in the base used by the floating-point representation, it can be too long to fit in the number of bits available. In that case it will also be cut off, and the resulting value will not be exactly right.

5a. 0110100101000011

The Mantissa is 0.110100101
The Exponent is 000011 which has a value of 3
Moving the binary point of the mantissa by 3 places gives:

= 0 1 1 0. 1 0 0 1 0 1
1 0 0 1. 0 1 1 0 1 0 Invert
+1 Add
1 0 0 1. 0 1 1 0 1 1

1	0	0	1	0	1	1	0	1	1
8	4	2	1	.5	.25	0.125	0.0625	0.03125	0.015625

8+1+.25+.125+.03125+.015625= 9.484375

Or

The Mantissa is 0.110100101
The Exponent is 000011 which has a value of 3
Moving the binary point of the mantissa by 3 places gives:

0110.100101
Which has value of
256+128+0.5+0.0625+0.015625= 384.578125

5b.1110110000111100

The Mantissa has the 1.110110000 which slightly more than 1

The Exponent has a value 111100 in two's compliment notation. Converting this to its positive equivalent gives 000011, which equals to 3.
So exponent is therefore -3
Moving the binary point of the mantissa by -3 places gives:

$$1.110110000 \times 2^{-3}$$

= 1110.110000
0001.001111 Invert
+ 1 Add
0001.010000

64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125	0.015625
		0	0	0	0	1	0	1	0	0	0	0

1+0.25= -1.25

5c. Maximum positive number

512	256	128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1	1	1

512+256+128+64+32+16+8+4+2+
=1023

6. **-2.25 in Normalised form**

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**
512	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**	**0.5**	**0.25**	**0.125**	**0.0625**
2										**25**			

0000000010/010000 binary

0.000000010

The Mantissa is 0.000000010

The Exponent is 0100 which has a value of 4
Moving the binary point of the mantissa by 4 places gives:

= 0000.000010
1111.111101 Invert
+1 Add
1111.111110

Or

0000000010.010000 binary
1111111101.101111 Invert
-1 Minus
1111111101.101110

7. <u>Floating Point</u>

A floating-point number is a digital representation for a number in a certain subset of the rational numbers, and is often used to approximate an arbitrary real number on a

computer. In particular, it represents an integer or fixed-point number (the significant or, informally, the mantissa) multiplied by a base (usually 2 in computers) to some integer power (the exponent). When the base is 2, it is the binary analog of scientific notation (in base 10).
A *floating-point calculation* is an arithmetic calculation done with floating-point numbers and often involves some approximation or rounding because the result of an operation may not be exactly representable.
A floating-point number a can be represented by two numbers m and e, such that $a = m \times b^e$. In any such system we pick a base b (called the *base* of numeration, also the *radix*) and a precision p (how many digits to store). m (which is called the *significant* or, informally, *mantissa*) is a p digit number of the form ?d.ddd...ddd (each digit being an integer between 0 and $b-1$ inclusive). If the leading digit of m is non-zero then the number is said to be normalized. Some descriptions use a separate sign bit (s, which represents −1 or +1) and require m to be positive. e is called the *exponent*.
This scheme allows a large range of magnitudes to be represented within a given size of field, which is not possible in a fixed-point notation.

Fixed point

In computing, a fixed-point number representation is a real data type for a number that has a fixed number of digits after the decimal (or binary or hexadecimal) point. For example, a fixed-point number with 4 digits after the decimal point could be used to store numbers such as 1.3467, 281243.3234 and 0.1000, but would round 1.0301789 to 1.0302 and 0.0000654 to 0.0001.
Fixed-point can exactly represent decimal fractions while still employing the base 2 arithmetic efficient in most of today's computers. Most floating point representations in computers use base 2 values, which cannot exactly represent most fractions that are easily represented in base 10. For example, one-tenth (.1) and one-hundredth (.01) can be represented only approximately by base-2 floating point representations, while they can be represented exactly in fixed-point representations — one simply stores the data values multiplied by the appropriate power of 10.
As long as the numeric value uses only the number of digits specified after the decimal point, fixed-point values can exactly represent all values up to its maximum value (determined by the number of bits in its representation). This is in contrast to floating-point representations, which include an automatically-managed exponent but cannot represent as many digits accurately (given the same number of bits in its representation).

Advantage of floating poin t notation over fixed point notation

Very few computer languages include built-in support for fixed point values, because for most applications, floating-point representations are fast enough and accurate enough. Floating-point representations are more flexible than fixed-point representations, because they can handle a wider dynamic range. Floating-point representations are also slightly easier to use, because they do not require programmers to specify the number of digits after the decimal point.