

Student name: _____

Addition in Binary and Hexadecimal

Binary addition has four basic rules, the first 3 of which are pretty obvious:

```

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 carry 1.
  
```

Example 1:

```

      0  1  0  0  1  1  0  1
    +  0  0  0  0  0  1  1  0
  carry ->
  result -> 0  1  0  1  0  0  1  1
  
```

If you are adding in columns and you have to do 1 + 1 + carry then the result will be 1 carry 1 to the next column:

```

      0  0  0  0  1  1  1  1
    +  0  0  0  0  0  1  1  0
  carry ->
  result -> 0  0  0  1  0  1  0  1
  
```

Confirm that these additions are correct by converting the numbers to base 10. Try these:

```

      1  0  0  0  1  0  1  0
    +  0  0  0  1  1  1  0  1
  carry ->
  result ->
  
```


```

      1  1  0  0  1  1  1  1
    +  0  0  1  0  0  0  1  1
  carry ->
  result ->
  
```


```

      1  1  1  1  0  0  0  1
    +  0  0  0  0  1  1  1  1
  carry ->
  result ->
  
```


Student name: _____

C + D in hex gives 19, carrying the 1. Is that right? Convert to decimal and check.

```

          0   0   0   C
          0   0   0   D
    carry ->           1
    result -> 0   0   1   9
  
```

Of course, some people with a natural aptitude for number work will be able to do this in their heads – that’s OK, we don’t mind, we have other talents don’t we?

Try these - **no calculators** allowed! (Except for your poor old teacher, that is):

```

          0   2   9   A
          0   2   0   B
    carry -> 

|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|  |  |  |  |


    result -> 

|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|  |  |  |  |


```

```

          0   F   9   0
          0   2   9   D
    carry ->
    result ->
  
```

Student name: _____

Fixed Point Representation of Binary Numbers

If you consider decimal numbers, when we want to represent fractions we use a decimal point:

234.56

The fractional part is $5/10 + 6/100$.

So, for binary it is just an extension of the column weighting system to beyond the “becimal” point where the weights are $1/2$, $1/4$, $1/16$ etc.

In a similar way to converting decimal numbers to binary, where we used repeated division by 2, we can convert decimal fractions to binary fractions by repeated multiplication:

$$0.65 * 2 = 1.30 \quad \text{first bit is } 1$$

$$0.30 * 2 = 0.60 \quad \text{next bit is } 0$$

$$0.60 * 2 = 1.20 \quad \text{next bit is } 1$$

$$0.20 * 2 = 0.40 \quad 0$$

$$0.40 * 2 = 0.80 \quad 0$$

$$0.80 * 2 = 1.60 \quad 1$$

The pattern will now repeat itself. Depending on how many bits we want to store, we truncate or round this result - let's use 5 bits:

0.101001 truncates to 0.10100 and rounds to 0.10101

The rounded version represents $1/2 + 1/8 + 1/32 = 21/32 = 0.65625$ so this is not a very accurate way to store real numbers. What does the truncated version evaluate to as a decimal fraction?

A number such as

1 0 1 0 1 0 . 0 1 0 1 1

is 42.34375 in decimal. We can obtain the negative two's complement representation in exactly the same way as was done for integers:

Using 8 bits for the whole number part and 5 bits for the fractional part, then 42.34375 is:

-128	64	32	16	8	4	2	1	.	1/2	1/4	1/8	1/1	1/3
0	0	1	0	1	0	1	0	.	0	1	0	6	2
												1	1

Student name: _____

Write down the binary representation of -42.34375 :

									.				
--	--	--	--	--	--	--	--	--	---	--	--	--	--

Using the above 13 bit representation, convert the following numbers to binary, truncate any bits after 5 beyond the bicimal point, if necessary.

a) 1.7

									.				
--	--	--	--	--	--	--	--	--	---	--	--	--	--

b) the largest possible negative number

									.				
--	--	--	--	--	--	--	--	--	---	--	--	--	--

c) the smallest possible positive number

									.				
--	--	--	--	--	--	--	--	--	---	--	--	--	--

You can see that **fixed-point representation** has some limitations, just as it has in decimal. Really large and really small numbers are hard to represent unless you're prepared to allocate very many bits of storage.

Student name: _____

Floating Point Binary Numbers

First some terminology:

given 0.32855932×10^4 .

The number (ie 0.32855932) is called the **mantissa** and the power (ie 4) is called the **exponent**.

The number part is usually **normalized**, ie made to fit within a certain range (in the above examples, the first digit after the decimal point has to be 1 through 9).

Using the decimal number, 363.499 normalize it, then identify the mantissa and the exponent:

Normalised:..... mantissa:..... exponent:.....

To represent binary floating-point numbers we will use a binary (fractional) mantissa and an integer exponent which both use two's complement representation in 16-bits (more bits would be allocated in a real computer system).

A possible compromise between a large mantissa (greater accuracy) and a large exponent (greater range) could be to allocate 10 bits for the mantissa and 6 for the exponent:

implied point <----- mantissa -----><-----exponent ----->

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

-1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	-32	16	8	4	2	1
0	1	1	0	1	0	1	0	0	0	0	0	0	1	0	0

What is this in decimal?

Well, do it in two parts, first the mantissa stays as it is; the value of the exponent is +4.

Therefore the binary point must be moved **four places** to the **right** in the mantissa:

0.110101000 becomes 1101.01000 that is 13.25 in decimal.

What is this number in decimal?

-1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	-32	16	8	4	2	1
0	1	0	0	1	0	1	1	0	0	0	0	0	1	1	0

Answer:-----

Student name: _____

Now for a negative mantissa and exponent:

-1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	-32	16	8	4	2	1
1	1	0	1	1	0	1	0	0	0	1	1	1	0	1	0

Stage 1 - complement the mantissa:

1.101101000 becomes 0.0 1 0 0 1 1 0 0 0, that is - 0.010011000 in “ordinary” binary.

Stage 2 - shift the mantissa, the exponent is -6, so move the binary point 6 places to the left:

- 0.010011000 becomes - 0.000000010011000

or something quite small $1/256 + 1/2048 + 1/4096 = (16+2+1)/4096 = 0.00415039$ -ish.

Convert the following to decimal equivalents:

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

0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Answers:

1 -----

2 -----

3 -----

4 -----

In the second and third representations above, there are leading zeros following the binary point and therefore less space to store any low value bits.

Notice that the first number $2 \frac{1}{128}$ could not be stored with the higher exponents, the LSB of the mantissa would be lost. In order to preserve the maximum possible accuracy, the numbers are normalised (as for the decimal floating point number example given at the start of this section).

Student name: _____

Because we are using two's complement binary fractions, normalising positive and negative numbers is different. Positive is not so bad: the MSB (excluding the sign bit) must be a 1 (ie no trailing zeros after the point).

Thus

0.000111100 would be expressed as 0.111100000

With a corresponding shift in the exponent:

0.000111100 000000 then becomes 0.111100000 111101

an exponent of -3.

For negative numbers there must be no leading 1's to the left of the point, the MSB must be a zero (which is the significant bit in two's complement):

1111100100 would be expressed as 1.001000000

(Why? Leading 1's in a two's complement negative number may be ignored in just the same way that leading zeros may be ignored in a positive number).

This requires a shift of 5 places as shown below:

1111100100 000000 then becomes 1.001000000 000101

To confirm this, complement the mantissa and then shift it 5 places to the right. Write the binary number resulting here:

(if you got it right, the decimal value ought to be 28):

Quick Quiz:

Which of the following decimal numbers link with which normalised binary numbers:

123	0110000000111110
0.1875	1000100000111111
-15/32	1011000100000110
-39.5	0111101100000111

Question: What happens if you allocate 5 bits to the mantissa and 8 bits to the exponent?

Accuracy of numbers is.....

Range of numbers is.....

Student name: _____

Range of Normalised Floating Point Binary Numbers

Consider a representation using only 10 bits, 6 for the mantissa and 4 for the exponent: the principal is the same but the numbers are easier to handle mentally.

-1	1/2	1/4	1/8	1/16	1/32	-8	4	2	1

What is the largest positive number that can be represented? The minimum positive number will require the smallest positive mantissa (normalised, don't forget) and the largest negative exponent.

Binary:.....

Decimal:.....

What is the smallest positive number that can be represented?

Binary:.....

Decimal:.....

What is the largest negative number that can be represented?

Binary:.....

Decimal:.....

What is the smallest negative number that can be represented?

Binary:.....

Decimal:.....

(answers 124, 0.001953125, -128, -0.002075195313-ish)

Integer and Floating Point Representations Compared

The advantages of integer representation are:

- Complete accuracy of storage (as long as the number fits in the number of bits allocated).
- Large range is possible with enough bits.
- Overflow is easily recognised and trapped by programmers.

The advantages of floating point representation are:

- A larger range of numbers can be stored.
- Fractional numbers can be stored.

Floating point errors can also accumulate especially in long and complicated calculations leading to a loss of precision in output.

Student name: _____

Errors

Consider the addition of floating point numbers:

If the numbers are smaller, or use fewer bits we may be simply able to convert floating point numbers to a fixed point representation and then add them in the traditional way:

For example, using 12-bits (8 for a fractional two's complement mantissa and 4 for the two's complement integer exponent) add the following two numbers:

$$0.1001000\ 0010 + 0.1111000\ 0100$$

Converting the numbers from normalised form to fixed point gives

010.01000

and

01111.000

respectively, now we can add:

first			0	1	0	.	0	1	0	0	0
second	0	1	1	1	1	.	0	0	0		
result	1	0	0	0	1	.	0	1	0	0	0

To normalise we move the decimal point 5 places to the left, giving an exponent of 5, so the answer is:
0.1000101 0101

Do these: (Convert them all to decimal to check your answers!)

$$0.1011010\ 0110 + 0.1110100\ 0101$$

$$0.1001001\ 1110 + 1.0111000\ 0010$$

$$1.1001000\ 1011 + 1.0111000\ 1100$$

Student name: _____

(Do any of these involve inaccurate results?)

Adding two larger magnitude floating point numbers requires the following steps:

- Normalize the bigger (in magnitude) number.
- Change the other number so that the two exponents are the same
- Add (or subtract) the mantissas
- Normalize the result

As an algorithm this might be expressed as follows:

```

If the exponents of the two numbers are not equal Then
    For the number with the smaller exponent
        Repeat
            Shift mantissa one place to the right
            Increase exponent by 1
        Until the exponents are equal
    Add the mantissas

If the addition results in a carry from the MSB Then
    Shift this carry bit into the mantissa
    Increase the exponent by 1

Normalize the result
    
```

For example, using a 10-bit mantissa with a 6-bit exponent, where it is very difficult to convert numbers to a fractional representation:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	-32	16	8	4	2	1
1	0	1	1	0	1	1	0	1	1	0	0	1	1	1	1

Just so we can check the correct result later, let's convert this binary value to its decimal equivalent.

The bit pattern of the mantissa 1011011011 can be converted to:

- 0.100100101

As a fraction this is (negative) $1/2 + 1/16 + 1/128 + 1/512$ or $-293/512$.

Expressed as a decimal this is -0.5722656 approximately, as a power of 2 it is -293×2^{-9} , either way we have to multiply it by the exponent of (001111) 215 (32768).

Student name: _____

From here we can work out that its decimal value is:

$$-293 \times 2^{-9} \times 2^{15} = -293 \times 2^6 = -18752$$

or

$$-0.5722656 \times 32768 = -18752$$

Suppose we wish to add 294 to this number, that would be $(1/2 + 1/16 + 1/128 + 1/256) \times 29$ approximately or

0100100110 001001

in our representation. Now we have 001111 for the first exponent and 001010 for the second.

So, for the smaller, we have to repeatedly shift the mantissa one place to the right and increase the exponent by one until it is equal to the larger:

so 0100100110 001001 becomes

0010010010	001010 on the first shift
0001001001	001011 on the second shift
0000100100	001100 on the third shift
0000010010	001101 on the fourth shift
0000001001	001110 on the fifth shift
0000000100	001111 on the sixth shift

And the exponents of the two numbers are now equal. Notice, however, that our original value of 294 has been altered to 256 (32766/128) by this process, ie accuracy has been lost.

In this case, accuracy has been lost as the final bits “fell off” the end of our mantissa – a **truncation** error.

Anyway, now we can add the two mantissas:

$$\begin{array}{r} 1011011011 \\ + 0000000100 \\ \hline 1011011111 \end{array} \quad \text{exp } 001111$$

The final result is

$$\begin{aligned} & -1 + (1/4 + 1/8 + 1/32 + 1/64 + 1/128 + 1/256 + 1/512) \times 2^{15} \\ & \text{or} \\ & -1 + 223/512 \quad \text{or} \quad -289/512 \end{aligned}$$

again, using powers of 2:

$$-289 \times 2^{-9} \times 2^{15} = -289 \times 2^6 = -18496$$

The true answer $(-18752 + 294)$ is -18458 so the process has resulted in some error.

Student name: _____

In real systems these errors are reduced by allocating more bits to the storage of floating point numbers. For example, in Java, 4 bytes (32 bits) are used to store float primitives and 8 bytes are used to store double primitives.

To improve accuracy, intermediate calculations will use more bits than the final stored result (this would help to avoid the error in the calculation shown above).

Situations in which errors can occur

If two numbers are added together such that the result is too big to be stored in the allocated number of bits then we have **overflow**. Usually this causes a run-time error. An example is adding or multiplying two numbers together where the outcome is too big to store.

If the result is too small to be stored (after division or subtraction, for example) in normalised form then the error condition is known as **underflow**. Most systems will simply use zero to represent this condition.

As we have seen **truncation** errors occur when bits are “chopped off” from the end of a number in some process such as adjusting the mantissa of a number to be processed.

Student name: _____

Sample Questions

A control system is being designed for a washing machine using a 16-bit register to store data about the state of the machine, such as whether the door is open or closed, the level of water in the machine and the temperature selected for the wash (an integer).

1. How many bits are required to store the following data:
 - a. The state of the door:.....
 - b. Whether the water level is $\frac{1}{4}$ full, $\frac{1}{2}$ full or $\frac{3}{4}$ full:.....
 - c. The temperature range 0 to 75 degrees C :.....

[3 marks]

2. If the temperature can only be set at 5 degree intervals, from 25 to 75, how many bits are required to store the settings?

.....

[1 mark]

3. In another control system 9 bits are available to store a temperature range of -90 to plus 90 degrees. Show how this could be done using a floating point binary representation while maintaining the maximum possible precision.

--	--	--	--	--	--	--	--	--

[4 marks]

4. If the 9-bits were allocated to storing an integer in two's complement form, what would be the minimum and maximum value that could be stored (you may give your answer as a power of 2).

.....

.....

[2 marks]

Assume there exists a hypothetical decimal computer that stores numbers in a word. A word which stores an integer value can hold eight symbols, one sign (+ or -) in the leftmost position and seven decimal digits. A word which stores a floating-point number can hold eight symbols but the word is divided into two parts. One part can hold three symbols. It is called the exponent and is composed of a sign and two decimal digits. It is located in the leftmost (high order) three positions.

The second part which can hold five symbols is the mantissa. This part is composed of a sign and four decimal digits normalised to the most significant digit and is located in the rightmost (low order) five

Student name: _____

positions. The digits of the mantissa are normalised as indicated in the floating-point example (-2.73479) below.

Examples of integers and floating-point numbers are given below.

-672 is represented as

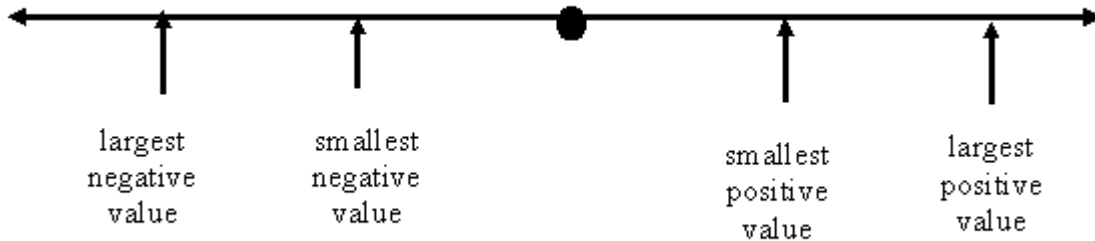
-	0	0	0	0	6	7	2
---	---	---	---	---	---	---	---

-2.73479 is represented as

+	0	1	-	2	7	3	6
---	---	---	---	---	---	---	---

- (a) What is the smallest integer and the largest integer that can be stored? [2 marks]
- (b) What is the largest floating-point number that can be stored? What is the smallest floating-point number greater than 0 that can be stored. In both cases give the answer as a power of 10. [2 marks]

(c)
Given the range of available real values in this computer, copy and complete the real number line below. Indicate the range of negative and positive real values, underflow and overflow, by means of labels.



[8 marks]