AQA Computer Science A-Level

4.5.4 Binary number system

Advanced Notes

Specification:

4.5.4.1 Unsigned binary:

Know the difference between unsigned binary and signed binary

Know that in unsigned binary the minimum and maximum values for a given number of bits, *n*, are 0 and 2^n -1 respectively

4.5.4.2 Unsigned binary arithmetic:

Be able to:

- add two unsigned binary integers
- multiply two unsigned binary integers

4.5.4.3 Signed binary using two's complement:

Know that signed binary can be used to represent negative integers and that one possible coding scheme is two's complement.

Know how to:

- represent negative and positive integers in two's complement
- perform subtraction using two's complement
- calculate the range of a given number of bits, *n*.

4.5.4.4 Numbers with a fractional part:

Know how numbers with a fractional part can be represented in:

- fixed point form in binary in a given number of bits
- floating point form in binary in a given number of bits

Be able to convert for each representation from:

- decimal to binary of a given number of bits
- binary to decimal of a given number of bits

4.5.4.5 Rounding errors:

Know and be able to explain why both fixed point and floating point representation of decimal numbers may be inaccurate.

4.5.4.6 Absolute and relative errors:

Be able to calculate the absolute error of numerical data stored and processed in computer systems.

Be able to calculate the relative error of numerical data stored and processed in computer systems.

Compare absolute and relative errors for large and small magnitude numbers, and numbers close to one.

4.5.4.7 Range and precision:

Compare the advantages and disadvantages of fixed point and floating point forms in terms of range, precision and speed of calculation.

4.5.4.8 Normalisation of floating point form:

Know why floating point numbers are normalised and be able to normalise un-normalised floating point numbers with positive or negative mantissas.

4.5.4.9 Underflow and overflow:

Explain underflow and overflow and describe the circumstances in which they occur.

Signed and unsigned binary

1011

The string of 0s and 1s above is a binary number. It could be either signed or unsigned, and there's no way to work that out. A computer has to be told whether a number is signed or unsigned.

Unsigned binary numbers can only represent positive numbers. Signed binary allows for the representation of negative numbers using binary.

If the number above were unsigned, it could be converted to decimal by assigning place values to each of the digits and adding up the total of the values under which a 1 falls, resulting in the decimal value 11.

Synoptic Link

Converting between binary and decimal is covered in the notes for **number bases**.

Range of unsigned numbers

The range of numbers that can be represented by an unsigned binary number depends on the number of bits available. With one bit, the decimal numbers 0 and 1 can be represented. With two bits, the decimal numbers 0, 1, 2 and 3 can be represented.

There is a pattern to the range of numbers that can be represented by a given number of bits. For n bits, there are 2^n possible permutations of the bits, giving a range of decimal numbers from 0 to 2^n-1.

For example, with eight bits, the decimal numbers 0 to 255 (= 2^8-1) can be represented.

Unsigned binary arithmetic

Adding two unsigned binary integers

When adding unsigned binary numbers, there are four important rules to remember:

1. 0 + 0 + 0 = 0
2. 0 + 0 + 1 = 1
3. 0 + 1 + 1 = 10
4. 1 + 1 + 1 = 11

Make sure you understand these rules and adding binary numbers will be easy.

Example: Add the unsigned binary integers 1011 and 1110.

	1	0	1	1
+	1	1	1	0

Place the two binary numbers above each other so that the digits line up.

	1	0	1	1
+	1	1	1	0
				1

Starting from the least significant bits (the right hand side), add the values in each column and place the total below. For the first column (highlighted), rule 2 from above applies.

	1	0	1	1
+	1	1	1	0
		^{1}	0	1

Move on to the next column. This time rule 3 applies. In the case that the result of addition for a single column is more than one digit, place the first digit of the result in small writing under the next most significant bit's column.

	1	0	1	1
+	1	1	1	0
	^{1}	0^{1}	0	1

On to the next column, where there is a 0, a 1 and a small 1. In this case, rule 3 applies again. Therefore the result is 10. Because 10 is two digits long, the 1 is written in small writing under the next most significant bit's column.

	1	0	1	1
+	1	1	1	0
1	1^{1}	0^{1}	0	1

Moving on to the most significant column where there are three 1s. Rule 4 applies, so the result for this column is 11. The first digit of the result is written under the next most significant bit's column, but it can be written full size as there are no more columns to add.

1	1	0	0	1

Finally, the result is read off from the full size numbers at the bottom of each column. In this case, 1011 + 1110 = 11001.

After carrying out binary addition, it's a good idea to check your answer by converting to decimal if you have time. In this example, 1011_2 (11_{10}) + 1110_2 (14_{10}) = 11001_2 (25_{10}) so we haven't made any mistakes.

Multiplying two unsigned binary integers

When multiplying unsigned binary numbers, write out one of the two numbers starting under each occurrence of a 1 in the second number and then add the contents of the columns.

Example: Multiply the binary numbers 1011 and 1010.

Guide: 1 0 1 1

Choose one of the two numbers as your guide and write it out in columns.

```
      1 0 1 0
    1 0 1 0 0
1 0 1 0 0 0 0
```

Write out the second number under each occurrence of a 1 in the guide number, aligning the least significant bit of the second number with the position of the 1 in the guide number.

```
        1 0 1 0
      1 0 1 0 0
+ 1 0 1 0 0 0 0
-----------------
  1 1¹ 0 1 1 1 0
```

Now perform binary addition on the columns, excluding the guide, using the technique explained earlier.

1 1 0 1 1 1 0

The result of the addition can now be read off from the full size numbers at the bottom of each column. In this case,
1011 × 1010 = 1101110

As with binary addition, binary multiplication can be checked by converting to decimal. In this example, 1011_2 (11_{10}) × 1010_2 (10_{10}) = 1101110_2 (110_{10}) so the multiplication has worked correctly.

Signed binary with two's complement

There are a few different coding schemes that can be used for signed binary. AQA uses one called two's complement to allow for the representation of both positive and negative numbers in binary.

When using two's complement, the most significant bit of a number is given a negative place value. For example, with four bits, the place values would be:

$$-8 \quad 4 \quad 2 \quad 1$$

This allows negative numbers to represented like so:

-8	4	2	1
1	0	1	1

$$-8 + 2 + 1 = -5$$

Subtraction using two's complement

Computers work by adding numbers. In order to perform subtraction, computers in fact add negative numbers. For example, if a computer had to work out the value of $14 - 6$, it would actually work out $14 + (-6)$.

Example: Subtract 12 from 8.

	-16	8	4	2	1
	0	1	0	0	0
+	1	0	1	0	0
	1	1	1	0	0

In five bit two's complement, 8_{10} is 01000_2 and -12_{10} is 10100_2. Five is the minimum number of bits required in order to represent -12.

The two's complement numbers are then added using the same technique for adding that is explained above before the result can be read off as 11100_2.

Checking the result, $-16 + 8 + 4 = -4$ so the calculation is correct.

Range of two's complement numbers

Given a certain number of bits, the range of a two's complement signed binary number includes both positive and negative values.

For example, with four bits, the largest positive value that can be represented is 7 and the most negative number that can be represented is -8 as shown on the right hand side of the page.

More generally, with *n* bits, the range of a two's complement signed binary number is from $2^{n-1}-1$ to -2^{n-1}.

-8	4	2	1
0	1	1	1

4 + 2 + 1 = 7

-8	4	2	1
1	0	0	0

= -8

Numbers with a fractional part

Binary can be used to represent numbers with a fractional part. There are two ways of doing this, one uses fixed point form and the other uses floating point form.

Fixed point binary

When using fixed point with a given number of bits, a specified number of bits are placed before a binary point and the remaining bits fall behind the binary point.

Standard binary place values are used for columns before the binary point, and the columns behind the binary point start at ½, then ¼ and ⅛ etc.

For example, with 8 bits split evenly with four bits before and after the binary point, the number 11.3125_{10} can be represented in binary as 10110101.

8	4	2	1	•	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$
1	0	1	1		0	1	0	1

$$8 + 2 + 1 + \frac{1}{4} + \frac{1}{16} = 11.3125$$

Floating point binary (binary to decimal)

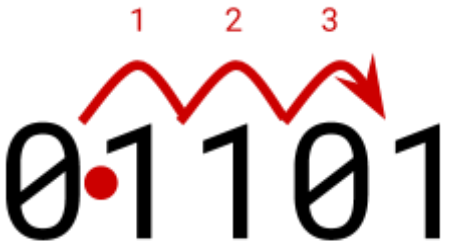
Floating point binary is comparable to scientific notation in that a number is represented as a mantissa and an exponent. In scientific notation, the number 3,100,000 could be written as 3.1×10^6 where 3.1 is the mantissa and 6 is the exponent.

In exam questions on floating point numbers, both the mantissa and exponent will be represented using two's complement signed binary.

In floating point binary, a number of bits are allocated to the mantissa the remaining bits form the exponent. For example, with 5 bits for the mantissa and three for the exponent:

0	1	1	0	1	0	1	1
Mantissa					Exp.		

In order to convert from floating point form to decimal, first convert the exponent to decimal. In this example, $011_2 = 3_{10}$.



Next, treating the number as if there were a binary point between the first and second digits of the mantissa, move the binary point the number of positions specified by the exponent. In this case, the binary point moves three positions.

Now treat the mantissa as a fixed point binary number, with the binary point fixed in the position specified by the exponent.

−8	4	2	1	•	$\frac{1}{2}$
0	1	1	0		1

$$4 + 2 + \frac{1}{2} = 6.5$$

As shown above, the result of converting the floating point binary number 01101011 to decimal (with a five bit mantissa and three bit exponent) is 6.5.

Floating point binary (decimal to binary)

In order to convert a decimal number to a floating point binary number, you must first convert your decimal number to fixed point binary.

Example: Convert the decimal number 14.625 to floating point binary.

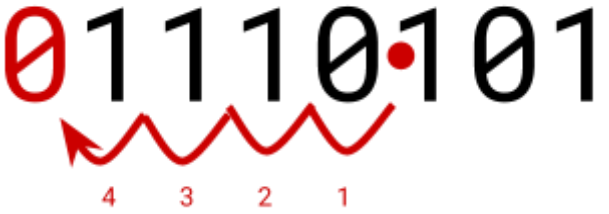
14.625 is smaller than 16, so we need columns before the binary point for -16, 8, 4 and 2. 0.625 = ½ + ⅛ so we need columns after the binary point for ½, ¼ and ⅛.

−16	8	4	2	1	•	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$
0	1	1	1	0		1	0	1

Now that we have a fixed point representation of our decimal number, we have to normalise the number. In order to be normalised, a floating point number must start with 01 (for a positive number) or 10 (for a negative number). As explained above, when converting from floating point to decimal, the binary point is assumed to be between the first two digits in the mantissa. In this case, we must add a leading 0 and move the binary point four positions to the left.

Note

Normalisation is covered in more detail later in this document.



Our exponent must therefore be four, and positive because we moved the decimal point to the left. If the point had moved to the right, our exponent would be negative. Converting this exponent into binary gives us 0100.

We now have our mantissa as 01110101 and our exponent as 0100. Therefore 14.625_{10} = 011101010100_2 in floating point notation with an eight bit mantissa and a four bit exponent.

Rounding errors

There are some decimal numbers that cannot possibly be represented exactly in binary, even with the use of fixed point or floating point notation. A bit like ⅓, which can only be represented in decimal as 0.3333…, there are some numbers which binary can only approximately represent.

There are many numbers that binary cannot accurately represent, one of which is 0.1_{10} which is 0.0001100110011001… in binary. For this reason, both fixed point and floating point representations of decimal numbers may be inaccurate.

Absolute and relative errors

You can calculate absolute and relative errors in order to see how close a particular number is to an actual value.

Absolute error calculation

An absolute error is the actual amount by which a value is inaccurate and can be calculated by finding the difference between the given value and the actual value.

Example: The number 14.6_{10} is represented in fixed point binary as 1110.1_2. Calculate the absolute error.

The binary 1110.1 is equal to 14.5_{10}, so the absolute error can be calculated like so:

$$14.6_{10} - 14.5_{10} = 0.1_{10}.$$

Relative error calculation

A relative error is a measure of uncertainty in a given value compared to the actual value which is relative to the size of the given value. A relative error can be calculated using the formula:

$$relative\ error = \frac{absolute\ error}{actual\ value}$$

This formula gives a relative error as a decimal, but can give a percentage if the result is multiplied by 100.

Example: The number 12.4 is represented in fixed point binary as 1100.011_2. Calculate the relative error as a percentage to four significant figures.

The binary 1100.011 is equal to 12.375_{10}. The absolute error is therefore 0.025. Using the formula, we can calculate the relative error.

$$relative\ error\ \% = \frac{0.025}{12.4} \times 100 = 0.2016\ to\ 4\ s.f.$$

Errors in relation to magnitude

An absolute error of 0.1cm in a measurement of 50m results in a very small relative error of 0.002% but the same absolute error of 0.1cm in a measurement of 1cm results in a much larger relative error of 10%.

Fixed point vs floating point

Although both fixed point and floating point perform the same function of representing numbers with fractional parts in binary, they each have their own relative advantages and disadvantages.

Floating point allows for the representation of a greater range of numbers with a given number of bits than fixed point. This is because floating point can take advantage of an exponent which can be either positive or negative. The number of bits allocated to each part of a floating point number affects the numbers that can be represented. A large exponent and a small mantissa allows for a large range but little precision. In contrast, a small exponent and a large mantissa allows for good precision but only a small range.

In a similar way, the placement of the binary point in fixed point notation determines the range and precision of the numbers that can be represented. A binary point close to the left of a number gives good precision but only a small range of numbers. However, move the binary point to the right and the range is increased while decreasing precision.

Normalisation

Floating point numbers are normalised in order to provide the maximum level of precision for a given number of bits. Normalisation involves ensuring that the a floating point numbers starts with 01 (for a positive number) or 10 (for negative numbers).

Example: Normalise the binary number 000110100101 which is a floating point number with an eight bit mantissa and a four bit exponent.

First, split the number into mantissa and exponent.

0	0	0	1	1	0	1	0	0	1	0	1
Mantissa								Exponent			

Next, adjust the mantissa so that it starts 01 or 10. In this case, because we're dealing with a positive number, we will move all of the bits two places to the left and add zeros to the end of the mantissa. Our new mantissa is 01101000.

Because we've made the mantissa bigger by shifting the bits two positions to the left, we must reduce the exponent by two so as to ensure the same number is still represented. The current exponent is 5_{10} so, subtracting two, the new exponent must be 3_{10} which is 0011_2 in binary.

0	1	1	0	1	0	0	0	0	0	1	1
Mantissa								Exponent			

We now have a mantissa that starts with the digits 01. A positive normalised number.

Underflow and overflow

Underflow and overflow are two types of error that can occur when working with binary.

Underflow

Underflow occurs when very small numbers are to be represented but there are not enough bits available. For example, the number 0.015625_{10} can be represented in fixed point binary as 0000001_2 with one bit before the binary point and six bits after the binary point. However, if only 5 bits are available after the binary point, the number would be represented as 000000_2 which is 0_{10}.

Synoptic Link

Overflow and underflow can also occur in data structures like stacks.

Data structures are covered in **data structures and abstract data types** under **fundamentals of data structures**.

Overflow

Overflow occurs when a number is too large to be represented with the available bits. Overflow is particularly important when using signed binary.

Example: Using 8 bit two's complement signed binary, perform the operation 127 + 1.

First, convert each number to two's complement binary. 127_{10} is 01111111_2 and 1_{10} is 00000001_2. Next, perform binary addition.

	-128	64	32	16	8	4	2	1
	0	1	1	1	1	1	1	1
+	0	0	0	0	0	0	0	1
	1^1	0^1	0^1	0^1	0^1	0^1	0^1	0

As the working above shows, the result of 01111111 + 00000001 in two's complement signed binary is 10000000. Converting these numbers to decimal, we have that 127 + 1 = −128 which obviously isn't right. Overflow has occurred.