1.3 Binary Arithmetic

What you'll learn in Module 1.3

After studying this section, you should be able to:

Understand the rules used in binary calculations.

- Addition.
- Subtraction.
- Use of carry, borrow & pay back.

Understand limitations in binary arithmetic.

- Word length.
- Overflow.

Binary Addition Rules

Arithmetic rules for binary numbers are quite straightforward, and similar to those used in decimal arithmetic. The rules for addition of binary numbers are:

Notice that in Fig. 1.3.1, 1+1 = (1)0 requires a 'carry' of 1 to the next column. Remember that binary $10_2 = 2_{10}$ decimal.

```
0 + 0 =  0
0 + 1 =  1
1 + 0 =  1
1 + 1 = (1)0
```

Fig. 1.3.1 Rules for Binary Addition

Example:

Binary addition is carried out just like decimal, by adding up the columns, starting at the right and working column by column towards the left.

	Decimal	Binary
	2	10
	1 +	01 +
Answer	3	11

Fig. 1.3.2 Simple Binary Addition

Just as in decimal addition, it is sometimes necessary to use a 'carry', and the carry is added to the next column. For example, in Fig. 1.3.3 when two ones in the right-most column are added, the result is 2_{10} or 10_2. The least significant bit of the answer is therefore 0 and the 1 becomes the carry bit to be added to the 1 in the next column.

	Decimal	Binary
	3	0011
	1 +	0001 +
Carry		0110
	4	0100

Fig. 1.3.3 Binary Addition with Carry

Binary subtraction rules

The rules for binary subtraction are quite straightforward except that when 1 is subtracted from 0, a borrow must be created from the next most significant column. This borrow is then worth 2_{10} or 10_2 as a 1 bit in the next column to the left is always worth twice the value of the column on its right.

```
0 - 0 =  0
0 - 1 =  1*
1 - 0 =  1
1 - 1 =  0
```

*After 10_2 is borrowed from next column on left.

Fig. 1.3.4 Rules for Binary Subtraction

Binary Subtraction

The rules for subtraction of binary numbers are again similar to decimal. When a large digit is to be subtracted from a smaller one, a 'borrow' is taken from the next column to the left. In decimal subtractions the digit 'borrowed in' is worth ten, but in binary subtractions the 'borrowed in' digit must be worth 2_{10} or binary 10_2.

After borrowing from the next column to the left, a 'pay back' must occur. The subtraction rules for binary are quite simple even if the borrow and pay back system create some difficulty. Depending where and when you learned subtraction at school, you may have learned a different subtraction method, other than 'borrow and payback', this is caused by changing fashions in education. However any method of basic subtraction will work with binary subtraction but if you do not want to use 'borrow and payback' you will need to apply your own subtraction method to the problem.

Fig. 1.3.5 shows how binary subtraction works by subtracting 5_{10} from 11_{10} in both decimal and binary. Notice that in the third column from the right (2^2) a borrow from the (2^3) column is made and then paid back in the MSB (2^3) column.

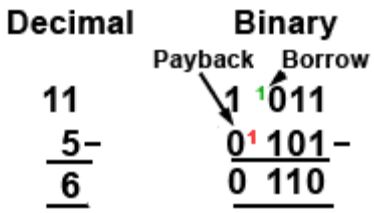


Fig. 1.3.5 Binary Subtraction

Note: In Fig 1.3.5 a borrow is shown as $^1 0$, and a payback is shown as 0^1. Borrowing 1 from the next highest value column to the left converts the 0 in the 2^2 column into 10_2 and paying back 1 from the 2^2 column to the 2^3 adds 1 to that column, converting the 0 to 01_2.

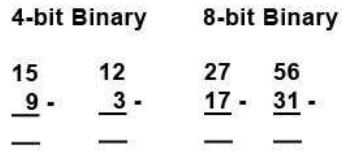
Once these basic ideas are understood, binary subtraction is not difficult, but does require some care. As the main concern in this module is with electronic methods of performing arithmetic however, it will not be necessary to carry out manual subtraction of binary numbers using this method very often. This is because electronic methods of subtraction do not use borrow and pay back, as it leads to over complex circuits and slower operation. Computers therefore, use methods that do not involve borrow. These methods will be fully explained in Number Systems Modules 1.5 to 1.7.

Subtraction Exercise

Just to make sure you understand basic binary subtractions try the examples below on paper. Don't use your calculator, click the image to download and print the exercise sheet. Be sure to show your working, including borrows and paybacks where appropriate. Using the squared paper helps prevent errors by keeping your binary columns in line. This way you will learn about the number systems, not just the numbers.

Limitations of Binary Arithmetic

Now back to ADDITION to illustrate a problem with binary arithmetic. In Fig. 1.3.6 notice how the carry goes right up to the most significant bit.



This is not a problem with this example as the answer 1010_2 (10_{10}) still fits within 4 bits, but what would happen if the total was greater than 15_{10}?

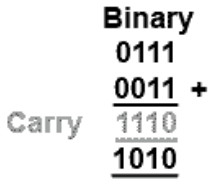


Fig. 1.3.6 Limits of 4 Bit Arithmetic

As shown in Fig 1.3.7 there are cases where a carry bit is created that will not fit into the 4-bit binary word. When arithmetic is carried out by electronic circuits, storage locations called registers are used that can hold only a definite number of bits. If the register can only hold four bits, then this example would raise a problem. The final carry bit is lost because it cannot be accommodated in the 4-bit register, therefore the answer will be wrong.

To handle larger numbers more bits must be used, but no matter how many bits are used, sooner or later there must be a limit. How numbers are held in a computer system depends largely on the size of the registers available and the method of storing data in them, however any electronic system will have a way of overcoming this 'overflow' problem, but will also have some limit to the accuracy of its arithmetic.

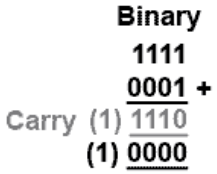


Fig. 1.3.7 The Overflow Problem

1.4 Signed Binary

Signed Binary Notation

What you'll learn in Module 1.4

After studying this section, you should be able to:

Recognise numbers using Signed Binary Notation.

- Identify positive binary numbers.
- Identify negative binary numbers.

Understand Signed Binary arithmetic

- Number representation.
- Advantages of Signed Binary for arithmetic.
- Disadvantages of Signed Binary for arithmetic.

All the binary arithmetic problems looked at in Module 1.3 used only POSITIVE numbers. The reason for this is that it is not possible in PURE binary to signify whether a number is positive or negative. This of course would present a problem in any but the simplest of arithmetic.

There are a number of ways in which binary numbers can represent both positive and negative values, 8 bit systems all use one bit of the byte to represent either + or − and the remaining 7 bits to give the value. One of the simplest of these systems is SIGNED BINARY, also often called 'Sign and Magnitude', which exists in several similar versions, but is commonly an 8 bit system that uses the most significant bit (msb) to indicate a positive or a negative value. By convention, a 0 in this position indicates that the number given by the remaining 7 bits is positive, and a most significant bit of 1 indicates that the number is negative.

For example:

$+45_{10}$ in signed binary is $(0)0101101_2$

-45_{10} in signed binary is $(1)0101101_2$

Note:

The brackets around the msb (the sign bit) are included here for clarity but brackets are not normally used. Because only 7 bits are used for the actual number, the range of values the system can represent is from -127_{10} or 11111111_2, to $+127_{10}$.

A comparison between signed binary, pure binary and decimal numbers is shown in Table 1.4.1. Notice that in the signed binary representation of positive numbers between $+0_{10}$ and $+127_{10}$, all the positive values are just the same as in pure binary. However the pure binary values equivalents of $+128_{10}$ to $+255_{10}$ are now considered to represent negative values −0 to −127.

This also means that 0_{10} can be represented by 00000000_2 (which is also 0 in pure binary and in decimal) and by 10000000_2 (which is equivalent to 128 in pure binary and in decimal).

Table 1.4.1

Binary	Decimal	Signed Binary	
11111111	255	−127	−
11111110	254	−126	
11111101	253	−125	
11111100	252	−124	
↑	↑	↑	
10000011	131	−3	
10000010	130	−2	
10000001	129	−1	
10000000	128	−0	
01111111	127	+127	+
01111111	126	+126	
01111101	125	+125	
01111100	124	+124	
↑	↑	↑	
00000011	3	+3	
00000010	2	+2	
00000001	1	+1	
00000000	0	+0	

Signed Binary Arithmetic

Because the signed binary system now contains both positive and negative values, calculation performed with signed binary arithmetic should be more flexible. Subtraction now becomes possible without the problems of borrow and payback described in Number Systems Module 1.3. However there are still problems. Look at the two examples illustrated in Fig. 1.4.1 and 1.4.2, using signed binary notation.

In Fig. 1.4.1 two positive (msb = 0) numbers are added and the correct answer is obtained. This is really no different to adding two numbers in pure binary as described Number Systems Module 1.3.

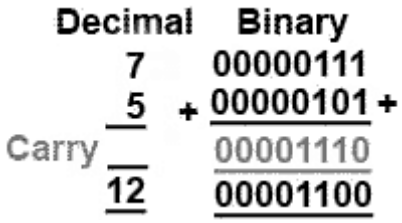


Fig. 1.4.1 Adding Positive Numbers in Signed Binary

In Fig. 1.4.2 however, the negative number −5 is added to +7, the same action in fact as SUBTRACTING 5 from 7, which means that subtraction should be possible by merely adding a negative number to a positive number. Although this principle works in the decimal version the result using signed binary is 10001100_2 or -12_{10} which of course is wrong, the result of 7 − 5 should be +2.

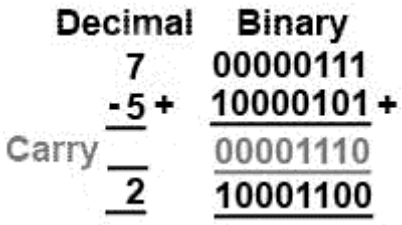


Fig. 1.4.2 Adding Positive & Negative Numbers in Signed Binary

Although signed binary can represent positive and negative numbers, if it is used for calculations, some special action would need to be taken, depending on the sign of the numbers used, and how the two values for 0 are handled, to obtain the correct result. Whilst signed binary does solve the problem of REPRESENTING positive and negative numbers in binary, and to some extent carrying out binary arithmetic, there are better sign and magnitude systems for performing binary arithmetic. These systems are the ONES COMPLEMENT and TWOS COMPLEMENT systems, which are described in Number Systems Module 1.5.

1.5 Ones and Twos Complement

What you'll learn in Module 1.5

After studying this section, you should be able to:

Understand ones complement notation.

- Sign bit.
- Value range.
- Ones complement arithmetic.
- End around carry.

Understand ones complement notation.

- Additive inverse
- Twos complement addition.
- Twos complement subtraction.
- Negative results
- Overflow situations.
- Flag registers.

Ones Complement

The complement (or opposite) of +5 is −5. When representing positive and negative numbers in 8-bit ones complement binary form, the positive numbers are the same as in signed binary notation described in Number Systems Module 1.4 i.e. the numbers 0 to +127 are represented as 00000000_2 to 01111111_2. However, the complement of these numbers, that is their negative counterparts from −128 to −1, are represented by 'complementing' each 1 bit of the positive binary number to 0 and each 0 to 1.

For example:

$+5_{10}$ is 00000101_2 and

-5_{10} is 11111010_2

Notice in the above example, that the most significant bit (msb) in the negative number -5_{10} is 1, just as in signed binary. The remaining 7 bits of the negative number however are not the same as in signed binary notation. They are just the complement of the remaining 7 bits, and these give the value or magnitude of the number.

The problem with signed the binary arithmetic described in Number Systems Module 1.4 was that it gave the wrong answer when adding positive and negative numbers. Does ones complement notation give better results with negative numbers than signed binary?

Fig. 1.5.1 shows the result of adding −4 to +6, using ones complement, this is the same as **subtracting** +4 from +6, so it is crucial to arithmetic.

The result, 00000001_2 is 1_{10} instead of 2_{10}.

This is better than subtraction in signed binary, but it is still not correct. The result should be $+2_{10}$ but the result is +1 (notice that there has also been a carry into the none existent 9th bit).

Decimal	Binary
+6	00000110
-4 +	11111011 +
Carry	(1)11111100
+2	00000001

Fig. 1.5.1 Adding Positive & Negative Numbers in Ones Complement

Fig. 1.5.2 shows another example, this time adding two negative numbers −4 and −3.

Because both numbers are negative, they are first converted to ones complement notation.

$+4_{10}$ is 00000100_2 in pure 8 bit binary, so complementing gives 11111011.

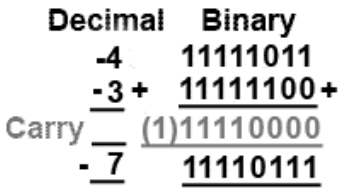


Fig. 1.5.2 Adding Two Negative Numbers in Ones Complement

This is -4_{10} in ones complement notation.

$+3$ is 00000011_{10} in pure 8 bit binary, so complementing gives 11111100.

This is -3_{10} in ones complement notation.

The result of 11110111_2 is in its complemented form so the 7 bits after the sign bit (1110111), should be re-complemented and read as 0001000, which gives the value 8_{10}. As the most significant bit (msb) of the result is 1 the result must be negative, which is correct, but the remaining seven bits give the value of −8. This is still wrong by 1, it should be −7.

End Around Carry

There is a way to correct this however. Whenever the ones complement system handles negative numbers, the result is 1 less than it should be, e.g. 1 instead of 2 and −8 instead of −7, but another thing that happens in negative number ones complement calculations is that a carry is 'left over' after the most significant bits are added. Instead of just disregarding this carry bit, it can be added to the least significant bit of the result to correct the value. This process is called 'end around carry' and corrects for the result −1 effect of the ones complement system.

There are however, still problems with both ones complement and signed binary notation. The ones complement system still has two ways of writing 0_{10} (00000000_2 = +0 and 11111111_2 = -0_2). Additionally there is a problem with the way positive and negative numbers are written. In any number system, the positive and negative versions of the same number should add to produce zero. As can be seen from Table 1.5.1, adding +45 and −45 in decimal produces a result of zero, but this is not the case in either signed binary or ones complement.

Table 1.5.1

	Decimal	Signed Binary	Ones Complement
	+45	00101101	00101101
	−45	10101101	11010010
Binary Sum		11011010	11111111
Decimal Sum	0_{10}	-90_{10}	-127_{10}

This is not good enough, however there is a system that overcomes this difficulty and allows correct operation using both positive and negative numbers. This is the Twos Complement system.

Twos Complement Notation

Twos complement notation solves the problem of the relationship between positive and negative numbers, and achieves accurate results in subtractions.

To perform binary subtraction the twos complement system uses the technique of complementing the number to be subtracted. In the ones complement system this produced a result that was 1 less than the correct answer, but this could be corrected by using the 'end around carry' system. This

still left the problem that positive and negative versions of the same number did not produce zero when added together.

The twos complement system overcomes both of these problems by simply adding one to the ones complement version of the number **before** addition takes place. The process of producing a negative number in Twos Complement Notation is illustrated in Table 1.5.2.

Table 1.5.2	
Producing a Twos Complement Negative Number	
+5 in 8-bit binary (or 8-bit Signed Binary) is	00000101
Complementing to produce the Ones Complement	11111010
With 1 added	1
So -5 in Twos Complement is	11111011

This version of −5 now, not only gives the correct answer when used in subtractions but is also the additive inverse of +5 i.e. when added to +5 produces the correct result of 0, as shown in Fig. 1.5.3

Note that in twos complement the (1) carry from the most significant bit is diskarded as there is no need for the 'end around carry' fix.

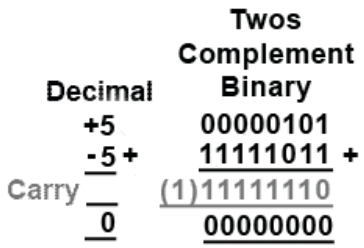


Fig. 1.5.3 Adding a Number to its Twos Complement Produces Zero

With numbers electronically stored in their twos complement form, subtractions can be carried out more easily (and faster) as the microprocessor has simply to add two numbers together using nearly the same circuitry as is used for addition.

6 − 2 = 4 is the same as (+6) + (−2) = 4

Twos Complement Examples

Note: When working with twos complement it is important to write numbers in their full 8 bit form, since complementing will change any leading 0 bits into 1 bits, which will be included in any calculation. Also during addition, carry bits can extend into leading 0 bits or sign bits, and this can affect the answer in unexpected ways.

Twos Complement Addition

Fig 1.5.4 shows an example of addition using 8 bit twos complement notation. When adding two positive numbers, their sign bits (msb) will both be 0, so the numbers are written and added as a pure 8-bit binary addition.

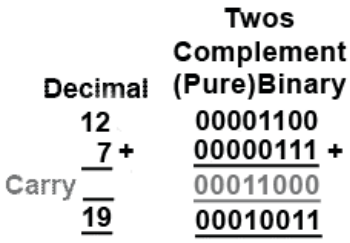


Fig. 1.5.4 Adding Positive Numbers in Twos Complement

Twos Complement Subtraction

Fig.1.5.5 shows the simplest case of twos complement subtraction where one positive number (the subtrahend) is subtracted from a larger positive number (the minuend). In this case the minuend is 17_{10} and the subtrahend is 10_{10}.

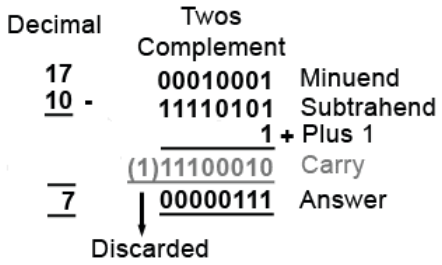


Fig. 1.5.5 Subtracting a Positive Number from a Larger Positive Number

Because the minuend is a positive number its sign bit (msb) is 0 and so it can be written as a pure 8 bit binary number.

The subtrahend is to be subtracted from the minuend and so needs to be complemented (simple ones complement) and 1 added to the least significant bit (lsb) to complete the twos complement and turn +10 into −10.

When these three lines of digits, and any carry 1 bits are added, remembering that in twos complement, any carry from the most significant bit is diskarded. The answer (the difference between 17 and 10) is $00000111_2 = 7_{10}$, which is correct. Therefore the twos complement method has provided correct subtraction by using only addition and complementing, both operations that can be simply accomplished by digital electronic circuits.

Subtraction with a negative result

Some subtractions will of course produce an answer with a negative value. In Fig. 1.5.6 the result of subtracting 17 from 10 should be -7_{10} but the twos complement answer of 11111001_2 certainly doesn't look like -7_{10}. However the sign bit is indicating correctly that the answer is negative, so in this case the 7 bits indicating the value of the negative answer need to be 'twos complemented' once more to see the answer in a recognisable form.

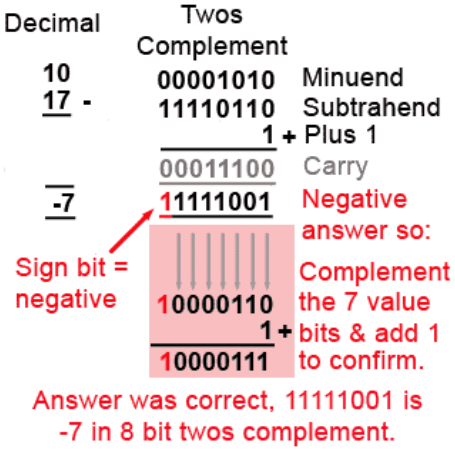


Fig. 1.5.6 Subtraction Producing a Negative Result

When the 7 value bits are complemented and 1 is added to the least significant bit however, like magic, the answer of 10000111_2 appears, which confirms that the original answer was in fact −7 in 8 bit twos complement form.

It seems then, that twos complement will get the right answer in every situation?

Well guess what − it doesn't! There are some cases where even twos complement will give a wrong answer. In fact there are four conditions where a wrong answer may crop up:

1. When adding large positive numbers.

2. When adding large negative numbers.

3. When subtracting a large negative number from a large positive number.

4. When subtracting a large positive number from a large negative number.

The problem seems to be with the word 'large'. What is large depends on the size of the digital word the microprocessor uses for calculation. As shown in Table 1.5.3, if the microprocessor uses an 8-bit word, the largest positive number that can appear in the problem OR THE RESULT is $+127_{10}$ and the largest negative number will be -128_{10}. The range of positive values appears to be 1 less than the negative range because 0 is a positive number in twos complement and has only one occurrence (00000000_2) in the whole range of 256_{10} values.

Table 1.5.3

Decimal	8-bit Twos Complement	
+127	01111111	+
+126	01111110	
+125	01111101	
↑	↑	
+2	00000010	
+1	00000001	
0	00000000	
-1	11111111	-
-2	11111110	
↑	↑	
-126	10000010	
-127	10000001	
-128	10000000	

With a 16-bit word length the largest positive and negative numbers will be $+32767_{10}$ and -32768_{10}, but there is still a limit to the largest number that can appear in a single calculation.

Overflow Problems.

Steps can be taken to accommodate large numbers, by breaking a long binary word down into byte sized sections and carrying out several separate calculations before assembling the final answer. However this doesn't solve all the cases where errors can occur.

A typical overflow problem that can happen even with single byte numbers is illustrated in Fig. 1.5.7.

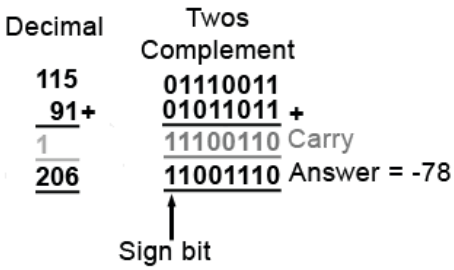


Fig. 1.5.7 Carry Overflows into Sign Bit

In this example, the two numbers to be added (115_{10} and 91_{10}) should give a sum of 206_{10} and converting 11001110_2 to decimal looks like the correct answer (206_{10}), but remember that in the 8 bit twos complement system the most significant bit is the sign of the number, therefore the answer appears to be a negative value and reading just the lower 7 bits gives 1001110_2 or -78_{10}. Although twos complement negative answers are not easy to read, this is clearly wrong, as the result of adding two positive numbers must give a positive answer.

According to the information in Fig 1.5.6, as the answer is negative, complementing the lower 7 bits of 11001110_2 and adding 1 should reveal the value of the correct answer, but carrying out the complement+1 on these bits and leaving the msb unchanged gives 10110010_2 which is -50_{10}. This is nothing like the correct answer of 206_{10} so what has happened?

The 8 bit twos complement notation has not worked here because adding 115 + 91 gives a total greater than +127, the largest value that can be held in 8-bit twos complement notation.

What has happened is that an overflow has occured, due to a 1 being carried from bit 6 to bit 7 (the most significant bit, which is of course the sign bit), this changes the sign of the answer. Additionally it changes the value of the answer by 128_{10} because that would be the value of the msb in pure binary. So the original answer of 78_{10} has 'lost' 128_{10} to the sign bit. The addition would have been correct if the sign bit had been part of the value, however the calculation was done in twos complement notation and the sign bit is not part of the value.

Of course in real electronic calculations, a single byte overflow situation does not usually cause a problem; computers and calculators can fortunately deal with larger numbers than 127_{10}. They achieve this because the microprocessors used are programmed to carry out the calculation in a number of steps, and although each step must still be carried out in a register having a set word

length, e.g. 8 bits, 16 bits etc. corrective action can also be taken if an overflow situation is detected at any stage.

Microprocessors deal with this problem by using a special register called a status register, flag register or conditions code register, which automatically flags up any problem such as an overflow or a change of sign that occurs. It also provides other information useful to the programmer, so that whatever problem occurs; corrective action can be taken by software, or in many cases by firmware permanently embedded within the microprocessor to deal with a range of math problems.

Whatever word length the microprocessor is designed to handle however, there must always be a limit to the word length, and so the programmer must be aware of the danger of errors similar to that described in Fig. 1.5.7.

A typical flag register is illustrated in Fig. 1.5.8 and consists of a single 8-bit storage register located within the microprocessor, in which some bits may be set by software to control the actions of the microprocessor, and some bits are set automatically by the results of arithmetic operations within the microprocessor.

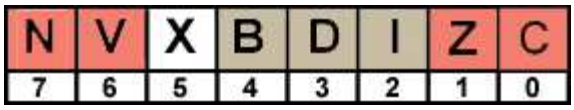


Fig. 1.5.8 Typical 8-bit Flag Register

Typical flags for an 8-bit microprocessor are listed below:

Bit 0 (C) (set by arithmetic result) = 1 Carry has been created from result msb.

Bit 1 (Z) (set by arithmetic result) = 1 Calculation resulted in 0.

Bit 2 (I) (set by software) 1 = Interrupt disable (Prevents software interrupts).

Bit 3 (D) (set by software) 1 = Decimal mode (Calculations are in BCD).

Bit 4 (B) (set by software) 1 = Break (Stops software execution).

Bit 5 (X) Not used on this particular microprocessor.

Bit 6 (V) (set by arithmetic result) = 1 Overflow has occurred (result too big for 8 bits).

Bit 7 (N) (set by arithmetic result) = 1 Negative result (msb of result is 1).

It seems therefore, that the only math that microprocessors can do is to add together two numbers of a limited value, and to complement binary numbers. Well at a basic level this is true, however there are some additional tricks they can perform, such as shifting all the bits in a binary word left or right, as a partial aid to multiplication or division. However anything more complex must be done by software.

Subtraction and Division

Subtraction can be achieved by adding positive and negative numbers as described above, and multiplication in its simplest form can be achieved by adding a number to itself a number of times, for example, starting with a total of 0, if 5 is added to the total three times the new total will be fifteen (or 5 x 3). Division can also be accomplished by repeatedly subtracting (using add) the divisor from the number to be divided until the remainder is zero, or less than the divisor. Counting the number of subtractions then gives the result, for example if 3 (the divisor) is repeatedly subtracted from 15, after 5 subtractions the remainder will be zero and the count will be 5, indicating that 15 divided by 3 is exactly 5.

There are more efficient methods for carrying out subtraction and division using software, or extra features within some microprocessors and/or the use of embedded maths firmware.

1.6 Number Systems Quiz

Try our quiz, based on the information you can find in Digital Electronics Module 1 − Number Systems.

1.

The number .126 x 10^2 written in normalised form represents the number:

a) 12600_{10}

b) 12.6_{10}

c) 10.26_{10}

d) 1111110_2

2.

What is the highest decimal number that can be held in an 8-bit binary register?

a) 127

b) 256

c) 65536

d) 255

3.

What is the decimal equivalent of the number $3A_{16}$?

a) 58

b) 39

c) 310

d) 49

4.

Refer to Fig. 1.7.1.Which of the tables correctly describes the rules of binary addition?

a)

b)

c)

d)

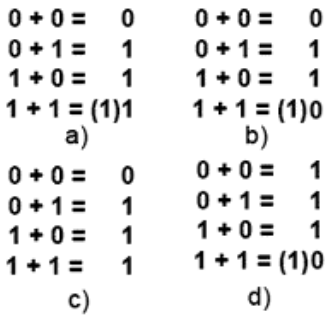


Fig. 1.7.1

5.

What is the 8 bit binary result of $56_{10} - 31_{10}$?

a) 00011001_2

b) 00010101_2

c) 00110001_2

d) 00001101_2

6.

What would be the result of adding 7_{10} and -4_{10} using 8 bit signed binary notation?

a) 10000011_2

b) 00001011_2

c) 10001011_2

d) 00000011_2

7.

What is the widest range of decimal numbers that can be written in 8 bit signed binary notation?

a) −127 to +127

b) −0 to +256

c) −128 to +128

d) −256 to −1

8.

End around carry is used to correct the result of additions in which of the following number systems?

a) 8 bit Signed Binary.

b) 8 bit Ones Complement.

c) 8 bit Twos Complement.

9.

Which of the following Twos Complement binary numbers is equivalent to -75_{10}?

a) 11001011

b) 01001100

c) 11001100

d) 10110101