Module 1

Digital Electronics

1.0 Introduction to Number Systems

What you'll learn in Module 1

Why so many Number Systems?

Ask most people what the most commonly used number system is, and they would probably reply (after a bit of thought), the decimal system. But actually many number systems, and counting systems are used, without the users thinking much about it. For example clocks and compasses use the ancient Babylonian number system based on 60 rather than the decimal system based on 10. Why? Because 60 is easier to divide into equal segments, it can be evenly divided by 1,2,3,4,5,6,12,10,15, 20 and 30. This is much better for applications such as time, or degrees of angle than a base of 10, which can only be divided into equal parts by 1, 2 and 5.

Many counting systems are ancient in origin and are still in use because they are useful for particular purposes.

Using the decimal system it is easy to count up to ten fingers, using just the fingers on two hands. In northern Britain farmers, for centuries, used an ancient Celtic counting system, based on 20 (also called a score), to count their animals, and its use still persisted even into the second half of the twentieth century.

The binary system, based on 2, is just another special number system, and is used by digital electronic devices because digital circuits work on an electrical ‘on or off’ two state system, a number system based on 2 is therefore much easier for electronic devices to use. However binary is not a natural choice for human counting or calculation.

This module explains how binary, and some other number systems used in electronics work, and how computers and calculators use different forms of binary to carry out calculations.

1.1 Number Systems in Electronics

What you'll learn in Module 1.1

After studying this section, you should be able to:

Know the base values of commonly used number systems.

- Decimal
- Binary.
- Octal.
- Hexadecimal.

Understand methods for extending the scope of number systems.

- Exponents.
- Floating point notation.
- Normalised form.

Know how numerical values may be stored in electronic systems

- Bits.
- Bytes.
- Words.
- Registers.

Number Systems

Most number systems follow a common pattern for writing down the value of a number:

A fixed number of values can be written with a single numerical character, then a new column is used to count how many times the highest value in the counting system has been reached. The number of numerical values the system uses is called the base of the system. For example, the decimal system has 10 numerical characters and so has a base of 10:

0 1 2 3 4 5 6 7 8 9

For writing numbers greater than 9 a second column is added to the left, and this column has 10 times the value of the column immediately to its right.

Because number systems commonly used in digital electronics have different base values to the decimal system, they look less familiar, but work in essentially the same way.

Decimal, (base 10)

Decimal has ten values 0 to 9. If larger values than 9 are needed, extra columns are added to the left. Each column value is ten times the value of the column to its right. For example the decimal value twenty-two is written 22 (2 tens + 2 ones).

Binary, (base 2)

Binary has only two values 0 and 1. If larger values than 1 are needed, extra columns are added to the left. Each column value is now twice the value of the column to its right. For example the decimal value three is written 11 in binary (1 two + 1 one).

Octal, (base 8)

Octal has eight values 0 to 7. If larger values than 7 are needed, extra columns are added to the left. Each column value is now 8 times the value of the column to its right. For example the decimal value twenty-seven is written 33 in octal (3 eights + 3 ones).

Hexadecimal, (base 16)

Hexadecimal has sixteen values 0 to 15, but to keep all these values in a single column, the 16 values (0 to 15) are written as 0 to F, using the letters A to F to represent numbers 10 to 15, so avoiding the use of a second column. Again, if higher values than 15 (F in hexadecimal) are needed, extra columns to the left are used. Each column value is sixteen times that of the column to its right. For example the decimal value sixty-eight is written as 44 in hexadecimal (4 sixteens + 4 ones).

The reason for these differences is because each system has a different base, and the column values in each system increase by multiples of the base number as columns are added to the left.

Table 1.1.1				
Some column values of different number systems				
Decimal	1000	100	10	1
Binary	8	4	2	1
Octal	512	64	8	1
Hexadecimal	4096	256	16	1

Because this module describes several different number systems, it is important to know which system is being described. Therefore if there is some doubt which system a number is in, the base of the system, written as a subscript immediately after the value, is used to identify the number system.

For example:

10_{10} represents the decimal value ten. (1 ten + 0 units)

10_2 represents the binary value two. (1 two + 0 units)

10_8 represents the octal value eight. (1 eight + 0 units)

10_{16} represents the hexadecimal value sixteen. (1 sixteen + 0 units)

The System Radix

The base of a system, more properly called the RADIX, is the number of different values that can be expressed using a single digit. Therefore the decimal system has a radix of 10, the octal system has a radix of 8, hexadecimal is radix 16, and binary radix 2.

The range of number values in different number systems is shown in Table 1.1.2, Notice that because the hexadecimal system must express 16 values using only one column, it uses the letters A B C D E & F to represent the numbers 10 to 15.

Table 1.1.2			
Decimal	**Binary**	**Octal**	**Hexadecimal**
(Radix 10)	**(Radix 2)**	**(Radix 8)**	**(Radix 16)**
0	0	0	0
1	1	1	1
2		2	2
3		3	3
4		4	4
5		5	5
6		6	6
7		7	7
8			8
9			9
			A
			B
			C
			D
			E
			F

The Radix Point.

When writing a number, the digits used give its value, but the number is ‘scaled’ by its RADIX POINT.

For example, 456.2_{10} is ten times bigger than 45.62_{10} although the digits are the same.

Notice also that when using multiple number systems, the term ‘RADIX point’ instead of ‘DECIMAL point’ is used. When using decimal numbers, a decimal point is used, but if a different system is used, it would be wrong to call the point a decimal point, it would need to be called "Binary point" or "Octal point" etc. The simplest way around this is to refer to the point in any system (which will of course have its value labelled with its radix) as the RADIX POINT.

Exponents

A decimal number such as 456.2_{10} can be considered as the sum of the values of its individual digits, where each digit has a value dependent on its position within the number (the value of the column):

Table 1.1.3

Col 2	Col 1	Col 0	Col -1
4 hundreds	+ 5 tens	+ 6 units	+ 2 tenths
(4×10^2)	$+ (5 \times 10^1)$	$+ (6 \times 10^0)$	$+ (2 \times 10_{-1})$
400	+ 50	+ 6	+ 0.2

$$= \mathbf{456.2_{10}}$$

Each digit in the number is multiplied by the system radix raised to a power depending on its position relative to the radix point. This is called the **EXPONENT**. The digit immediately to the left of the radix point has the exponent 0 applied to its radix, and for each place to the left, the exponent increases by one. The first place to the right of the radix point has the exponent -1 and so on, positive exponents to the left of the radix point and negative exponents to the right.

This method of writing numbers is widely used in electronics with decimal numbers, but can be used with any number system. Only the radix is different.

$$\text{Hexadecimal exponents } 98.2_{16} = (9 \text{ x } 16^1) + (8 \text{ x } 16^0) + (2 \text{ x } 16^{-1})$$

$$\text{Octal exponents } 56.2_8 = (5 \text{ x } 8^1) + (6 \text{ x } 8^0) + (2 \text{ x } 8^{-1})$$

$$\text{Binary Exponents } 10.1_2 = (1 \text{ x } 2^1) + (0 \text{ x } 2^0) + (1 \text{ x } 2^{-1})$$

When using your calculator for the above examples you may find that it does not like radix points in anything other than decimal mode. This is common with many electronic calculators.

Floating Point Notation

If electronic calculators cannot use radix points other than in decimal, this could be a problem. Fortunately for every problem there is a solution. The radix exponent can also be used to eliminate the radix point, without altering the value of the number. In the example below, see how the value remains the same while the radix point moves. It is all done by changing the radix exponent.

$$102.6_{10} = 102.6 \text{ x } 10^0 = 10.26 \text{ x } 10^1 = 1.026 \text{ x } 10^2 = .1026 \text{ x } 10^3$$

The radix point is moved one place to the left by increasing the exponent by one.

It is also possible to move the radix point to the right by decreasing the exponent. In this way the radix point can be positioned wherever it is required - in any number system, simply by changing the exponent. This is called FLOATING POINT NOTATION and it is how calculators handle decimal points in calculations.

Normalised Form

By putting the radix point at the front of the number, and keeping it there by changing the exponent, calculations become easier to do electronically, in any radix.

Electronic storage of numbers

A number written (or stored) in this way, with the radix point at the left of the most significant digit is said to be in NORMALISED FORM. For example $.11011_2$ x 2^3 is the normalised form of the binary number $110.11_{2.}$ Because numbers in electronic systems are stored as binary digits, and a binary digit can only be 1 or 0, it is not possible to store the radix point within the number. Therefore the number is stored in its normalised form and the exponent is stored separately. The exponent is then reused to restore the radix point to its correct position when the number is displayed.

In electronics systems a single binary digit is called a bit (short for **B**inary Dig**IT**), but as using a single digit would seriously limit the maths that could be performed, binary bits are normally used in groups.

4 bits = 1 nibble

8 bits = 1 byte

Multiple bytes, such as 16 bits, 32 bits, 64 bits are usually called ‘words’, e.g. a 32 bit word. The length of the word depends on how many bits can be physically handled or stored by the system at one time.

4 Bit Binary Representation

Table 1.1.4

Decimal	MSB	4 Bit Binary		LSB
	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

When a number is stored in an electronic system, it is stored in a memory location having a fixed number of binary bits. Some of these memory locations are used for general storage whilst others, having some special function, are called registers. Wherever a number is stored, it will be held in some form of binary, and must always have a set number of bits. Therefore a decimal number such as 13, which can be expressed in four binary bits as 1101_2 becomes 00001101_2 when stored in an eight-bit register. This is achieved by adding four NON-SIGNIFICANT ZEROS to the left of the most significant '1' digit.

Using this system, a binary register that is n bits wide can hold 2^n values.

Therefore an 8 bit register can hold 2^8 values = 256 values (0 to 255)

A 4 bit register can hold 2^4 values = 16 values (0 to 15)

HOW MANY VALUES CAN A 16 BIT REGISTER HOLD______________________?

Filling the register with non-significant zeros is fine - if the number is smaller than the maximum value the register will hold, but how about larger numbers? These must be dealt with by dividing the binary number into groups of 8 bits, each of which can be stored in a one-byte location, and using several locations to hold the different parts of the total value. Just how the number is split up depends on the design of the electronic system involved.

Summary:

- Electronic systems may use a variety of different number systems, (e.g. Decimal, Hexadecimal, Octal, Binary).
- The number system in use can be identified by its radix (10, 16, 8, 2).
- The individual digits of a number are scaled by the Radix Point.
- The Exponent is the system radix raised to a power dependent on the column value of a particular digit in the number.
- In Floating Point Notation the Radix Point can be moved to a new position without changing the value of the number if the Exponent of the number is also changed.
- In Normalised Form the radix point is always placed to the left of the most significant digit.
- When numbers are stored electronically they are stored in a register holding a finite number of digits; if the number stored has less digits than the register, non-significant zeros are added to fill spaces to the left of the stored number. Numbers containing more digits than the register can hold are broken up into register sized groups and stored in multiple locations.

1.2 Converting Between Number Systems

What you'll learn in Module 1.2

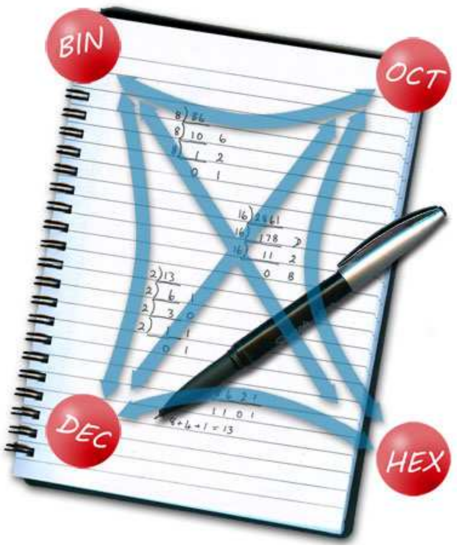
After studying this section, you should be able to:

Convert numerical data between number systems.

- Decimal.
- Binary.
- Octal.
- Hexadecimal.

Understand the relationships between number systems used in digital electronics.

- Decimal fractions
- Decimal & hexadecimal.



It is often necessary to convert values written in one number system to another. The simplest way is to reach for your calculator or use a conversion app. from the web. That is fine, but converting a number in this way does not help you to understand the way each number system works and how different systems are related. The purpose of this module is to explain just that, and to get you to carry out some simple conversions so that you can not only convert between number systems, but also understand how the conversion process works. There are various ways to tackle conversions without a calculator; once the conversion methods are learned, the only skills needed are the ability to multiply and divide, and to add together a few numbers.

Conversion from any system to decimal.

The number of values that can be expressed by a single digit in any number system is called the system radix, and any value can be expressed in terms of its system radix.

Octal to Decimal

For example the system radix of octal is 8, since any of the 8 values from 0 to 7 can be written as a single digit.

Convert 126_8 to decimal.

Using the values of each column, (which in an octal integer are powers of 8) the octal value 126_8 can also be written as:

$(1x8^2) + (2x8^1) + (6 \times 8^0)$

As ($8^2 = 64$), ($8^1 = 8$) and ($8^0 =1$), this gives a multiplier value for each column.

Multiply the digit in each column by the column multiplier value for that column to give:

1x64 = 64   2x8 =16   6x1 = 6

Then simply add these results to give the decimal value.

$64 + 16 + 6 = 86_{10}$

Therefore $126_8 = 86_{10}$.

The same method can be used to convert binary number to decimal:

Convert 1101_2 to decimal.

$= (1x2^3)+(1x2^2)+(0x2^1) +(1x2^0)$

$= 8 + 4 +0 +1$

$= 13_{10}$ **Therefore $1101_2 = 13_{10}$**

Using the same method to convert hexadecimal to decimal.

Convert $B2D_{16}$ to decimal.

$= (Bx16^2)+(2x16^1)+(Dx16^0)$

$= (11x16^2)+(2x16^1)+(13x16^0)$

$= 2816 + 32 +13$

$= 2861_{10}$

Therefore $B2D_{16} = 2861_{10}$.

The same method can be used to convert any system to decimal.

Try these conversions to decimal WITHOUT YOUR CALCULATOR.

110_2 67_8 AFC_{16} FC_{16}

How do you know if your answer is correct?
Convert your decimal answer back into its original format.

Converting from Decimal to any Radix

To convert a decimal integer number (a decimal number in which any fractional part is ignored) to any other radix, all that is needed is to continually divide the number by its radix, and with each division, write down the remainder. When read from bottom to top, the remainder will be the converted result.

Decimal to Binary

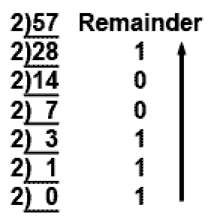
For example, to convert the decimal number 57_{10} to binary:

Divide 57_{10} by the system radix, which when converting to binary is 2. This gives the answer 28, with a remainder of 1.

Continue dividing the answer by 2 and writing down the remainder until the answer = 0

Now simply write out the remainders, starting from the bottom, to give 111001_2

Therefore $57_{10} = 111001_2$



Example 1.2.1 Decimal to Binary Conversion

Decimal to Octal

The same process works to convert decimal to octal, but this time the system radix is 8:

```
8)57  Remainder
8) 7      1  ↑
2) 0      7  |
```

Example 1.2.2 Decimal to Octal Conversion

Therefore $57_{10} = 71_8$

Decimal to Hexadecimal

It also works to convert decimal to hexadecimal, but now the radix is 16:

```
16)57  Remainder
16) 3      9  ↑
16) 0      3  |
```

Example 1.2.3 Decimal to Hexadecimal Conversion

Therefore $57_{10} = 39_{16}$

Numbers with Fractions

It is very common in the decimal system to use fractions; that is any decimal number that contains a decimal point, but how can decimal numbers, such as 34.625_{10} be converted to binary fractions?

In electronics this is not normally done, as binary does not work well with fractions. However as fractions do exist, there has to be a way for binary to deal with them. The method used is to get rid of the radix (decimal) point by NORMALISING the decimal fraction using FLOATING POINT arithmetic. As long as the binary system keeps track of the number of places the radix point was moved during the normalisation process, it can be restored to its correct position when the result of the binary calculation is converted back to decimal for display to the user.

However, for the sake of completeness, here is a method for converting decimal fractions to binary fractions. By carefully selecting the fraction to be converted, the system works, but with many numbers the conversion introduces inaccuracies, a good reason for not using binary fractions in electronic calculations.

Converting the Decimal Integer to Binary

The radix point splits the number into two parts; the part to the left of the radix point is called the INTEGER. The part to the right of the radix point is the FRACTION. A number such as 34.62510 is therefore split into 3410 (the integer), and .62510 (the fraction).

To convert such a fractional decimal number to any other radix, the method described above is used to covert the integer.

```
2)34  Remainder
2)17      0  ↑
2) 8      1  |
2) 4      0  |
2) 2      0  |
2) 1      0  |
   0      1  |
```

Example 1.2.4 Converting the Integer to Binary

So $34_{10} = 100010_2$

Converting the Decimal Fraction to Binary

To convert the fraction, this must be MULTIPLIED by the radix (in this case 2 to convert to binary). Notice that with each multiplication a CARRY is generated from the third column. The Carry will be either 1 or 0 and these are written down at the left hand side of the result. However when each result is multiplied the carry is ignored (don't multiply the carry). Each result is multiplied in this way until the result (ignoring the carry) is 000. Conversion is now complete.

```
             Carry
Fraction             625
 x Radix              x2
  Result  |    1     250
 x Radix  |           x2
  Result  |    0     500
 x Radix  |           x2
  Result  ↓    1     000
```

Example 1.2.5 Converting the Fraction to Binary

For the converted value just read the carry column from top to bottom.

So $0.625_{10} = .101_2$

Therefore the complete conversion shows that $34.625_{10} = 100010.101_2$

However, with binary, there is a problem in using this method, .625 converted easily but many fractions will not. For example if you try to convert .626 using this method you would find that the binary fraction produced goes on to many, many places without a result of exactly 000 being reached.

With some decimal fractions, using the above method will produce carries with a repeating pattern of ones and zeros, indicating that the binary fraction will carry on infinitely. Many decimal fractions can therefore only be converted to binary with limited accuracy. The number of places after the radix point must be limited, to produce as accurate an approximation as required.

Quick Conversions

The most commonly encountered number systems are binary and hexadecimal, and a quick method for converting to decimal is to use a simple table showing the column weights, as shown in Tables 1.2.1a and 1.2.1b.

Converting Binary to Decimal

To convert from binary to decimal write down the binary number giving each column its correct 'weighting' i.e. the value of the columns, starting with a value of one for the right hand (least significant column − or LEAST SIGNIFICANT BIT) column. Giving each column twice the value of the previous column as you move left.

Table 1.2.1a

Bit	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Value (weighting) of each bit	128	64	32	16	8	4	2	1
8 Bit Binary	0	1	0	0	0	0	1	1

Example:

To convert the binary number 01000011_2 to decimal, write down the binary number and assign a 'weighting' to each bit as in Table 1.2.1a

Now simply add up the values of each column containing a 1 bit, ignoring any columns containing 0.

Applying the appropriate weighting to 01000011 gives 256 + 64 + 2 + 1 = 67

Therefore: $01000011_2 = 67_{10}$

Converting Hexadecimal to Decimal

A similar method can be used to quickly convert hexadecimal to decimal, using Table 1.2.1b

The hexadecimal digits are entered in the bottom row and then multiplied by the weighting value for that column.

Adding the values for each column gives the decimal value.

Therefore: $25CB_{16} = 9675_{10}$

Table 1.2.1b

Column	16^3	16^2	16^1	16^0
Value (weighting) of each column	4096	256	16	1
Hex value	2	5	C	B

2 x 4096 = 8192
5 x 256 = 1280
C (12_{10}) x 16 = 192
B (11_{10}) x 1 = 11
9675

Now try some conversions yourself. Use pencil and paper to practice the method, rather than just finding the answer.

Convert:

11010011_2 to decimal.

10111011_2 to decimal.

$34F2_{16}$ to decimal.

$FFFF_{16}$ to decimal.

Check your answer by converting the decimal back to binary or hexadecimal.

Don't use your calculator - you need to learn the method, not just the answer!

Binary and Hexadecimal

Converting between binary and hexadecimal is a much simpler process; hexadecimal is really just a system for displaying binary in a more readable form.

Binary is normally divided into Bytes (of 8 bits) it is convenient for machines but quite difficult for humans to read accurately. Hexadecimal groups each 8-bit byte into two 4-bit nibbles, and assigns a value of between 0 and 15 to each nibble. Therefore each hexadecimal digit (also worth 0 to 15) can directly represent one binary nibble. This reduces the eight bits of binary to just two hexadecimal characters.

Table 1.2.2

Binary	Hex.
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

For example:

11101001_2 is split into 2 nibbles 1110_2 and 1001_2 then each nibble is assigned a hexadecimal value between 0 and F.

The bits in the most significant nibble (1110_2) add up to $8+4+2+0 = 14_{10} = E_{16}$

The bits in the least significant nibble (1001_2) add up to $8+0+0+1 = 9_{10} = 9_{16}$

Therefore $11101001_2 = E9_{16}$

Converting hexadecimal to binary of course simply reverses this process.

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.

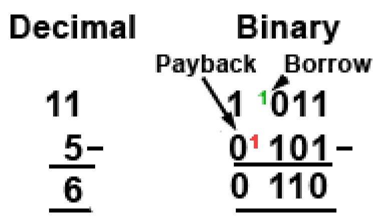


Fig. 1.3.5 Binary Subtraction

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.

4-bit Binary		8-bit Binary	
15	12	27	56
9 -	3 -	17 -	31 -

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}?

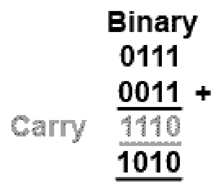


Fig. 1.3.6 Limits of 4 Bit Arithmetic

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.

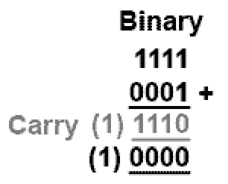


Fig. 1.3.7 The Overflow Problem

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 for example normally 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).

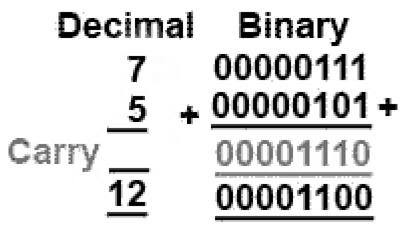
Table 1.4.1

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.

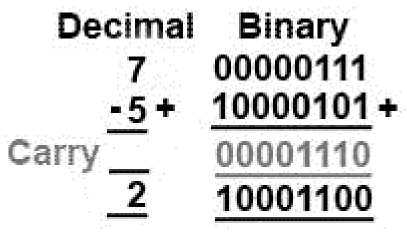
In Fig. 1.4.1 two positive (msb = 0) numbers are added and the correct answer is obtained. This is really no different to adding two numbers in pure binary as described Number Systems Module 1.3.



	Decimal		Binary	
	7		00000111	
	5	+	00000101	+
Carry			00001110	
	12		00001100	

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.



	Decimal		Binary	
	7		00000111	
	-5	+	10000101	+
Carry			00001110	
	2		10001100	

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 −127 to −0, 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, and 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).

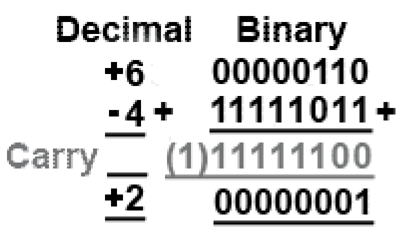


Fig. 1.5.1 Adding Positive & Negative Numbers in Ones Complement

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.

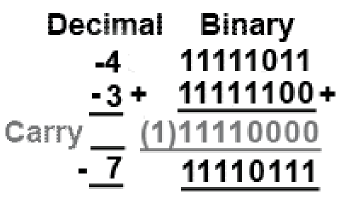


Fig. 1.5.2 Adding Negative Numbers in Ones Complement

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 discarded as there is no need for the ‘end around carry’ fix.

	Decimal	Twos Complement Binary
	+5	00000101
	-5 +	11111011 +
Carry		(1)11111110
	0	00000000

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.

	Decimal	Twos Complement (Pure)Binary
	12	00001100
	7 +	00000111 +
Carry		00011000
	19	00010011

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}.

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.

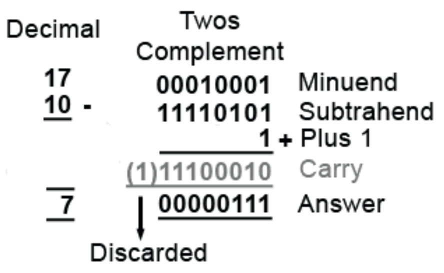


Fig. 1.5.5 Subtracting a Positive Number from a Larger Positive Number

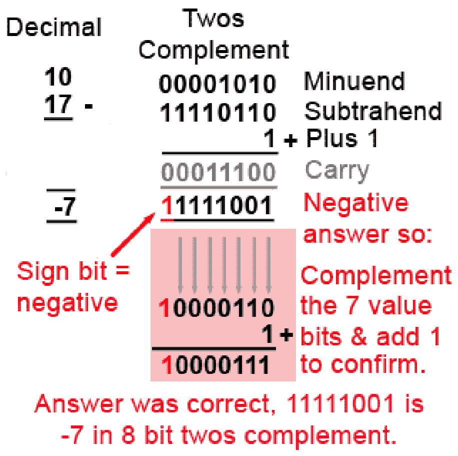
The subtrahend is to be subtracted from the minuend and so needs to be complemented (simple ones complement) and 1 added to the least significant bit (lsb) to complete the twos complement and turn +10 into −10.

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 discarded. 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.

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.



Decimal
Twos Complement
10
17 -
00001010 Minuend
11110110 Subtrahend
1 + Plus 1
00011100 Carry
-7
11111001 Negative answer so:
Sign bit = negative
Complement the 7 value bits & add 1 to confirm.
10000110
1 +
10000111
Answer was correct, 11111001 is -7 in 8 bit twos complement.

Fig. 1.5.6 Subtraction Producing a Negative Result

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.

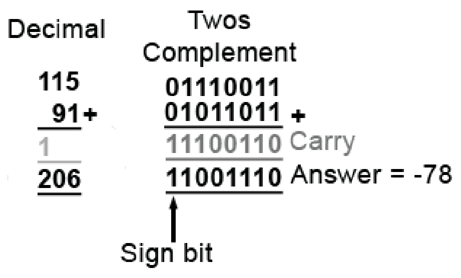


Fig. 1.5.7 Carry Overflows into Sign Bit

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 occurred, 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.

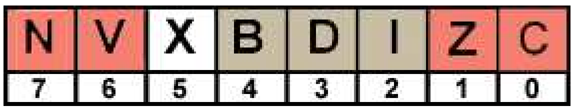


Fig. 1.5.8 Typical 8-bit Flag Register

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

While addition and subtraction can be achieved by adding positive and negative numbers as described above, this does not include the other basic forms of mathematics, multiplication and division. Multiplication in its simplest form can however 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 Binary Codes

What you'll learn in Module 1.6

After studying this section, you should be able to:

Understand binary coded decimal.

- 4 bit BCD codes.
- Converting between binary and BCD.
- Converting between BCD and decimal.
- Compare BCD codes with different weighting.

Understand Gray Code.

- Composition of Gray Code.
- Gray Coded Disks.

Representing Decimal Numbers

When calculations are carried out electronically they will usually be in binary or twos complement notation, but the result will very probably need to be displayed in decimal form. A binary number with its bits representing values of 1, 2, 4, 8, 16 etc. presents problems. It would be better if a particular number of binary bits could represent the numbers 0 to 9, but this doesn't happen in pure binary, a 3 bit binary number represents the values 0 to 7 and 4 bit represents 0 to 15. What is needed is a system where a group of binary digits can represent the decimal numbers 0-9, or ten times those values, 10-90 etc.

To make this possible, binary codes are used that have ten values, but where each value is represented by the 1s and 0s of a binary code. These special 'half way' codes are called BINARY CODED DECIMAL or BCD. There are several different BCD codes, but they have a basic similarity. Each of the ten decimal digits 0 to 9 is represented by a group of 4 binary bits, but in codes the binary equivalents of the 10 decimal numbers do not necessarily need to be in a consecutive order. Any group of 4 bits can represent any decimal value, so long as the relationship for that particular code is known.

In fact any ten of the 16 available four bit combinations could be used to represent 10 decimal numbers, and this is where different BCD codes vary. There can be advantages in some specialist applications in using some particular variation of BCD. For example it may be useful to have a BCD code that can be used for calculations, which means having positive and negative values, similar to the twos complement system, but BCD codes are most often used for the display of decimal digits. The most commonly encountered version of BCD binary code is the BCD_{8421} code. In this version the numbers 0 to 9 are represented by their pure binary equivalents, 4 bits per decimal number, in consecutive order.

Table 1.6.1

	MSB	BCD_{8421}		LSB
Decimal	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

BCD Codes

BCD_{8421} code is so called because each of the four bits is given a 'weighting' according to its column value in the binary system. The least significant bit (lsb) has the weight or value 1, the next bit, going left, the value 2. The next bit has the value 4, and the most significant bit (msb) the value 8, as shown in Table 1.6.1.

So the 8421_{BCD} code for the decimal number 6_{10} is 0110_{8421}. Check this from Table 1.6.1.

For numbers greater than 9 the system is extended by using a second block of 4 bits to represent tens and a third block to represent hundreds etc.

24_{10} in 8 bit binary would be 00011000 but in BCD_{8421} is 0010 0100.

992_{10} in 16 bit binary would be 0000001111100000_2 but in BCD_{8421} is 1001 1001 0010.

Therefore BCD acts as a half way stage between binary and true decimal representation, often preparing the result of a pure binary calculation for display on a decimal numerical display. Although BCD can be used in calculation, the values are not the same as pure binary and must be treated differently if correct results are to be obtained. The facility to make calculations in BCD is included in some microprocessors.

One of the main drawbacks of BCD is that, because sixteen values are available from four bits, but only ten are used, there are several redundant values whichever BCD system is used. This is wasteful in terms of circuitry, as the fourth bit (the 8s column) is under used.

Try some simple conversions between Decimal and BCD_{8421}

321_{10} to BCD_{8421}

65231_{10} to BCD_{8421}

001101110110 BCD_{8421} to decimal.

0011001011000110 BCD_{8421} to decimal.

Display Decoder/Drivers

Depending on the type of display some further code conversion may also be needed. One popular type of decimal display is the 7 segment display used in LED and LCD numerical displays, where any decimal digit is made up of 7 segments arranged as a figure 8, with an extra LED or LCD dot that can be used as a decimal point, as shown in Fig 1.6.1. These displays therefore require 7 inputs, one to each of the LEDs a to g (the decimal point is usually driven separately). Therefore the 4-bit output in BCD must be converted to supply the correct 7 bit pattern of outputs to drive the display.

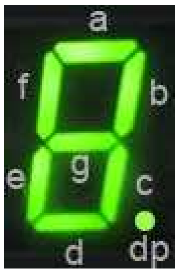


Fig. 1.6.1 Seven Segment Display

The four BCD bits are usually converted (decoded) to provide the correct logic for driving the 7 inputs of the display by integrated circuits such as the HEF4511B BCD to 7-segment decoder/driver from NXP Semiconductors and the 7466 BCD to 7-segment decoder.

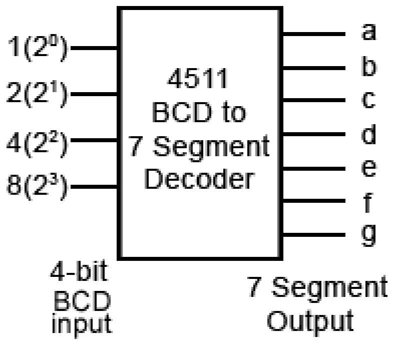


Fig. 1.6.2 Driving a 7 Segment Display

Question

BCD to 7 segment decoders implement a logic truth table such as the one illustrated in Table 1.6.2. There are different types of display implemented by different types of decoder, notice in table 1.6.2 that some of the output digits* may be either 1 or 0 (depending on the IC used). Why would this be, and what effect would it have on the display?

Notice that the 4 bit input to the decoder illustrated in Table 1.6.2 can, in this case, be in either BCD_{8421} or in 4 bit binary as any binary number over 9 will result in a blank display.

Table 1.6.2

BCD Input				7 Segment Output							Display
2^3	2^2	2^1	2^0	a	b	c	d	e	f	g	
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2
0	0	1	1	1	1	1	1	0	0	1	3
0	1	0	0	0	1	1	0	0	1	1	4
0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	0*	0	1	1	1	1	1	6
0	1	1	1	1	1	1	0	0	0*	0	7
1	0	0	0	1	1	1	1	1	1	1	8
1	0	0	1	1	1	1	0*	0	1	1	9
1	0	1	0	0	0	0	0	0	0	0	Blank
1	0	1	1	0	0	0	0	0	0	0	Blank
1	1	0	0	0	0	0	0	0	0	0	Blank
1	1	0	1	0	0	0	0	0	0	0	Blank
1	1	1	0	0	0	0	0	0	0	0	Blank
1	1	1	1	0	0	0	0	0	0	0	Blank

Alternative BCD Codes

Although BCD_{8421} is the most commonly used version of BCD, a number of other codes exist using other values of weighting. Some of the more common variations are shown below. The weighting values in these codes are not randomly chosen, but each has particular merits for specific applications. Some codes are more useful for displaying decimal results with fractions, as with financial data. With others it is easier to assign positive and negative values to numbers. For example with Excess 3 code, 3_{10} is added to the original BCD value and this makes the code 'reflexive', that is the top half of the code is a mirror image and the complement of the bottom half. Other codes are designed to improve error detection in specific systems. Some of these less common BCD codes are shown in Table 1.6.3.

Table 1.6.3

Decimal	7421	5421	5211	2421	Excess 3
0	0000	0000	0000	0000	0011
1	0001	0001	0010	0001	0100
2	0010	0010	0011	0010	0101
3	0011	0011	0101	0011	0110
4	0100	0100	0111	0100	0111
5	0101	1000	1000	1011	1000
6	0110	1001	1010	1100	1001
7	1000	1010	1100	1101	1010
8	1001	1011	1101	1110	1011
9	1010	1100	1111	1111	1100

Gray Code

Binary codes are not only used for data output. Another special binary code that is extensively used for reading positional information on mechanical devices such as rotating shafts is Gray Code. This is a 4-bit code that uses all 16 values, and as the values change through 0-15_{10} the code's binary values change only 1 bit at a time, (see Table 1.6.4). The binary values are encoded onto a rotating disk (Fig. 1.6.3) and as it rotates the light and dark areas are read by optical sensors.

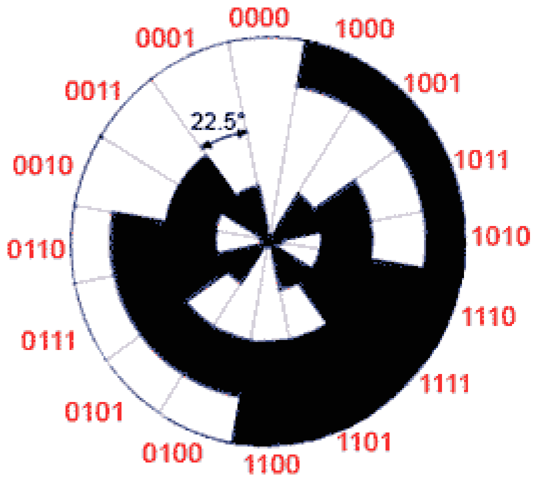


Fig. 1.6.3 Four Bit Gray Code Disk

Table 1.6.4

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

As only one sensor sees a change at any one time, this reduces errors that may be created as the sensors pass from light to dark (0 to 1) or back again. The problem with this kind of sensing is that if two or more sensors are allowed to change simultaneously, it cannot be guaranteed that the data from the sensors would change at exactly the same time. If this happened there would be a brief time when a wrong binary code may be generated, suggesting that the disk is in a different position to its actual position. The one bit at a time feature of Gray Code effectively eliminates such errors. Notice also that the sequence of binary values also rotates continually, with the code for 15 changing back to 0 with only 1 bit changing. With a 4 bit coded disk as illustrated in Fig. 1.6.3, the position is read every 22.5° but with more bits, greater accuracy can be achieved.

1.7 Number Systems Quiz

Try our quiz, based on the information you can find in Digital Electronics Module 1 − Number Systems. Check your answers at http://www.learnabout-electronics.org/Digital/dig17.php and see how many you get right. If you get any answers wrong. Just follow the hints to find the right answer and learn about the number systems used in digital electronics as you go.

1.

Which of the following numbers is equivalent to the normalised number .126 x 10^2 ?

a) 12600_{10}

b) 12.6_{10}

c) 10.26_{10}

d) 1111110_2

2.

Which of the following decimal numbers is equivalent to the highest value that can be held in an 8-bit binary register using unsigned binary?

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.

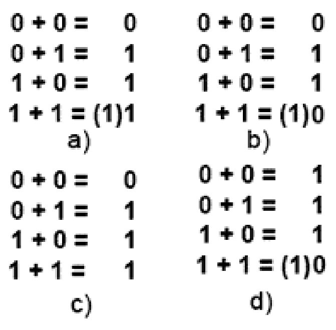
Refer to Fig. 1.7.1.Which of the tables correctly describes the rules of binary addition?

a)

b)

c)

d)



0 + 0 =	0	0 + 0 =	0
0 + 1 =	1	0 + 1 =	1
1 + 0 =	1	1 + 0 =	1
1 + 1 =	(1)1	1 + 1 =	(1)0
a)		b)	
0 + 0 =	0	0 + 0 =	1
0 + 1 =	1	0 + 1 =	1
1 + 0 =	1	1 + 0 =	1
1 + 1 =	1	1 + 1 =	(1)0
c)		d)	

Fig. 1.7.1

5.

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

a) 00011001_2

b) 00010101_2

c) 00110001_2

d) 00001101_2

6.

What is 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.

d) Excess 3 $_{\text{BCD}}$

9.

Which of the following 4 bit Excess 3 numbers is equivalent to 5_{10}?

a) 1101_{bcdxs3}

b) 0010_{bcdxs3}

c) 1000_{bcdxs3}

d) 1010_{bcdxs3}

10.

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

a) 11001011

b) 01001100

c) 11001100

d) 10110101