Chapter 11 - More about translation

11.1 High-level languages

People write programs in programming languages! There are many different languages, and many different ways of classifying, or grouping together, languages. You will have been learning a language on the course. You might be learning Python or Java, for example. Below is an example of a short Python program. Can you work out what it does?

```
print("Hello world")
```

Python, BASIC, C++ and so on are examples of a group of languages known as 'high-level languages'. These are languages that use 'keywords' in programs, words that are very close to English. These types of languages also refer to memory locations using variable names rather than their actual addresses in RAM. For these reasons, high-level languages are easier to learn than low-level languages. They can also be more easily 'read' and so are easier to debug. These kinds of languages, because they are written using keywords, must be changed (or 'translated') into a form that the Central Processing Unit can understand (known as 'machine code') before they can be run. A computer does not 'understand' the word 'print'. It does, however, know how to make the hardware print something out if it receives the machine code instruction for the keyword 'print'. The machine code equivalent of 'print' might look like this: 00010101 00001001 01110010 11110101 01110010.

11.2 Low-level languages

Low-level languages do not use keywords such as 'print', 'pause', 'if' 'while' and so on. They use **mnemonics**. These are codes for instructions, such as DEC (short for 'decrement') or LD (short for 'load'). Mnemonics are designed to be easily remembered (hence the word 'mnemonic'). An example of a short low-level program using mnemonics is shown below.

```
ADD (#344A)
DEC IY
CALL PAGE
LD B, #1195
```

Low-level languages are much closer to the workings of a computer. Often, control programs that require very fast execution speeds are written in a low-level language, known as **assembly**. This is because when they are converted into machine code (using an **assembler**), they produce less machine code than if the equivalent program was written in a high-level language. They therefore run faster! There are also applications that require you to manipulate a computer's hardware in a way that is difficult to do with high-level languages. For example, a programmer might use a low-level language to write a print driver for a new printer and a high-level language to write a sales program that calculates salesmen's commissions.

1) High-level languages are designed to allow a programmer to solve real-world problems.
2) Low-level languages are designed to allow a programmer to manipulate a computer's hardware. In fact, they are sometimes known as 'machine-orientated languages'.

Low-level languages, while quite difficult to learn compared to the newer high-level languages, are a big improvement on what went on before. Programmers had to write in the code that the computer could actually work with and use! Below is an example of part of a machine code program. Imagine writing programs in ones and zeros!!

```
0100 1010 0000 1101
0000 1011 0111 0111
0011 1111 0010 1000
```

There are lots of assembler simulations on the Internet that you can try. Search Google and try out some.

11.3 The need for translators

Whether the source code for a program is written in a low-level language or a high-level language, it must be translated into the code that the CPU can use, the ones and zeros, before the CPU can actually run it. The source code is passed to a special translating program that then converts it into 'object code'. Object code is used interchangeably with the terms 'machine code' or 'executable code' or even 'executable machine code'! We have seen how low level languages use an assembler but what about high level languages?

Some high level languages such as C take the whole source code and translate it **in one go** using a **compiler**. The object code then runs very **quickly**. Other languages such as BASIC take **one line of the source code at a time**. It translates that one line using an **interpreter** and then runs that one line. Then it gets the next line and repeats the process. Interpreted code runs much **slower** than compiled code but it is very useful for writing, developing and **debugging programs** because the program will run correctly up to any error in the program. It will stop if it finds an error. The programmer can then examine the code at that point and re-run it, without the need for re-compilation.

These are the ways most languages translate programs although just to confuse things, some languages such as Java do not produce machine code upon translation but are compiled into an 'intermediate code' called 'bytecode'. This is then converted into machine code by an interpreter when executed. We will discuss this in a later chapter.

11.4 Why companies usually distribute object code and not source code

When you buy an application or a game, for example, you are usually buying just the object code, not the source code.

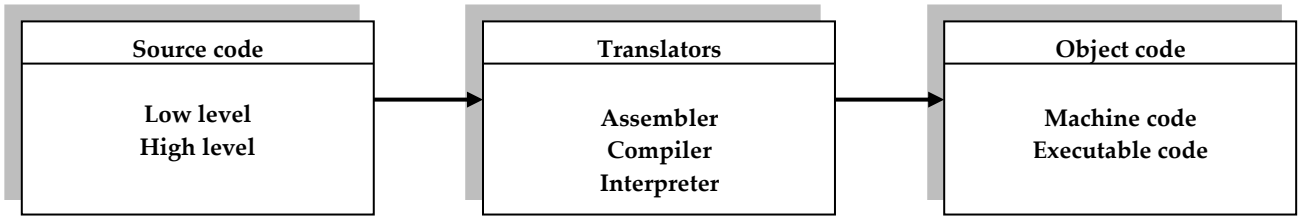
1) If you were sold the source code, you would need to ensure that you had the correct translator program so that you could convert your application into something the computer could understand, into object code! Suppose you didn't have the correct translator? You wouldn't be able to run the program! If you received the object code when you bought an application or game, you could simply execute it. You wouldn't need to translate it.
2) You also you wouldn't need to use up so much of your valuable RAM. This is because you wouldn't need RAM to store the source code, or the translator program, or any temporary storage whilst the translation was being done. You would only need the RAM to store the object code.
3) If you do only have the object code in your possession, however, you would not be able to modify it in any way. It is very difficult to take some object code and reverse engineer it back into source code.
4) The source code itself would be jealously guarded by the company who wrote the application. They would want to protect their copyright. (Writing software is a very expensive business). If they needed to make any changes or updates to the application, they would be made to the source code as required. After any modifications, the source code would need to be translated again and the updated object code made available to users, perhaps via the web.

11.5 Different types of translators

There are three different types of translating program. Which one you would use depends upon the actual language you are writing the source code in. The three types of translators are assemblers, interpreters and compilers. There are also the programming languages which make use of 'intermediate code'.

1) If you wrote a program in a low-level language called Assembly then you would translate the source code into object code with an assembler.
2) If you wrote a program with certain high level languages such as Pascal, C or COBOL, you would translate the source code into object code using a compiler.
3) If you wrote a program with certain high level languages such as BASIC or Perl, you would translate the source code into object code using an interpreter.
4) If you wrote a program with certain high level languages such as VB or JAVA you would translate the source code into an intermediate code using a compiler. The intermediate code would then be run with an interpreter.

We can summarise what happens with the following diagram.



A summary of the translation process.

Q1. Explain the terms machine code, low level language, high level language, source code and object code.
Q2. Why do the programs we write need to be translated?
Q3. How does compilation differ from interpretation?
Q4. Why are interpreted languages good for debugging programs compared to compiled programs?
Q5. Why is the object code for a new computer game distributed to customers rather than the source code?

Chapter 12 - Testing your programs and quality control

12.1 Programming errors

When a programmer writes a program, errors are sometimes made in the code. Finding errors, or 'bugs', in a program is known as 'debugging' a program. Errors can be classified under the three headings.

- Syntax errors.
- Logical errors.
- Run-time errors.

These classifications come about because the first type of error should be picked up by your translator's diagnostics. In other words, when you try to translate your program from source code to object code. The second type of error can be translated successfully and won't actually cause your computer to display an error at all. It's up to the programmer to find these. The third type of error again can be translated but this time will throw up an error when the object code is run.

12.1.1 Syntax errors

Every program has its own set of rules regarding the instructions that can be used and the way that each instruction must be constructed. If a programmer were to break one of those rules, the program would not be translatable into object code. For example, look at the following Python program:

```
Print("Hello world")
```

If we tried to translate this program into object code, our translator program would diagnose a syntax error because the rules of Python haven't been strictly followed. The keyword Print starts with a capital letter. Python's keywords are case sensitive. Print should have started with a small letter p.

12.1.2 Logical errors

When a programmer has written a program that contains logical errors, it can be translated and run. However, the program will produce incorrect results or displays. For example, suppose you had a program line that multiplied two numbers:

```
Result = valueA + valueB
```

This would be translated into object code correctly. When the executable code is run, however, it would produce an incorrect result. This is because the + was used instead of the multiplication symbol. It is up to the programmer to discover these types of bugs by writing a very good test plan to pick them up. If the test plan isn't comprehensive, then the programmer will think that they have written a good program but it will be flawed!

12.1.3 Run-time errors and arithmetic errors

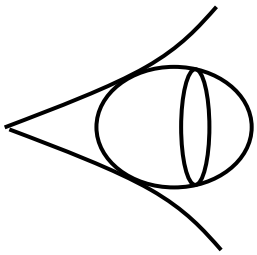
As with logical errors, a program with these types of errors can be translated without an error being discovered. Unlike logical errors, however, the program will stop running and a run-time error will be displayed. A typical example of a run-time error is when a division by zero occurs in a calculation. Dividing by zero is a meaningless calculation. Impossible arithmetic operations that result in errors are also called arithmetic errors. Another run-time error would occur when a program has run out of memory. There is a fixed amount of RAM in any computer. Some of it will be used for the operating system and the rest will be used for applications and data files currently needed by the CPU. If the CPU can't run a program because there is not enough memory available to store temporary values, for example, then it will display a run-time error message and stop working.

12.2 Introduction to test strategies

The aim of any test plan is to make a program fail.

The aim of a good test plan is to try and find bugs in a program. If a program can't be failed and errors can't be found then the programmer can be *reasonably* certain that the program works. Writing test plans that prove a program works is incredibly difficult. When programs are thousands and thousands of lines of code long, it is difficult to test every path through the code, to ensure that every path through it works as intended. Formal test strategies, however, improve the chances of finding bugs.

12.2.1 White box testing (also known as 'glass box testing')



```
IF TEMP < 6 THEN
        WRITE ('Enter sales')
        READ ('SALES')
        TEMP := TEMP + SALES
ELSE
        WRITE ('Enter ID')
        READ (ID)
ENDIF
```

An example of white box testing.

This kind of testing can be done either by the programmers who wrote the code or some people employed specifically to test software. This type of testing is known as 'white box testing' or 'glass box testing'. The test plan is based upon the structure and design of the actual code. The person who writes the test plan can actually see the code. Consider this example to calculate an exam mark comment. You have some pseudo-code with lots of nested IF statements.

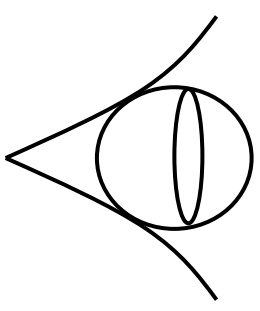
```
IF ITEM < 100 THEN
        PRINT "Poor mark"
ELSE
        IF ITEM < 200 THEN
                PRINT "Could do better"
        ELSE
                IF ITEM < 300 THEN
                        PRINT "GOOD"
                ELSE
                        PRINT "Well done"
                ENDIF
        ENDIF
ENDIF
```

Because the tester can see this code they will know that, to ensure they test every bit of the code (every path through the code), they must try:

- a value less than 100.
- a value greater than or equal to 100 but less than 200.
- a value greater than or equal to 200 but less than 300.
- a value greater than or equal to 300.

They can, of course, only design the tests for white box testing once the code has been written! It's possible for bugs to be discovered in a program years after it was first used because a particularly unusual path through the code was not tested. It is almost impossible to test every path in a big program, especially, for example, when many nested IF statements are used.

12.2.2 Black box testing



An example of black box testing.

Black box testing is so described because the code is viewed as being inside a black box, or unreadable. The tester has no knowledge of the code. This kind of test plan should be written **before** the program is written. This is possible because you are not testing the code as such but testing to see if the program can do what it was written to do, as laid down in a **Requirements Specification**. The focus is on the **functionality** of the program. For this reason, the tests will be written so that data is inputted via the interfaces provided and the outputs observed to see if what was predicted would happen actually does happen.

Whether you are doing black box testing or white box testing, a plan should use a range of data to produce a range of results. The programmer would:

1) Decide what data to use.
2) Decide why you will use that data.
3) Predict the results.
4) Carry out the test.
5) Compare the prediction with the actual results and comment on whether the test passed or failed.

When deciding what data to use, the programmer would consider the following categories of input data:

- Valid data.
- Invalid data.
- Extreme data.
- Mad data.
- Borderline data - bugs lurk in corners!

Valid data in the exam mark comment program on the previous page might include, 34, 150 and 250, for example. Invalid data might include -30. Extreme data might include 560444033. Mad data would be of a completely different data type, 'Gr', for example. Although no code is shown in our program to cope with this invalid data type of input, we would expect there to be some in a real program. We would want to observe an "invalid input" message if this case occurred. Borderline data would include 99, 100, 101, 199, 200, 201, 299, 300 and 301, for example, with observations of the correct output in each case.

12.2.2.1 An example of black box testing

Suppose a subroutine has been written.

- The program asks the user to enter in a test mark between 0 and 100 as an integer.
- It outputs a grade.
- A student must re-sit the test if they get less than 50.
- A pass is from 50 to 70 inclusive.
- A pass with honours is from 71 to 100 inclusive.

You have been given the subroutine by the programmer to test. You do not have knowledge of the actual code. You do, however, know what it does because you have the Requirements Specification. You might draw up the following black box test plan, which also shows the results of carrying out each test.

Data to be used	Why this data has been selected	Prediction of what will be displayed	Actual results	Comment
30	Valid	"Re-sit"	"Re-sit"	Test passed
60	Valid	"Pass"	"Pass"	Test passed
85	Valid	"Honours"	"Honours"	Test passed
-1	Invalid (Negative / borderline)	"Please re-enter data"	"Please re-enter data"	Test passed
0	Valid (Borderline)	"Re-sit"	"Re-sit"	Test passed
1	Valid (Borderline)	"Re-sit"	"Re-sit"	Test passed
49	Valid (Borderline)	"Re-sit"	"Pass"	Test failed
50	Valid (Borderline)	"Pass"	"Pass"	Test passed
51	Valid (Borderline)	"Pass"	"Pass"	Test passed
-345466	Invalid (Extreme)	"Please re-enter data"	"Please re-enter data"	Test passed
87876	Invalid (Extreme)	"Please re-enter data"	"Please re-enter data"	Test passed
4.5	Invalid (Non-integer)	"Please re-enter data"	"Please re-enter data"	Test passed
-67.8	Invalid (Negative non-integer)	"Please re-enter data"	"Please re-enter data"	Test passed
FG	Invalid (Mad)	"Please re-enter data"	"Please re-enter data"	Test passed
???6F	Invalid (Mad)	"Please re-enter data"	"Please re-enter data"	Test passed

An example of a black box test plan and results.

When you completed the tests, you would pass the results back to the programmer. They would need to investigate why the test that used the data 49 failed - it did not produce the expected outcome. They would then re-submit it to you for re-testing.

12.2.3 Alpha testing

Software can be fully or partially written and then tested using black box and white box testing. Whether complete or only partially complete, the code is passed to a restricted audience within the company that produced the software, possibly a Testing Department set up for the purpose of testing software products. They will use it and give feedback to the programmers. They will report any faults they find, make comments about the ease of use of the software and suggest improvements, for example. This kind of testing is known as ‘alpha testing’. It is used to further improve a product and to help track down bugs.

12.2.4 Beta testing

Once a product has been alpha tested, it can then be improved and 'finished off'. It is released to a limited audience in return for their feedback. The audience might include reviewers, special customers and people who write textbooks about software products. This information can be used to further improve the quality of the final product before it is finally released and sold as a finished program to an awaiting public. This kind of testing is known as ‘beta testing’.

12.2.5 Unit testing

Whenever a module of code (or subroutine, or unit, or function, or procedure, or whatever other name you want to give to the block of self-contained code) is written, it needs to be tested. Testing can be done using black or white box testing, as discussed earlier. Whether you use black box or white box testing, testing an individual module of code is known as ‘unit testing’.

12.2.6 Integration testing

Once modules of code are tested using unit testing, they need to be combined and tested together. It is possible that two fully tested modules (tested individually using unit testing) will produce a problem when they are combined and asked to work together. Bringing modules together and checking that there are no unintentional problems is known as ‘integration testing’.

12.3 Identifying programming errors

Once an error in a program has been discovered, whether it is a syntax error, a logical error or a run-time error, whether it has been discovered during white box testing, black box testing, alpha testing or beta testing, the program needs to be debugged! This can sometimes be easier said than done. Fortunately, there are tools to help! Programming languages usually come with debugging programs. These are software tools for the programmer to use to help find bugs.

12.3.1 Translator diagnostics

When some source code is translated, what actually happens is that the source code is passed over to a program called a 'translator'. The translator checks each line in turn against the rules of the language. You will know from experience that programming instructions using keywords must be written in exactly the correct way. If there is any deviation from the rules in the way an instruction is constructed then an error will be reported. This is known as a **syntax error**. An appropriate error message would be displayed on the screen and the location of the error in the program would be highlighted. Unfortunately, the exact position of the error is not always obvious from the information given by the translator diagnostics program and the error messages given can be rather cryptic sometimes. Translator diagnostics do, however, help a programmer locate syntax errors in a program.

12.3.2 Dry run

A section of a program that a programmer suspects of having an error might be 'dry run'. This involves the programmer copying out a list of instructions from the program into a table on paper. This table is known as a ‘trace table’. The programmer then adds columns for any variables that are important for this part of the code. When this has been done, the programmer works through the program on paper, line by line, filling into the trace table the values of variables as they change. By doing this, the programmer can spot the exact position when things start going wrong with the program - when variables suddenly contain unexpected values. Trace tables can be done on computer but it is more usual to do it on paper by hand - programmers seem to be able to find problems more efficiently this way.

12.3.3 Watch and Trace

It is possible for a programmer to tell the computer to display (or 'watch') certain variables in a program. The programmer can then step through the program line-by-line (or 'trace' through the program) watching the variables as they change. This is a little like doing a dry run on paper. This tool, however, is very powerful when used with the breakpoint tool described below.

12.3.4 Breakpoints

Consider a program that has 10000 thousand of lines of code. A programmer trying to track down a fault is confident that the first 8000 lines are fine but suspects the problem lies somewhere after line 8000. They can set a 'breakpoint' at line 8000 and then run the program. The program will run from the beginning of the program right up to line 8000 and will then stop. At that point, the programmer can then examine the contents of the variables they have told the computer to 'watch'. When they have done that they will either have spotted the problem and will set about correcting it or they will set the program running to the next breakpoint. They could also 'watch and trace' from that breakpoint.

12.3.5 Cross-referencing

This debugging tool identifies every place that a particular variable occurs. This is useful, because it allows the programmer to check that each variable has been named correctly and that one variable name hasn't been used for two different things.

12.3.6 Bottom-up testing

When programs are designed using the top-down approach, it means that a big problem is split up into lots of smaller problems and a program, or 'module', is then written for each problem. Each of the modules produced should ideally perform one task. When a program is being tested, the smaller modules are each tested individually first. They are then combined with other modules and they are tested together. These are then combined with even more modules, and tested, until the whole program has effectively been rebuilt and completely tested. While a problem is designed using a top-down approach, testing is done using a bottom-up approach. This makes it easier to find faults. It is easier to find faults in lots of smaller blocks of code and then recombine them than trying to find bugs in one big block of code.

12.4 Good program design techniques

It is very easy to write code so that it is difficult for anyone to follow or understand! It is also possible, with very little extra effort, to write code that is easy to follow and understand. It is important that code can be understood. There may be a bug in the program. It may be that a program needs to be changed - perhaps extra functions need to be added or the law has changed and something in the program needs to be changed. Whatever the reason, if it is difficult to follow how the code works then it will be difficult to modify it. There are a number of simple things that a programmer can do to ensure that a program's code is 'readable'. These can be summarised as top-down programming, comments, variable names, indentation and line spacing.

12.4.1 Top-down programming (also known as 'stepwise refinement')

Programs should be written in modules. Whatever the language being used, it will be easier to understand a program if it has been broken down into small modules, with each of the modules performing ideally just one task and coded up as a standalone piece of code. The modules are effectively standalone units of code that can, however, interact with other modules of code. The approach of splitting up a big problem into a smaller problem is known as 'top-down programming' and this approach aids modular programming. Top-down programming and modular design aid program understanding.

12.4.2 Comments

Code is just that, code. By adding a commentary and explanations throughout a program, it will help someone follow what has been done and how a particular piece of 'clever' code works.

12.4.3 Variable names

Variable names should reflect the item of data they hold. If this is done, then a program can be 'read' that bit easier than if meaningless variable names are used. Consider the following programs, program 1 and program 2:

Program 1	Program 2
Input TaxRate	Input Value1
Input GrossSalary	Input Value2
TaxDue=GrossSalary*TaxRate/100	Value3=Value2*Value1/100
NetSalary=GrossSalary-TaxDue	Value4=Value2-Value3
Input Bonus	Input Value5
NetSalary=Netsalary+Bonus	Value4=Value4+Value5
Print NetSalary	Print Value4

Both programs are the same but Program 1 can be easily read. You quickly lose track of Program 2.

12.4.4 Indentation

Programs are rarely written as a list of instructions. Parts of the code are 'indented' (moved in a few spaces) to emphasise that the code belongs to one of the programming constructions 'selection' or 'iteration'. Code that is indented in this way is far easier for programmers to read and follow than code written as a list of instructions. Which of these two identical programs is easiest to understand and why?

Program 1	Program 2

INPUT ExamMark
IF (ExamMark < 40) THEN
PRINT "You have failed."
ELSE
IF (ExamMark < 60) THEN
PRINT "Pass."
ELSE
IF (ExamMark < 70) THEN
PRINT "Good."
ELSE
IF (ExamMark <80) THEN
PRINT "Well done."
ELSE
PRINT "Outstanding!"
ENDIF
ENDIF
ENDIF
ENDIF
	INPUT ExamMark
	IF (ExamMark < 40) THEN
	PRINT "You have failed."
	ELSE
	IF (ExamMark < 60) THEN
	PRINT "Pass."
	ELSE
	IF (ExamMark < 70) THEN
	PRINT "Good."
	ELSE
	IF (ExamMark <80) THEN
	PRINT "Well done."
	ELSE
	PRINT "Outstanding!"
	ENDIF
	ENDIF
	ENDIF
	ENDIF

12.4.5 Line spacing

Programs should be spaced out. One endless block of code making up a program is harder to read than a program that has been split up into sections by the liberal use of line spaces. For example, functions and procedures should be clearly separated from each other with line spaces. Variable declarations and the main program should be separated from the functions and procedures with line spaces.

12.4.6 Summary

Programming code can be written to aid understanding by using a few simple techniques. These are:

- employing top-down programming and modular design
- using comments throughout a program
- using meaningful variable names
- using indentation
- using line spaces.

Q1. Give an example of a syntax error and describe how syntax errors are found.
Q2. Give an example of a logical error and describe how you find logical errors.
Q3. Give an example of when a run-time error might occur.
Q4. What is the difference between black box and white box testing?
Q5. What is the difference between alpha and beta testing?
Q6. What is meant by unit testing?
Q7. Explain the need for integration testing.
Q8. Describe the tools that can be used to track down a logical error.
Q9. What is meant by 'stepwise refinement'?
Q10. How do programmers make their code as readable as possible?

Chapter 13 - Denary, binary, octal, hex and BCD.

13.1 Introduction

We have already seen in an earlier chapter that the common numbering system we use in our daily lives is called the 'denary' system, or counting using 'base 10'. In practice, what this means is that we count using 10 different digits: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. We have also seen that it isn't just a digit that is important. It is the *position* of the digit relative to the other digits in a number that is also important. For example, consider the digit 6. It is worth one amount if it is on its own. It is worth a different amount in 63, and worth a lot more in 6324, and a lot less in 54.68. In all these examples, the amount that the digit is worth is determined by where exactly it is in relation to the other digits in the number.

Let's look at a number, for example 658, and remind ourselves what we said in a previous chapter. How do we know how much this number is 'worth'? We know because without thinking about it, we judge the digits that make up the number relative to the other digits! When you were at primary school, you may have been helped to learn your numbers by writing down column headings for a particular number, just so you could get used to the idea that position is important. For example, consider what the number 658 means.

Hundreds	Tens	Units	
6	5	8	
(6×10^2)	(5×10^1)	(8×10^0)	
(6 x 100) +	(5 x 10) +	(8 x 1) =	658

How to count in denary.

Of course, when you are working doing maths you (probably) don't write down the headings anymore. You don't need to because you're so used to counting in base 10. Until you are experts in the other counting systems you need to look at, you should get into the habit of writing down the 'worth' of each position.

13.2 The binary number system

Computers make heavy use of the binary counting system. The reason for this is that the electronic circuits that make up a computer are essentially made up of millions and millions of switches. The switches can have two states - on or off. The actual position of any particular switch relative to other switches is also very important. (In fact, a switch can have three states: on, off and 'not connected', but that is beyond this book)!

The base 10 system (denary) uses 10 digits, zero to nine. The base 2 system (binary) uses just 2 digits (or 'bits') zero and one. We are going to start our work by using groups of 8 bits, known as a 'byte'.

Here is one possible allocation of the 'worth' of each bit in my byte.

128	64	32	16	8	4	2	1

The weighting of each bit in a byte.

It is not just whether a bit is a one or a zero that is important. The position of the bit in the whole byte is also important. The bit on the left is 'worth' far more than the bit on the right, for example. This is for no other reason than it is the way I have defined it.

Incidentally, you can see when I used the denary system, there was a pattern to working out the 'worth' of each position. It went 10^3, 10^2, 10^1, 10^0 and so on. (Any number to the power of zero is one.) There is a similar pattern when using base 2.

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

How the 'worth' of each position is calculated in the binary system.

13.2.1 An example of using the binary system

Imagine I want to keep a record of the score of one set of throws in a darts match! The minimum score we need to hold is zero. The maximum is 180, and the score will always be an integer, a whole number i.e. you cannot score 34.5, for example. So now, to represent 180, I would set the bits as shown on the next page. (Note: when I talk about 'setting a bit' I mean that it is made a 'one'. When I talk about resetting a bit, it is made a 'zero').

128	64	32	16	8	4	2	1
1	0	1	1	0	1	0	0

180_{10} in binary.

Incidentally, the maximum number that I can represent using my system is 255. I don't need numbers this high for my darts score, but it is always nice to know what the maximum number you can store is. This is shown next.

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

255_{10} is the maximum number that can be represented in binary.

It's also nice to know what the smallest number is using any number system. The smallest number you can represent here is zero. How you do this is shown below.

128	64	32	16	8	4	2	1
0	0	0	0	0	0	0	0

0_{10} is the minimum number that can be represented in binary.

So with this numbering system I can represent any whole number from zero to 255.

Computers are digital devices! If we had to work in binary all the time, we would soon go mad! The numbers are very long. It takes a while to work them out and it isn't very convenient. There are other numbering systems around that, because of their very close relationship with binary, make them highly suitable. Programmers will sometimes work in binary, for example, but may far more often work in Hex! Using Hex, we can group 4 bits together, to create a 'Hex code'. It is a lot easier to remember 4 Hex numbers than 16 bits, for example! To see how this system works, we need to start at the beginning!

13.3 Hexadecimal (Hex - base 16)

We have already seen that a digit's worth depends on what position it is in relative to the other digits in the number.

- Base 10 positions are worth 10^7, 10^6, 10^5, 10^4, 10^3, 10^2, 10^1, 10^0
- Base 2 positions are worth 2^7, 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, 2^0
- Base 16 positions are worth 16^7, 16^6, 16^5, 16^4, 16^3, 16^2, 16^1, 16^0

How does the hexadecimal system work? The first thing to note is that there are 16 'numbers' in this system: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. It may well seem a little odd using letters to represent numbers: 10=A, 11=B, 12=C, 13=D, 14=E, 15=F. With a little practice, you will see what an excellent system this is.

Just to remind you, to show what system is being used when you write down a number, it is common to use a subscript. So for example: 34_{10} means (3 x 10) + (4 x 1) whereas 34_{16} means (3 x 16) + (4 x 1) = 58 + 4 = 62_{10}

As you know, when we write down numbers in our daily life, we omit the subscript because we assume that every one is using base 10. Sometimes, especially in computer circles, it is a dangerous assumption to make! If there is any doubt, then add a subscript! When doing exam questions, always use a subscript, just to show how clever you are!

Let's convert a few more Hex numbers into denary:

- $3C_{16}$ is the same as (3 x 16) + (12 x 1) = 60_{10}
- 8_{16} is the same as (8 x 1) = 8_{10}
- $3AF_{16}$ is the same as (3 x 256) + (10 x 16) + (15 x 1) = 943_{10}

13.3.1 A better way to convert into and out of Hex

Going from Hex to denary is relatively easy after you've done a few of them. You have to think a little bit harder going the other way, from denary to Hex. But there is a better way! If you remember, we said that Hex and binary were very closely related. As long as we can do binary to denary conversion off the top of our heads, there is a method for converting denary to Hex, and also back again, very quickly. See if you can follow this example. We are going to convert 125_{10} into a Hex number.

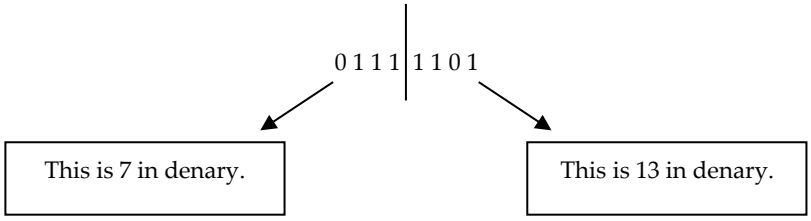
STEP 1 - Convert 125_{10} into binary.

128	64	32	16	8	4	2	1
0	1	1	1	1	1	0	1

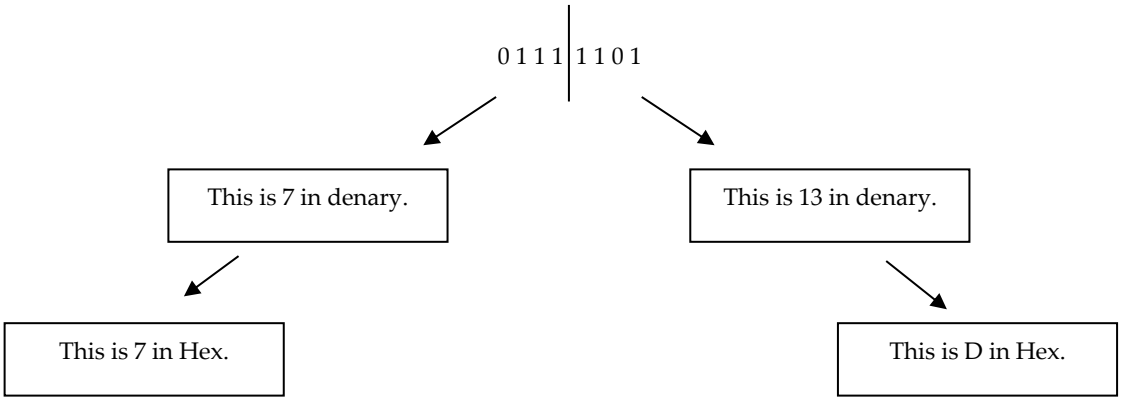
STEP 2 - Split the binary number into two halves.

0 1 1 1 | 1 1 0 1

STEP 3 - Treat each half as a 4 bit binary number. Convert each half back into a decimal again.



STEP 4 - Now convert each of the two decimal numbers into Hex using the Hex number numbering system. Remember, 10=A, 11=B, 12=C, 13=D, 14=E, 15=F.



STEP 5 - Recombine the answer and then proudly display it!

125_{10} is $7D_{16}$

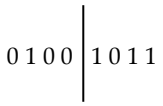
You should always check the Hex answer you got. $7D_{16} = (7 \times 16) + (13 \times 1) = 125_{10}$ so our answer is correct.
(Of course, you could always check your answer using a calculator! In Windows, Go to WINDOWS - ACCESSORIES - CALCULATOR - VIEW - SCIENTIFIC).

Let's do another example. We are going to convert 75_{10} into a Hex number.

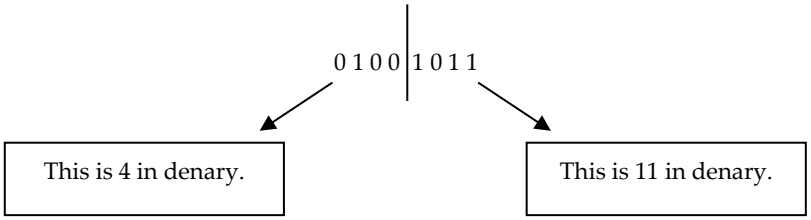
STEP 1 - Convert 75_{10} into binary.

128	64	32	16	8	4	2	1
0	1	0	0	1	0	1	1

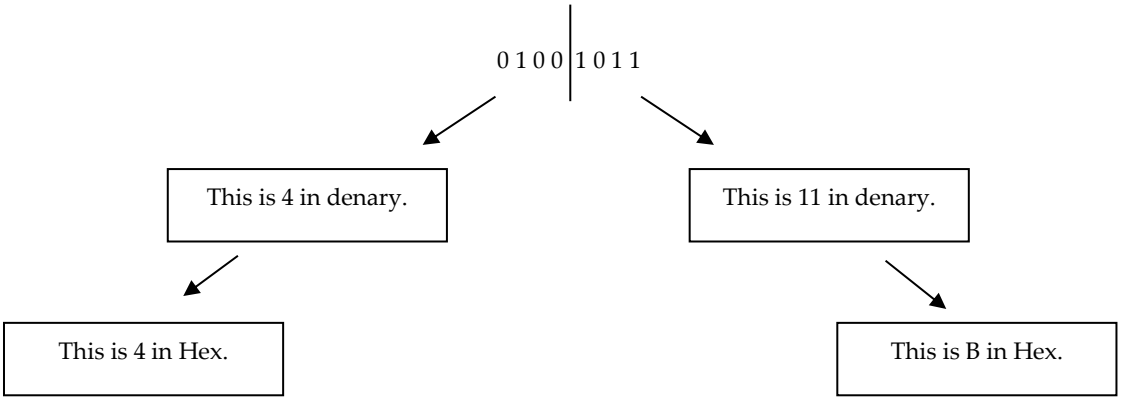
STEP 2 - Split the binary number into two halves.



STEP 3 - Treat each half as a 4 bit binary number. Convert each half back into a decimal again.



STEP 4 - Now convert each of the two decimal numbers into Hex using the Hex number numbering system. Remember, 10=A, 11=B, 12=C, 13=D, 14=E, 15=F.



STEP 5 - Recombine the answer and then proudly display it!

75_{10} is $4B_{16}$

You should always check the Hex answer you got. $4B_{16} = (4 \times 16) + (11 \times 1) = 75_{10}$ so our answer is correct. t may seem a little long-winded, but this method is very mechanical. Once you've done a few, you'll be an expert. Besides, it's good practice for binary conversion! You can use this method to convert from Hex to decimal, too! Start at Step 5 and work backwards to Step 4, then 3, 2 and 1.

13.4 Octal (base 8)

This number system is another important numbering system in computing. Octal, or the base 8 system, uses 8 digits: 0, 1, 2, 3, 4, 5, 6 and 7. We have already seen that a digit's worth depends on what position it is in relative to the other digits.

- Base 10 positions are worth 10^7, 10^6, 10^5, 10^4, 10^3, 10^2, 10^1, 10^0
- Base 2 positions are worth 2^7, 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, 2^0
- Base 16 positions are worth 16^7, 16^6, 16^5, 16^4, 16^3, 16^2, 16^1, 16^0

You probably have guessed what the positions in the base 8 system are worth:

$$8^7, 8^6, 8^5, 8^4, 8^3, 8^2, 8^1, 8^0$$

So, for example:

- 24_8 in decimal is (2 * 8) + (4 x 1) = 20_{10}
- 361_8 in decimal is (3 x 64) + (6 x 8) + (1 x 1) = 241_{10}
- 4125_8 in decimal is (4 x 512) (1 x 64) + (2 x 8) + (5 x 1) = 2133_{10}

As with Hex, it is slightly easier going from Octal to decimal than from decimal to Octal. Again like Hex, we can make use of the fact that Octal is related very closely to binary!

We are going to convert 78_{10} into an Octal number.

STEP 1 - Convert 78_{10} into binary.

128	64	32	16	8	4	2	1
0	1	0	0	1	1	1	0

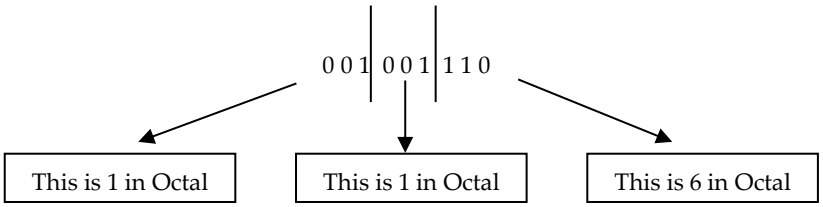
STEP 2 - Add an extra zero bit on the left, so you have 9 bits in total.

0 0 1 0 0 1 1 1 0

STEP 3 - Split the number into thirds.

0 0 1 | 0 0 1 | 1 1 0

STEP 4 - Treat each third as a 3-bit number. Convert each 3-bit number straight into Octal! (Remember, in Octal, you use 0, 1, 2, 3, 4, 5, 6 and 7).



STEP 5 - Recombine the numbers and proudly display the answer!

78_{10} is 116_8

You should always check the Octal answer you got. 116_8 = (1 x 64) + (1 x 8) + (6 x 1) = 78_{10} so our answer is correct.

Let's do another example. We are going to convert 31_{10} into an Octal number.

STEP 1 - Convert 31_{10} into binary.

128	64	32	16	8	4	2	1
0	0	0	1	1	1	1	1

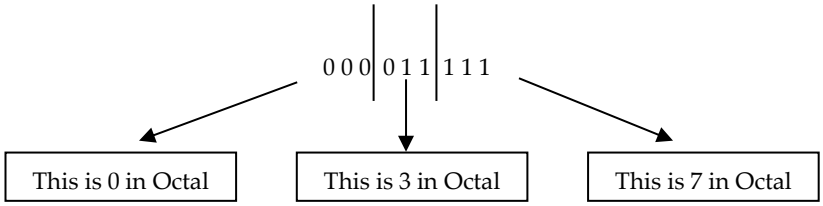
STEP 2 - Add an extra zero bit on the left, so you have 9 bits in total.

0 0 0 0 1 1 1 1 1

STEP 3 - Split the number into thirds.

0 0 0 | 0 1 1 | 1 1 1

STEP 4 - Treat each third as a 3-bit number. Convert each 3-bit number straight into Octal! (Remember, in Octal, you use 0, 1, 2, 3, 4, 5, 6 and 7).



STEP 5 - Recombine the numbers and proudly display the answer!

31_{10} is 037_8 which is 37_8 (as leading zeros in front of numbers are irrelevant).

You should always check the Octal answer you got. $37_8 = (3 \times 8) + (7 \times 1) = 31_{10}$ so our answer is correct.

You can use this method to convert from Octal to decimal, too! All you do is start at Step 5 and go backwards to Step 1! After a bit of practice, you can miss out Step 2! It's only there for 'balance' and can be omitted once you know what you are doing.

13.5 Binary Coded Decimal (BCD)

So far, we have stored numbers as binary, or in a form very close to binary such as Octal and Hex. This is very useful, and allows us to do maths on the codes. In some applications, however, it is more important to maintain a link between the position of a decimal number and the way it is represented in binary. It is useful where you need to store the digits as data type 'character', where you don't do any actual processing on them. An example is a coded sales number.

A mail order company uses codes such as 24361885. This isn't a number! It's a code made up of characters (not 'numbers' in the sense of 'an amount'). The first two characters might be a product code, the third character might be a country code and so on. Now you never 'add' or 'subtract' or do any other maths operations on these codes, but you might need to strip out parts of the code at some point. For example, if you needed to know the country, you would strip out the country character. You could store each character as a byte. If you did this you would need 8 bytes in total (because there are 8 characters). But that is inefficient compared to storing it using BCD. In this system, each character would only need 4 bits, so you would only need a total of 4 bytes to store 8 characters.

Another example of the use of BCD is when you need to store the codes that get displayed on a calculator. Binary Coded Decimal (BCD) is a method for doing just this. You can store each separate digit in a number as a 4 bit code. Each one can then

be sent to each part of the display unit that displays just one digit. The BCD can be quickly decoded and turned into instructions that tell the calculator which digit to display. BCD conversion is very straightforward!!

13.5.1 Work through this example

Convert 345291_{10} into BCD.

STEP 1 - Break up the individual digits in the denary number

3	4	5	2	9	1

STEP 2 - Convert each digit into a 4 bit binary number.

3	4	5	2	9	1
0011	0100	0101	0010	1001	0001

STEP 3 - Proudly display your answer!

0011 0100 0101 0010 1001 0001

It doesn't matter how many denary digits you have - just convert each one into a 4 bit binary code! Going from BCD to denary is equally straightforward. For example, 0111 1001 1000 0000 0100 is the denary number 79804. Isn't conversion really fast? After a few of these, you hardly need to think!

Notice the very close relationship between the position of each denary digit and the position of each 4-bit group.

Q1. Represent the denary numbers 10, 31 and 250 in binary using one byte.
Q2. What is the denary equivalent of these binary numbers: 00000100, 10101010, 10000001
Q3. How many bits are there in three bytes?
Q4. What is the maximum and minimum numbers I can represent using one byte?
Q5. Convert these numbers into their denary form: 36_{16} and 3_{16} and FA_{16} and $2DE_{16}$
Q6. Showing your working step by step, convert these decimal numbers into Hex:

a) 103_{10} b) 14_{10} c) 58_{10} Don't forget to check your answer.

Q7. Showing your working step by step, convert these Hex numbers into decimal:

a) 29_{16} b) 85_{16} c) $A1_{16}$ Don't forget to check your answer.

Q8. Showing your working step by step, convert these decimal numbers into Octal:

a) 50_{10} b) 14_{10} c) 8_{10} Don't forget to check your answer.

Q9. Showing your working step by step, convert these Octal numbers into decimal:

a) 261_8 b) 20_8 c) 2_8 Don't forget to check your answer.

Q10. Convert these denary numbers into BCD: a) 45 b) 71009 c) 10

Chapter 14 - Negative binary numbers

14.1 Negative binary numbers

So far, we have seen how to represent numbers using binary, Hex, Octal and BCD. We have only seen how to represent positive numbers. There will often be times when we want to store negative numbers. For example, we may need to record negative temperatures, or we could be writing a 'bank account' program where people's bank balances are stored, and they're sometimes overdrawn, or negative! There are two systems we need to know about; 'sign and magnitude' and 'twos complement'. They each have advantages and disadvantages.

14.2 Sign and magnitude

Suppose I want to record the weather temperature at 12.30pm on one day in degrees C in one byte. After some research, I've found out that I need to represent numbers from -50 to +50 and I've decided to store only whole numbers. Clearly, in the systems we've looked at so far, we have no way of recording negative numbers. I need to look again at my byte and redefine the bits in it. What I have decided to do is to use bit 7 (the bit on the far left), to represent the 'sign' of the number - whether it is positive or negative. In my system, I have decided that if bit 7 is set, then the number is negative. If it is reset, then this signals to me the number is positive. Bit 7 will only tell me the sign - it won't have a 'magnitude'. The other bits can keep the values they had before! (Incidentally, the bit on the far right is 'bit zero'! In one byte, you would talk about 'bit 0', 'bit 1', 'bit 2' and so on up to 'bit 7'). My new numbering system is now shown below!

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
+/-	64	32	16	8	4	2	1

The biggest number I can represent in my new system is shown below.

+/-	64	32	16	8	4	2	1
0	1	1	1	1	1	1	1

This corresponds to +127. I can't represent anything bigger than this. Notice that bit 7 is reset, because the number is positive. The smallest number I can represent is shown below.

+/-	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1

This corresponds to -127. I can't represent anything smaller than this. Notice that bit 7 is now set, because the number is negative. This numbering system has a name. We talk about a **sign and magnitude** system, because the bits not only represent the size (or '**magnitude**') of a number but one bit is also used to represent the **sign** of the number.

Now the sign and magnitude system is great for my purpose - to record one temperature. Using this system, I could now use two bytes to record the temperatures on 2 successive days if I wanted to. If I now wanted to work out the average of these temperatures, I'd need to begin by adding the two numbers together. For example, suppose I measured the temperature on two successive days and found them to be 23 degC and -16 degC. How could I add +23 to -16?

+/-	64	32	16	8	4	2	1
0	0	0	1	0	1	1	1

PLUS

+/-	64	32	16	8	4	2	1
1	0	0	1	0	0	0	0

There is a small problem! We can't add 'symbols' (the + and the -) together in the same way that we can add 'amounts of something' together. It's a little bit like adding +23 and -16 in denary. You can add only the number parts together to get 39, and then you're not sure how to 'add' the symbols, and you end up with a wrong answer anyway!

So, sign and magnitude is a nice, simple system for recording positive and negative numbers. If we want to do sums, however, we cannot use this system (easily)! We need a different one! Also note that if we have 8 bits, we use the bit on the left for the sign. If we have 5 bits, we also use the bit on the left for the sign. In fact, however many bits we have available, we always use the bit on the left for the sign bit.

14.3 Two's complement

This system might seem strange at first but stick with it and follow the examples! I'm going to do away with the system of using bit 7 to show the sign of the number. I'm going to reallocate the 'worth' of each bit position in the following manner:

-128	64	32	16	8	4	2	1

Notice what bit 7 is worth now. It's worth -128. The biggest number I can represent is when bit 7 is zero, and all the other, positive, bits are one. This is shown below.

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

You should be able to work out that the largest number you can represent is +127. The smallest number is when bit 7 is set, and the other numbers are reset. This is shown below.

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

So, the smallest number you can represent is -128. So how do you represent +12 using this system? You do it in the normal, binary way! This is shown below.

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

And how do you represent -12 using this system? There are a number of tricks you can use to quickly turn a negative number into its two's complement form. My favourite is:

1) Write down the number as a positive binary number.
2) Starting with bit zero, copy all the bits up to and including the first set bit.
3) Invert all the others.

For example, to convert -12 into its two's complement form:

bit 7 -128	bit 6 64	bit 5 32	bit 4 16	bit 3 8	bit 2 4	bit 1 2	bit 0 1	Comment
0	0	0	0	1	1	0	0	Write down the binary for +12.
					1	0	0	Start at bit zero. Copy all the bits up to and including the first set bit. Leave the others as they are until the next stage.
1	**1**	**1**	**1**	**0**	**1**	**0**	**0**	Now invert all the remaining bits. And there's your answer!

Just to double-check, 1111 0100 is the same as (4+16+32+64) - 128 = -12. It works!

Let's do another example, just for fun. Put -80 into its two's complement form.

bit 7 -128	bit 6 64	bit 5 32	bit 4 16	bit 3 8	bit 2 4	bit 1 2	bit 0 1	Comment
0	1	0	1	0	0	0	0	Write down the binary for +80.
			1	0	0	0	0	Start at bit zero. Copy all the bits up to and including the first set bit. Leave the other bits until the next stage.
1	**0**	**1**	**1**	**0**	**0**	**0**	**0**	Now invert all the remaining bits. Here's the answer!

The two's complement of -80 is 1011 0000. Just to double-check, (16 + 32) - 128 = -80 It works!

Let's do another example, just for fun. Put -96 into its two's complement form.

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
-128	64	32	16	8	4	2	1	Comment
0	1	1	0	0	0	0	0	Write down the binary for +96.
		1	0	0	0	0	0	Start at bit zero. Copy all the bits up to and including the first set bit. Leave the others as they are until the next stage.
1	**0**	**1**	**0**	**0**	**0**	**0**	**0**	Now invert all the remaining bits. And there's your answer!

Just to double-check, 1010 0000 is the same as 32 - 128 = -96. It works!

Let's do another example, just for fun. Put -100 into its two's complement form.

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
-128	64	32	16	8	4	2	1	Comment
0	1	1	0	0	1	0	0	Write down the binary for +100.
					1	0	0	Start at bit zero. Copy all the bits up to and including the first set bit. Leave the other bits until the next stage.
1	**0**	**0**	**1**	**1**	**1**	**0**	**0**	Now invert all the remaining bits. Here's the answer!

The two's complement of -100 is 1001 1100. Just to double-check, (16 + 8 + 4) - 128 = -100. It works!

Note that if we have 8 bits, then the bit on the left is negative. If we have 5 bits, the bit on the left is also negative. In fact, however many bits we have available, the bit on the left is negative.

Q1. What is meant by an 'integer'?
Q2. Is 0 an integer? Discuss.
Q3. Describe how a byte can be used to represent a negative number using sign and magnitude.
Q4. What is the biggest number that can be represented using sign and magnitude if you have six bits?
Q5. What is the smallest number that can be represented using sign and magnitude if you have six bits?
Q6. Describe how a nibble can be used to represent a negative number using two's complement.
Q7. What is the biggest number that can be represented using two's complement if you have a nibble?
Q8. What is the smallest number that can be represented using sign and magnitude if you have a nibble?
Q9. Convert -23 and -106 into a byte using sign and magnitude representation.
Q10. Convert -23 and -106 into a byte using two's complement representation.

Chapter 15 - Binary arithmetic

15.1 Binary addition

We've now seen how to represent both positive and negative numbers in binary. Before we look at adding some numbers, let's look the rules of binary addition.

- Zero plus zero = zero! In binary, 0 + 0 = 0 (0 is the binary equivalent of zero)!
- One plus zero (or zero plus one) = one! In binary, 1 + 0 = 1 (1 is the binary equivalent of one)!
- One plus one = two! In binary, 1 + 1 = 10 (10 is the binary equivalent of two).
- One plus one plus one = 11 (11 is the binary equivalent of three).
- One plus one plus one plus one = 100 (100 is the binary equivalent of four).
- And so on.

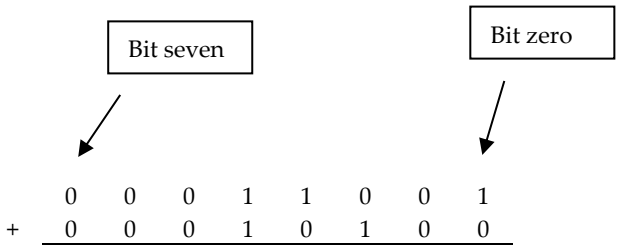
Let's look at some straightforward examples before we look at some more complicated ones later in the chapter.

15.1.1 Example 1

Add 25 to 20 in pure binary, using a byte for each number.

The binary equivalent of 25 is 0001 1001
The binary equivalent of 20 is 0001 0100

So we need to do the sum:



	0	0	0	1	1	0	0	1
+	0	0	0	1	0	1	0	0

Remember the convention. Bit zero is on the right. Bit seven is on the left. Starting on the right hand side, just as if we were adding decimal numbers, we follow the rules in 15.1 above. This gives us:

	0	0	0	1	1	0	0	1
+	0	0	0_1	1	0	1	0	0
	0	0	1	0	1	1	0	1

We can double-check our answer. 0010 1101 is the same as 32 + 8 + 4 + 1 or 45 in decimal. So we are correct! Notice the carry bit in bit 5 resulted from when the two bit 4 digits were added together.

15.1.2 Example 2

Add 15 to 15 in pure binary, using a byte for each number. The binary equivalent of 15 is 0000 1111 so we need to do the sum:

	0	0	0	0	1	1	1	1
+	0	0	0	0	1	1	1	1

Starting on the right hand side, just as if we were adding decimal numbers, we follow the rules in 15.1 above. This gives us:

	0	0	0	0	1	1	1	1
+	0	0	0	0_1	1_1	1_1	1_1	1
	0	0	0	1	1	1	1	0

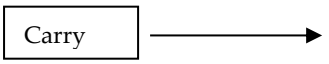
We can double-check our answer. 0001 1110 is the same as 16 + 8 + 4 + 2 or 30 in decimal. So we are correct! Notice how the carry bits were used and shown in our calculations. It is always a good idea to show your working when doing arithmetic.

15.1.3 Example 3

Add 150 to 140 in pure binary, using a byte for each number. The binary equivalent of 150 is 1001 0110 and the binary equivalent of 140 is 1000 1100 so we need to do the sum:

```
    1  0  0  1  0  1  1  0
+   1  0  0  0  1  1  0  0
```

Starting on the right hand side, just as if we were adding decimal numbers, we follow the rules in 15.1. This gives us:




```
    1  0  0  1  0  1  1  0
+   1  0  01 01 11 1  0  0
1   0  0  1  0  0  0  1  0
```

Something strange has happened! We have generated a **carry**. Our answer needs to be in a byte, but we can't hold the answer in a byte. We can if we have 9 bits, but a byte only has 8. If ever you add some **pure binary positive** numbers together and you generate a **carry**, then you must **disregard** the answer. In this example, 1001 0110 + 10001100 gave us the answer 0010 0010 with a **carry**. Because a carry was generated when we did the addition, we know that the answer can't be stored in a byte, is meaningless and should therefore be disregarded.

If this ever happens in a program, an error message would be displayed. The error message is known as an **overflow error**. It means that an answer was generated but it was too big to store in the data structure set up for the answer. Our carry bit in this instance could also be referred to as an **overflow bit**. It's also possible to have an **underflow error**. This is generated when the answer is too small to be represented in the data structure set up for it.

15.2 Binary subtraction

Subtraction is the same as adding a negative number! For example,

- 10 - 5 is the same as 10 + (-5).
- 123 - 100 is the same as 123 + (-100).
- -23 -34 is the same as -23 + (-34).

We can use this property if we need to subtract numbers in binary because we know how to convert a negative number into its two's complement form. If we have to do the sum 23 - 5 (in binary, of course) then we will

- Rearrange it so that it reads 23 + (-5).
- Work out what 23 is as a two's complement number.
- Work out what -5 is as a two's complement number.
- Add 23 to -5.

So, any time we have to do a subtraction, we will rearrange the numbers so that it is an addition! Let's look at some examples. It is worth noting before we begin that all numbers in the next examples, both positive and negative, will be in two's complement form. This is actually quite **important** to note because it has an impact on the way we interpret any carry generated.

15.2.1 Example 1

Let's do the sum 28 - 16 = ? The first thing that we need to do is to rearrange it slightly.

28 - 16 = ? is equivalent to 28 + (-16) = ?

We have rearranged this sum for a very good reason. It involved a negative number. By putting the sum in the form 28 + (-16) we represent easily both 28 and also (-16), using the two's complement numbering system. Once we have got both numbers into the two's complement numbering system, we can then add them both together. Notice that we have converted (-16) into a two's complement number using the three steps we met in the previous chapter. We write down the positive version of the number. Then we copy all the bits from bit zero up to and including the first one bit. Finally, we invert all the remaining bits.

The steps to doing this calculation are shown below.

Carry	bit 7 -128	bit 6 64	bit 5 32	bit 4 16	bit 3 8	bit 2 4	bit 1 2	bit 0 1	Comment
	0	0	0	1	1	1	0	0	Write down the binary for +28.
	0	0	0	1	0	0	0	0	Write down the binary for +16
				1	0	0	0	0	Start at bit zero. Copy all the bits up to and including the first set bit. Leave the others as they are until the next stage.
	1	1	1	1	0	0	0	0	Now invert all the remaining bits. This is the two's complement form of -16!
									Let's now add the two's complement form of +28 to the two's complement form of -16.
	0	0	0	1	1	1	0	0	+28
	1	1	1	1	0	0	0	0	-16
1	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	+28 + (-16) = +12

Just to double-check, 0000 1100 is 8 + 4 = 12 in the denary system.

Note the 'carry' bit here. We are working with single bytes. Our answer is in a single byte. Unfortunately, when we did our sum, the one byte wasn't big enough to hold the answer. We had to use an extra bit, called a 'carry'.

15.2.2 The rules when a carry occurs with two's complement arithmetic.

Sometimes, the carry tells us something. Sometimes we ignore it! How do you decide?

1) When you add a positive number to a negative number, the rule is to "ignore any carry produced and accept the result as true". You saw that in the above example.
2) If you add 2 negative numbers together, you will always get a carry. If bit 7 is a zero, then ignore the result. If bit 7 is a one, then accept the result.
3) If you add two positive numbers together (in two's complement form), you'll never get a carry. To check if the result is valid or not, you need to look at bit 7. Remember, in two's complement, all positive numbers begin with zero. If you add two positive numbers together, the result must be positive. You can check if an answer is out of the range that can be held by the byte by looking at bit 7. If it is a zero, accept it. If it is a one, reject the result.

15.2.3 Example 2

Let's do another sum. Let's do -80 - 40 = ? This is the same as -80 + (-40) = ?

Carry	bit 7 -128	bit 6 64	bit 5 32	bit 4 16	bit 3 8	bit 2 4	bit 1 2	bit 0 1	Comment
	0	1	0	1	0	0	0	0	Write down the binary for +80.
				1	0	0	0	0	Copy all the bits up to and including the first set bit
	1	0	1	1	0	0	0	0	Invert all the remaining bits. This is the two's complement form of -80!
	0	0	1	0	1	0	0	0	Write down the binary for +40.
					1	0	0	0	Copy all the bits up to and including the first set bit
	1	1	0	1	1	0	0	0	Invert all the remaining bits. This is the two's complement form of -40!
									Let's now do (-80) + (-40)
	1	0	1	1	0	0	0	0	-80
	1	1	0	1	1	0	0	0	-40
1	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	(-80) + (-40) = -120

We are adding two negative numbers together. We will always get a carry. Our rule says if you add two negative numbers together then check bit 7. If it's a one (as it is here), the result is good. Just to double-check the binary, we have -128 + 8 = -120, so our answer is correct.

15.2.4 Example 3

Let's do another sum. Let's do -80 - 90 = ? This is the same as -80 + (-90) = ?

Carry	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
	-128	64	32	16	8	4	2	1	Comment
	0	1	0	1	0	0	0	0	Write down the binary for +80.
				1	0	0	0	0	Copy all the bits up to and including the first set bit
	1	0	1	1	0	0	0	0	Invert all the remaining bits. This is the two's complement form of -80!
	0	1	0	1	1	0	1	0	Write down the binary for +90.
							1	0	Copy all the bits up to and including the first set bit
	1	0	1	0	0	1	1	0	Invert all the remaining bits. This is the two's complement form of -90!
									Now do (-80) + (-90)
	1	0	1	1	0	0	0	0	-80
	1	0	1	0	0	1	1	0	-90
1	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	(-80) + (-90) = ?

We are adding two negative numbers together. Our rule says if you add two negative numbers together then check bit 7 - if it's a zero, reject the result. Bit 7 is a zero so the result is meaningless.

15.2.5 Example 4

Let's do 96 + 50 = ?

Carry	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
	-128	64	32	16	8	4	2	1	Comment
	0	1	1	0	0	0	0	0	This is the two's complement form of 96.
	0	0	1	1	0	0	1	0	This is the two's complement form of 50.
0	1	0	0	1	0	0	1	0	96 + 50 = ?

We are adding two positive numbers together, so the result must be positive. We aren't checking the carry bit when we add two positive numbers, we're checking bit 7. In this case, bit 7 is a one, so we reject the result.

Summary

- When doing binary arithmetic, all numbers are put in two's complement form unless you are told otherwise.
- You cannot simply accept a result. You must sometimes look at the carry and sometimes look at bit 7 to make a decision as to the validity of the result.
- If a number cannot be held in the data structure provided for it, then we say there is an 'overflow'.

Q1. What is 1 + 1 + 1 + 1 in binary?
Q2. What is 1 + 1 + 1 + 1 + 1 + 1 in binary?
Q3. What is a 'carry' bit?
Q4. What does an overflow bit indicate?
Q5. What does an underflow bit indicate?
Q6. Look up 'status word' and 'flag'. Describe what these terms mean.
Q7. Rearrange 34 – 20 so it is an addition.
Q8. Calculate 34 – 20 in binary. Show your working.
Q9. Rearrange 60 – 32 so it is an addition.
Q10. Calculate 60 – 32 in binary. Show your working.