Hardware Design:

An Introduction via

CECIL  
Dr. Matt Smith and Dr. David Argles

July 2003

# Contents

## Chapter Page

Chapter 1: Anatomy of a Computer 6

Chapter 2 Bits and Bytes 10

Chapter 3 Using Peripherals 16

Chapter 4 Dealing with Discs 22

Chapter 5 The SIM20 - an imaginary computer 26

Chapter 6 An Introduction to CECIL 29

Chapter 7: Life cycle of a CECIL program 34

Chapter 8: CECIL language Concepts 41

Chapter 9: Designing and Writing a Program 59

Appendices 78

Appendix A Introductory Tutorial for CECIL PC Windows Version 79

Appendix B: Debugging Hints 88

Appendix C: Machine Codes 89

Appendix D: The LEDs 90

Appendix E: Language Reference 91

Appendix F: ASCII character set 104

Appendix G: Register and Flag Virtual Addresses 105

Appendix H: Input and Output on the SIM20 106

# What this Course is About

The aim of this course is to introduce you to the basic principles of computer design on the assumption that you currently know little or nothing about it. It is not intended to cover the more subtle aspects of the subject; this can come later. However, if you already have some technical understanding, you will hopefully find a few hints and pointers to other sources that will deepen your understanding of what goes on “under the bonnet”.

## Why CECIL?

The course is distinctive and based on some key foundational beliefs. Firstly, we believe that it is important that theory should be understood in the context of practical experience. That’s not so easy with something theoretical, like computer design, but we believe it is possible by linking it with the study of a low-level language. Normally, you will program with high-level languages like Pascal or C, where each line of your program initiates a number of machine operations. In this course, you will learn “Cecil”, and see how it produces the machine code instructions which the computer actually uses in order to do things. This means that when you write your programs, you will see exactly what the computer is doing at each step in order to achieve your purpose and see how the practice of programming is affected by the theory of the underlying hardware design.

Learning CECIL also presents an opportunity to learn about a language very different from ones such as LOGO, Pascal or C. Both the power of low level languages (you can write a program to do ANYTHING the computer can do) and the amount of effort required to do the simplest things will become clear, providing understanding of why specialist, high level languages have been developed, and are continuing to be developed, in many fields of computing (for example for writing graphics, artificial intelligence, and database management applications).

One of the problems with learning low-level languages is that they tend to be complicated, and you need to learn a great deal before being able to do anything useful at all. In order to deal with this problem, Cecil has been specially designed to introduce the necessary principles without including the unnecessary complications that are found with other low-level languages. It has been designed around an imaginary computer, called the SIM20, but can be simulated on any modern computer. You will be able therefore to write real programs in Cecil, and to run them on your IBM PC, for example. Your first programs will consist of just a few lines of easily understood code, and yet will produce visible results.

## Seeing the Light

We will be looking at the internal structure of the computer itself, and seeing how it consists of individual lumps of information (bytes, if you want to get technical - but we’ll come to that later). These can be somewhat abstract, so another tactic we will be using will be to connect a light display to the printer port. This means that it is possible to store a number inside a certain part of the computer and actually see the lights demonstrating how the number is stored. You’ll be learning about binary arithmetic, since that’s what the computer uses, but you’ll also be *seeing* binary arithmetic. That makes it much easier to grasp.

## The Course

The course will begin with a focus on general design principles, and then move on to consider the imaginary SIM20 as a simple example of the theory. There will then be a strong emphasis on Cecil as we develop the principles of machine code programming. Matt Smith and I have worked on this course for a number of years, putting together the hardware and Cecil programming sides and ensuring that it all integrates properly with other work, particularly software engineering. You are not allowed to compartmentalise your thinking - it’s all related! Within the book you’ll find a guide to the CECIL assembly programming language. It includes a tutorial for learning the language, and a complete language reference of all commands. A set of appendices provides summaries of useful technical information.

## Is CECIL a “real” assembly language?

CECIL is a simple assembly language, based on an imaginary computer architecture. At the heart of the imaginary computer is an imaginary processor. CECIL is based on imaginary systems to allow a “gentle” introduction to assembly language programming. Although the architecture on which CECIL is based is imaginary, it is a real assembly language, and provides a sound introduction to both assembly language programming, and an understanding of how low level languages (and the programs written in them) relate to the architecture of the hardware on which they run.

# Chapter 1: Anatomy of a Computer

## What is a Computer?

In these days of Artificial Intelligence, it’s easy to start to believe that computers can think for themselves. That’s quite a vexed question, revolving around the definition of the word “think”, but for our purposes, it’s useful to consider that the computer is actually a machine that stores a set of instructions which it obeys mindlessly. If those instructions are defective, it will still obey them regardless of what may follow. There are different designs for computers, but the one design that has been central since they were first produced and is still essentially at the heart of modern PCs is what is termed the “Von Neumann” design. If you want to know a bit more detail about this, you should read up about the Von Neumann design - and its inherent problems! - and also about Alan Turing. He was a mathematician who produced the theory on which modern computer design is based. In effect, he proved that a machine that processed a stream of numbers could be made to solve any problem we wish. From this we have the multi-purpose PC that can do anything from working out our accounts to playing multi-user games over the Internet.



**Figure 1.1: Structure of a computer**

Figure 1.1 shows a simple diagram of a computer. It consists of five parts, the Central Processing Unit (CPU) surrounded by Input Devices, Output Devices, some Memory, and an Arithmetic and Logic Unit (ALU). We’ll look at these one by one.

**CPU:** Central Processing Unit. This is exactly what it says it is; a unit at the centre of the computer which processes instructions, reading them in from memory and doing what they say. In effect, it’s the part that we think of as being *the* computer. There are two main competing philosophies for CPU design at the moment, RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer). The Acorn and Sun workstations follow the RISC approach, whilst the IBM PC takes the CISC approach. The argument really boils down to whether bigger is better (CISC), or whether lean and mean (RISC) wins the day. If you want to know more about this, read it up.

**INPUT:** In order to get some instructions into the computer, it will be necessary to have some form of input. Figure 1.1 shows what is meant to be a keyboard, and that’s still the primary means of communication with computers. Other input devices include floppy discs, mouse, modem, bar codes, etc. We’ll look at discs in more detail later; if you want to know more about the technology behind the other input devices, read it up.

**OUTPUT:** Once the information is processed by the computer, it is necessary to get some answers out. The device used mostly is the monitor (or VDU, Visual Display Unit), but other forms of output include floppy discs (a device that can be used for input or output, notice), printers, modems, etc. We’ll look at printers in more detail later, but again, read it up if you want to know more about these technologies.

**MEMORY:** The computer can’t actually *do* anything with its information unless it can store it somewhere in order to process it. This memory (sometimes called “store”) can come in various forms from high speed “cache”, situated on the CPU chip itself, through slower memory kept on the “motherboard” (the board that the computer is built on), to slow external memory such as hard discs (yes, they can be used as memory, as well as input and as well as output…) There are also different forms of memory chip that can be used on the motherboard, such as RAM (Random Access Memory, a bad name, since all these types can be accessed randomly; a better name would be Read-Write Memory), ROM (Read Only Memory - a good name) and EPROM (Erasable Programmable Read Only Memory), for example. Again, you can read up about these if you want to know more.

**ALU:** The computer has to have somewhere to do its working out, whether this is numerical calculations for a spreadsheet or logical calculations for the layout of a word-processed document. This is done in the Arithmetic and Logic Unit. The basic operations include addition and subtraction, and logical operations so that the computer can decide that, IF this is true AND that is true, THEN it will do something. We will be finding out much more about this later.

That indicates the basic design structure for a simple computer. You won’t actually find a computer laid out like this; CPU chips have for years also contained the ALU, for example. But the function of the ALU is important, so it’s important to have it indicated separately on our diagram.

In terms of the detail of computer design, this moves fast. At the time of writing this, things like “pipelining” and “multiple buses” are important. A good way to keep up to date with this is not by reading books - they take too long to come to press. Instead, read the magazines and search the Internet. These are good sources for readable and informative articles on a rapidly developing subject area.

## Using Memory

Memory is a key concept for what we need to do. It is like a huge stack of electronic pigeonholes, where usually each one is numbered from 0 upwards consecutively. The computer is therefore able to store numbers in a particular pigeonhole, or memory location, and retrieve the value stored at a later time when it needs it. It’s like doing a complicated calculation with a notepad by your side. As you go through, you make notes on the pad, jotting down sub-totals that you will want again in a moment. The computer likewise needs to have information available for use at various times. Figure 1.2 shows how memory might be laid out to store information of the various sorts that are required.



Memory can be used for a number of purposes, but we are going to concentrate on four in particular. These are as follows.

**OPERATING SYSTEM:**

This consists of a set of instructions that tells the computer how to work; what to do when you turn on, how to load your program, where to put it and so on. It is large and usually sits towards one end of the computer’s memory, and ought not to be altered.

**WORKSPACE:**

The computer also needs somewhere that it can use as its “memo pad” or **workspace**. This is primarily for the *Operating System* to use; you may well be able to alter it, but it is not a good idea unless you know what you are doing! This is usually sited towards the bottom of the memory space (i.e. from around 0 upwards).

**PROGRAM SPACE:**

The **program** is a specialised set of instructions, fed in by you, to tell the computer how to do a particular task, such as collect your email, add two numbers together, or whatever. The operating System will store these instructions in an area of memory specially set aside for programs, often sitting on top of the workspace.

**USER DATA:**

Your program will also need to keep track of its own data, such as how many emails you’ve downloaded, or what the answer is when you’ve added two numbers together. The Operating System will again set aside an area of its memory for this, probably next to your program space.

## Passing the Numbers Round

We’ve looked at the various constituent parts of the computer; we’re now going to focus in on the CPU and how it communicates with memory. In particular, how does it manage to get the right number at the right time from the right box? In order to do this, the computer needs to use two “buses”, as in figure 1.3.



**Figure 1.3: Communication between CPU and Memory**

The main method of communication between CPU and memory consists of a *data bus* which connects the CPU to every single pigeonhole (location) in memory. As it stands, this would not be too helpful, since the CPU would have **all** the locations answering at once whenever it asked for a number! It would be like picking up the telephone and finding that all the subscribers in your road were all talking at once on the same line! To prevent this, there is also another connection, called the *address bus*. This is like a giant selector switch, controlled by something called the *program counter*. We have already mentioned that each location in memory is numbered from 0 upwards; this is its unique *address*. When a number is stored in the program counter, this is passed down the address bus, and selects only the location with that address. The memory locations are then like greyhound traps, closed with the greyhound inside before the race. The difference with a greyhound race is that all the greyhounds are released from their traps at once at the start. Here, only one “trap” - or rather location - is released at a time, depending on which address is specified in the program counter. Thus by setting the appropriate number in the program counter, the computer is able to examine any location it likes and to find out what it contains, or if required, can write a new value to the location.

The program counter is always used to examine any particular memory location. However, it is called the “program counter” because it is usually used to “remember” where the computer has got to in processing the current computer program. In effect, it therefore keeps a count of which program line it has reached.

Another vitally important part of the computer’s workings which is included in figure 1.3 is the *accumulator*. This has various names, such as *register R0*, or *A-reg*, depending on the manufacturer of the CPU chip, but it is basically a single space inside the CPU which it can use as a temporary holding space. This is where it does its adding and subtracting, for example, and it can’t do anything useful at all without it. Actually, all modern CPUs have multiple registers, probably at least 16, maybe as many as 64, and maybe even more. Some of them are likely to have specialised uses; but none of that need concern us for the moment. Time for you to go away and read that up on your own if you wish. We must move on to have a look at the *bits* and *bytes* ofour memory to see what that’s all about.

# Chapter 2 Bits and Bytes

In the previous chapter, we looked at how the basic building blocks of a computer are laid out and connected to eachother. When we looked at the computer’s memory, we noted that it is used to store data of various types, and in particular, the operating system (a special type of program), the workspace (system data), your program, and your data. In fact, anything that the computer stores is either a set of instructions (program) or information for those programs to use (data). In this chapter, we’re going to look in more detail at how this is handled in the computer’s memory.

## Storing Numbers

If you’ve written computer programs before, then you will be used to seeing lines of *code* (i.e. the lines of the program) which might seem strange at first, but at least have some sort of meaning to you once you learn the language. For example, a program to print the numbers from 1 to 10 might look like this:

FOR I=1 TO 10:PRINT I:NEXT (BASIC)

or:

for (I=1; I<=11; I++) printf(“%i\n”, I); (C )

Maybe that doesn’t look so readable! But the computer doesn’t even understand something as cryptic as programs written in C! Look back at figure 1.2. We said that the computer’s memory consists of a huge stack of electronic pigeonholes. Each one is identical in structure apart from its address, and is designed to hold a number in electronic form. The computer is basically made up of millions of transistor circuits where each little circuit can be switched on or off. An important circuit for memory is something called a *flip-flop*. This is a simple circuit that remembers what was last done to it. So if it was last switched on, its output remains on. If it was last switched off, its output remains off. This makes it into a little memory circuit, remembering ons and offs.

In computer terms, it’s not so convenient to talk in terms of ons and offs all the time. There are a number of conventions, depending on what sort of person is holding the conversation. Thus an electronic engineer building the circuits might use *on* and *off*, or *5Volts* (on) and *0Volts* (off). A mathematician might talk about *TRUE* (on) and *FALSE* (off). But generally, we will talk in terms of *0* (off) and *1* (on). If we put a number of these memory circuits together, we can now make a memory location that can store a number. So some typical memory might look like figure 2.1.

If we now zoom in on our computer’s memory, we can see that each pigeonhole is made up of a number of these little on-off circuits. To help us to talk about it, each memory location is called a *byte*, and each little memory circuit for each on-off is called a *bit* (short for *bi*nary digi*t*). This enables us to store a number in “binary” - a special number system which only uses 0s and 1s instead of all the digits from 0 to 9 as we’re used to in our decimal system. Since the computer works in binary, and we’re learning about how the computer works, we’d better learn a little about this binary number system. It’s not so hard as it looks at first sight!



Let’s start by looking at how the computer would count up in binary. Zero is easy; that’s 0. One is easy, too; that’s 1. But when we come to two, we can’t store that as 2 - the computer doesn’t have a digit to represent two. It therefore has to go over into the next column. Just like 10 represents 1 ten and no units in our decimal system, 10 represents 1 two and no units in the computer’s binary system. Whist our decimal system has column headings that go up multiplying by ten each time (1, 10, 100, 1000 etc.), the computer’s binary system has column headings that go up multiplying by two each time - 1, 2, 4, 8, 16 and so on. These are marked in figure 2.1. It’s a bit easier to see if we look at a table showing the numbers counting up from 0. This is given in figure 2.2.

**Column Heading 128 64 32 16 8 4 2 1 Value** 0 0  
 1 1  
 1 0 2  
 1 1 3  
 1 0 0 4  
 1 0 1 5  
 ------ etc -----  
1 1 1 1 1 1 0 1 253  
1 1 1 1 1 1 1 0 254  
1 1 1 1 1 1 1 1 255

**Figure 2.2: Counting up in Binary**

It’s a worthwhile exercise to continue this table onwards from 5; it gives a good feel for how the binary system works.

You’ll notice that the table stops after eight columns - after all, it’s got to stop somewhere! But there’s more to it than that. Some decision has to be made as to how many bits are going to be put into each memory location, and this is one area where the computer industry has managed to standardise; there are usually eight bits to a byte, hence the eight column headings. Although modern computers talk about being 16, 32 or even 64 bit machines, they achieve this by putting eight-bit bytes together - hence always being specified as multiples of eight. With eight columns (or 8 bits in the byte), you’ll see that the largest number that can be stored is 255.

While you’re playing with binary numbers, look at the relative importance of the different digits in a binary number. If you make a mistake in working out the value by missing out a digit on the right hand side, it only affects the answer by one. But if your mistake is at the other end, on the left, it makes a huge difference, even though you only forgot one digit. For this reason, the far right-hand digit is called the *least significant bit*, since it affects the answer only a little, whist the far left-hand digit is called the *most significant digit*, since it affects the answer the most. This is actually important when thinking about error correction. When the computer sends information over, say, a network, it must make sure that the right information comes out the other end. Knowing which bit to worry about most is therefore important.

Once you’ve got the idea of how to count up in binary, it’s worth spending a little while trying to add and subtract numbers in binary, and trying to convert from binary to decimal and back again. You’ll soon discover that working with binary might be all very well for a computer, but it’s very tedious for humans. Because of this, programmers and system designers usually use yet another number base when working with computer code - *hexadecimal*, which is base 16. This means that the column headings go up multiplying by 16 (1, 16, 256, and so on…). This works well, since binary converts very easily into base 16 and back again. If we take the eight columns in figure 2.2, they reduce to two columns in hexadecimal, since each group of four columns in binary gives a digit from 0 to 15 - exactly what hexadecimal needs. The only problem now is that we must be able to store a units column digit that can be anything from 0 to 15. But we can’t write 15 in the units column, because that means 1 in the sixteens column and 5 in the units column - a total of twenty-one! We must therefore invent six new digits to cover the range from ten to fifteen. Counting onwards from eight, hexadecimal therefore gives 8, 9, A (=10), B (=11), C (=12), D (=13), E (=14), F (=15), 10 (=16), 11 (=17)… etc. So forty-four is represented as 2C (2 in the sixteens column - i.e. 32, plus C, or 12, in the units column).

With all these number bases around, there is a potential problem - what do I actually mean by “11”? Do I mean eleven, or 11 in binary (=3), or 11 in *hex* (short for hexadecimal; =17)? The answer is to use conventions. These vary, depending on the context, so a mathematician might give them as 1110, 112 and 1116. However, for computing, you are more likely to meet conventions like *&11* (hex, 17), *%11* (binary, 3) and *11* (decimal, 11).

## Storing Letters

This is all very well for storing numbers; so long as the computer turns everything into binary, it can hold numbers in its memory, and add and subtract them and do just about anything we want it to with them. But how can it possibly store letters of the alphabet? If it can’t do this, it can’t run my word-processor!

This is actually quite simple. All it has to do is to use a conversion table, so that any given number uniquely represents a letter of the alphabet. If we are to store these letters, or *characters*, on disc and swap them between computers, it will be useful to have a standardised conversion code; several have been established for many years, the most commonly used being the ASCII code. You’ll find a conversion table in Appendix E; there’s a code for each capital letter, a different code for the lower case letters and a code for full stops and colons, for example. Thus code 65 represents A, 66 represents B and so on.

## Bigger Numbers

We’ve said that numbers held in computer memory are held in 8 bit bytes, and that this gives a largest number of 255. This is all very well for some jobs, and alright for giving 256 (don’t forget to include zero) different ASCII codes for storing letters. But we are often going to need bigger numbers. So how can we do this? The answer is to put 2 - or more - bytes together. See figure 2.3.



The second (left-hand) byte is used to extend the first (right-hand) one. When the computer comes to use it, the column heading for the second byte are treated as if they started at 256 instead of 1, and they therefore finish at 32768. Try multiplying up the column headings by two each time, and you should be able to confirm that the last heading on the left becomes 32768. You can now work out that the largest number that can be held in this way is 65535.

There is a still a problem here, though. Look back at figure 2.1 and you’ll see that the memory locations are just held in a great stack. one after the other. We need another convention; the bytes are usually stored in *lo-byte/hi-byte* format. That means that the lower byte in memory holds the left-hand (least significant) of the two bytes, whilst the right-hand (most significant) is held in the higher byte of memory. We’ll see more of this when we come to the work in CECIL.

This is all very well for our work in CECIL, but 65535 is still not an amazingly big number - we wouldn’t think much of a calculator if that was the largest number it could handle. So how do computers handle even bigger numbers? The answer is simply to use even more bytes. Many computer systems use four bytes to represent large numbers, organised in the same sort of way as the two bytes in the example above. You can work out for yourself what the largest number is that it can handle; it must be something like 3 600 000 000!

## Going Negative

We can now handle little numbers, bigger numbers and letters of the alphabet. However, we still have no method for dealing with a calculation like 2-3. There are two ways used for representing negative numbers.

**Use of MSB:** This method sounds simple; simply set aside the most significant bit of the number and use it as a *flag*. If the bit is set to 1, it marks it as a negative number. Otherwise, if the bit is 0, it marks it as a positive number.  
  
But there’s a better way.

**2’s Complement:** One way of understanding this method is to write out the numbersstarting at, say, 5 and going on down until you’ve gone below zero (you will need to assume that you can put an extra one in on the left-hand side as you go below zero).

This might seem odd, but it’s done because the computer can then do a calculation like -1 + 1 and it will get the answer zero (try it). It doesn’t work if you use the previous method; the answer will be -2!

There’s three steps involved in working out a 2’s complement. Suppose you want -5:

1. Start with +5: 00000101 (assuming an 8 bit number)
2. Change over 1s and 0s: 11111010
3. Add 1: 11111011 (2’s complement of 5, or -5)

## Floating Point

There’s one more thing that we need to be able to do with numbers, and that’s to deal with decimals - after all, not everything is a whole number. To see how this is dealt with, we first of all need to consider some very big numbers indeed. If we wanted to deal with a number that was billions of billions, even our very large integers wouldn’t handle it, and in any case, all those zeros on the right-hand side tend to get confusing. Many calculators, and all computers, get round this by using *scientific notation*. The number is split into the digits bit that we’re really interested in, and a count of the number of times that we’ve had to move the decimal point. For example:

36 341 000 000 000 becomes 3.6341 x 1013

This is known as *standard notation*; the number is always represented as x.xxx, with the decimal point coming after the first digit. In order to get the decimal point back to the correct position, the number must then be multiplied by 10 thirteen times (1013 is the mathematical notation that means just this).

Holding a number in this format in the computer’s memory is now easy. All we need to do is to store the two parts, the basic number (3.6341), or *mantissa*, and the number of times it must be multiplied by 10 (13 in our example), or *exponent*.

This method now allows is to deal with decimals as well. Instead of multiplying by ten, we have to divide by ten. For example:

0.00253 becomes 2.53 x 10-3

It’s easy to see that the mantissa is 2.53; that’s just the digits bit of the number that matters. However, in this case, we have to divide by ten three times rather than multiplying. In mathematics, rather than writing this as dividing by 103, it is usual to write it as *multiplying* by 10-3. The minus simply means that the decimal point is moved in the other direction. We already know how to store minus numbers, we can use 2s complement. So if a number is stored in floating point format with a negative exponent, that means that the number is between 0 and 1. In this case, the mantissa would be 2.53, and the exponent would be -3.

We’ll close this chapter with some exercises to give you some confidence with all these numbers.

## Exercises

1. Convert these decimal numbers to (i) Octal (ii) Binary (iii) Hexadecimal:

a) 22 b) 26 c) 31 d) 38 e) 44   
f) 53 g) 58 h) 93 i) 751 j) 1453

2. Convert these octal numbers to (i) Decimal (ii) Hexadecimal (iii) Binary:

a) 73 b) 347 c) 1521 d) 2345 e) 3733

3. Convert these hexadecimal numbers to decimal:

a) 6A b) EF c) 2C8 d) 347 e) 6DA

4. Express this binary number in octal and hexadecimal:

11011101010110101011

5. Magic Square:

Using the nine hexadecimal numbers from A to 12, fill in the following magic square so that each row, column and diagonal adds up to the same amount.



Hints: o Each row, column and diagonal should add up to 2A   
 o Try and work out which number should go in the centre square

6. Subsets:

A set of things - e.g. the days of the week, {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} - can have a number of subsets - e.g. the weekend, {Saturday, Sunday}. Work out a formula for the number of subsets that a particular set can have.

Hint: o Think in terms of binary numbers.

7. Find the two's complement of the following binary numbers:

a) 00110101 b) 01000000

8. Add the binary number 01010000 to each of the two's complement numbers that you worked out in (7) above. Check your results by converting all the numbers from binary to decimal and working it out again in decimal.

9. Perform the following subtractions using two's complement arithmetic:

a) 00110110 - 00011101 b) 00011111 - 11101010   
c) 01001001 - 01101000 d) 00001110 - 00001111

10. Work out the following additions for three bit binary numbers in two's complement. In each case, determine whether or not overflow has occurred:

000 000 111 100 100   
+001 +111 +110 +111 +100  
 \_\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_   
 \_\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_

11. Convert the following numbers into floating point format:

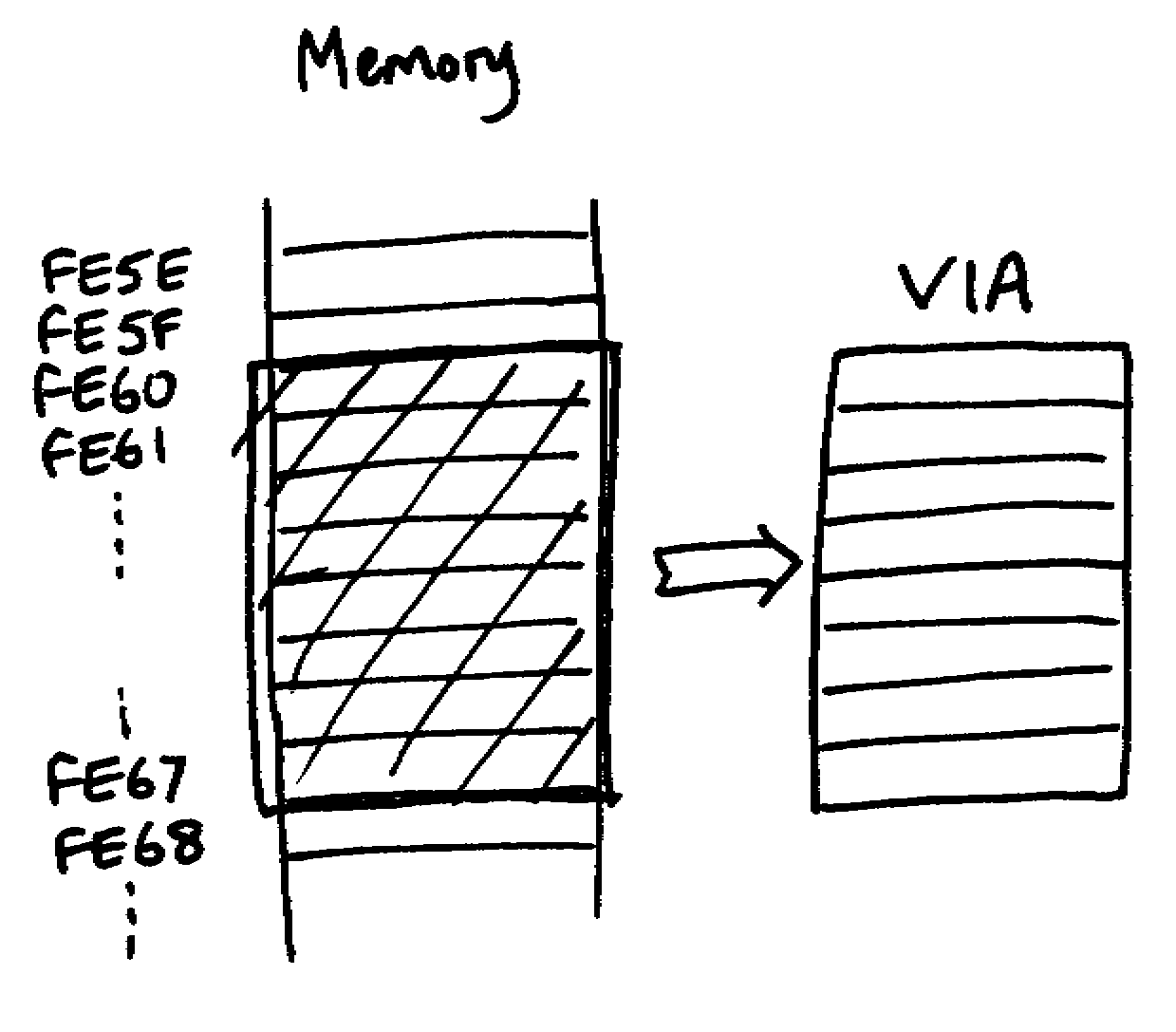
a) 26 b) 237 c) 0.5 d) 0.25 e) 0.375

# Chapter 3 Using Peripherals

So far, we’ve considered what the basic structure of a computer is like, and we’ve looked at the structure of computer memory in some detail, including how the computer communicates with memory. In this chapter, we now need to move on to consider how the computer communicates with “peripheral devices” (or *peripherals*) such as printers. We had a simple diagram in figure 1.1 which showed the CPU sending information to its output devices. We need to look much more carefully at how this information is actually transferred.

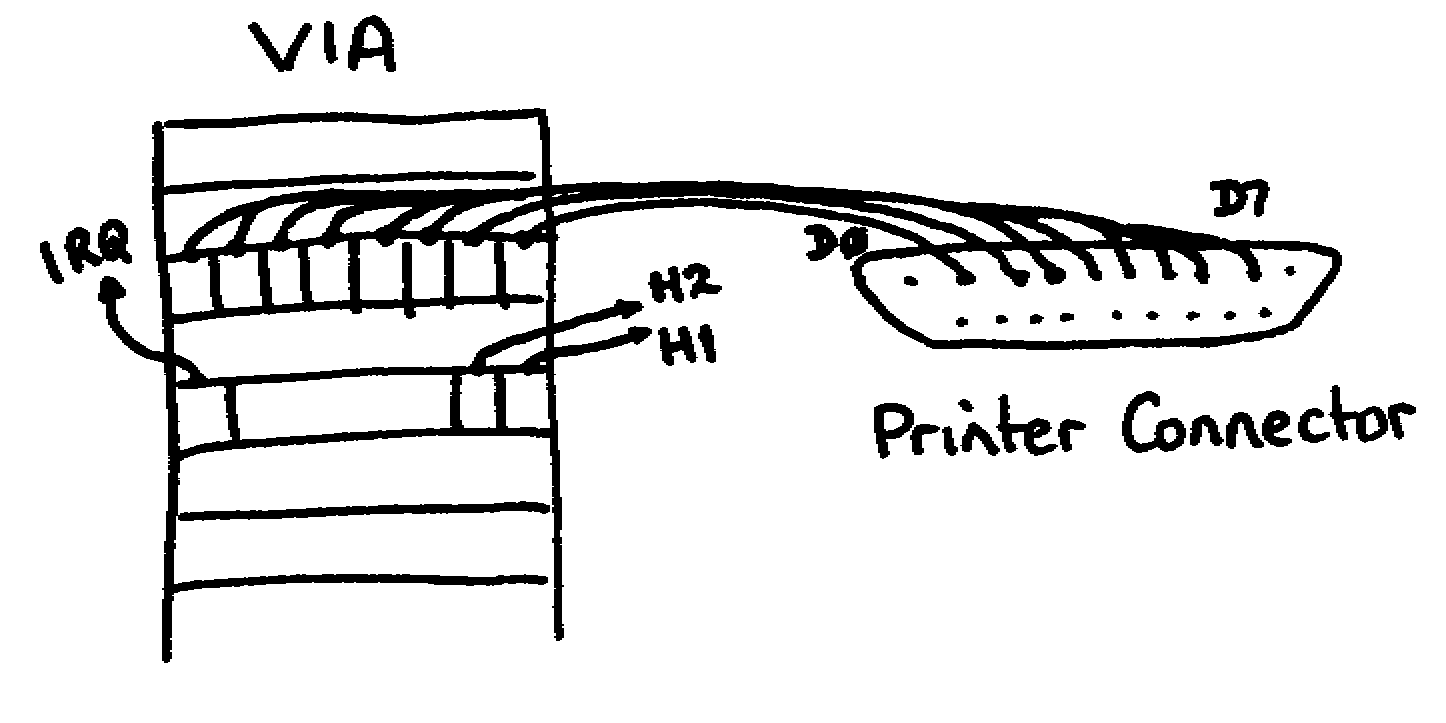
## Interface Adapter Chips

In figures 1.2 and 1.3, our computer’s memory was presented as if it consists only of locations where numbers can be stored and retrieved. In a sense this is true, but memory is also used to enable our computer to talk to devices like printers. Figure 3.1 shows a block of memory which operates just like all the rest as far as the CPU is concerned.



**Figure 3.1: The Versatile Interface Adapter**

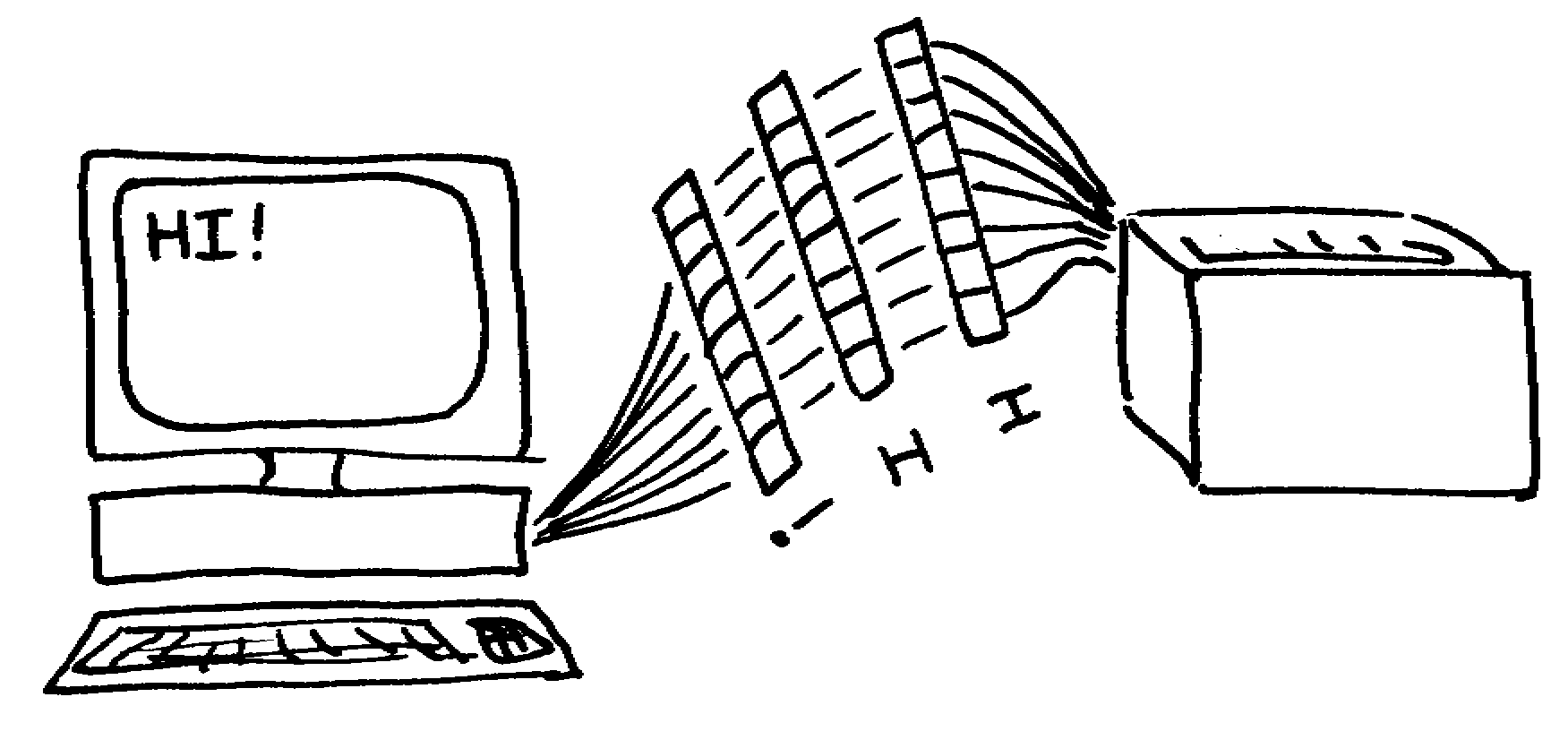
Notice that the individual memory locations are numbered consecutively, as you would expect; the numbers look a bit funny since they are written in *hexadecimal*. That’s just to remind of what we were looking at in the last chapter; it’s not otherwise important. What is important is that the block of memory from FE60 to FE67 (8 locations in this case) have been replaced by a different sort of chip called a *Versatile Interface Adapter* (VIA). This allows the computer to see this chip just as if it were any other part of its memory, and it can store numbers in each of these eight locations and read from them in just the same way. However, this VIA is connected directly to the computer’s printer port, as in figure 3.2.



**Figure 3.2: Connections from a VIA to a Printer Port**

Interface chips like our VIA can actually consist of a number of memory locations or registers; in this case, we’ve taken it to contain eight. In the diagram above, the third location is shown connecting straight to the printer port; the wires literally go directly from the chip itself to the connector on the back of the computer that you plug your printer cable into. You can see three other parts shown on the VIA, a wire labelled *IRQ*, which stands for *Interrupt,* and two wires labelled H1 and H2 - more of this later. For now, we can just notice that the other locations in the VIA are used for different things, and that some of the individual bits of some locations are used separately rather than as a whole byte, as we’ve thought so far.

Now, let’s assume that we wish to pass the message “HI!” from the computer to the printer and print it out. Figure 3.3 shows how it works.



**Figure 3.3: Transferring data from the computer to a printer**

We already know how to store the message “HI!” in the computer’s memory; we can do this in ASCII code, as we saw last chapter. If we can store the message in memory, then we can also transfer it in the same way as ASCII codes across the wires to the printer. However, information moves very slowly once it’s outside the computer. It can be transferred in one of two ways, in *serial*, or in *parallel*. If the transfer is in parallel, then eight (or more) wires connect the computer to the printer, and the information goes over a byte at a time. The alternative is to use one wire only. In this case, the transfer is *very* slow; typically, just one byte would be transferred every thousandth of a second. You might think this sounds fast, but in that time, a modern computer might have processed, say, twenty *thousand* program instructions! Parallel transfer is much faster (still slow by CPU standards), but has the drawback that eight wires are needed instead of one. A typical use for serial transfer includes the modem, for example. So when you dial up your Internet Service Provider on your modem, for example, the computer is, in effect, transferring all your web pages in serial over your modem.

On a typical parallel line, the transfer rate is more likely to be of the order of one whole byte every microsecond (millionth of a second). That’s fast, but not fast enough for the CPU. So in order to avoid having the computer freeze for ages while printing something out, *buffers* are used. The process is as follows. The computer identifies the message to be printed out - “HI!”. It then transfers the data to a specially set-aside part of its memory, called a buffer. It’s just normal computer memory, but the *operating system* (remember that from chapter1?) will have set this bit aside as a *printer buffer*. In fact, it is likely to reside in the *workspace* that we saw in figure 1.2.

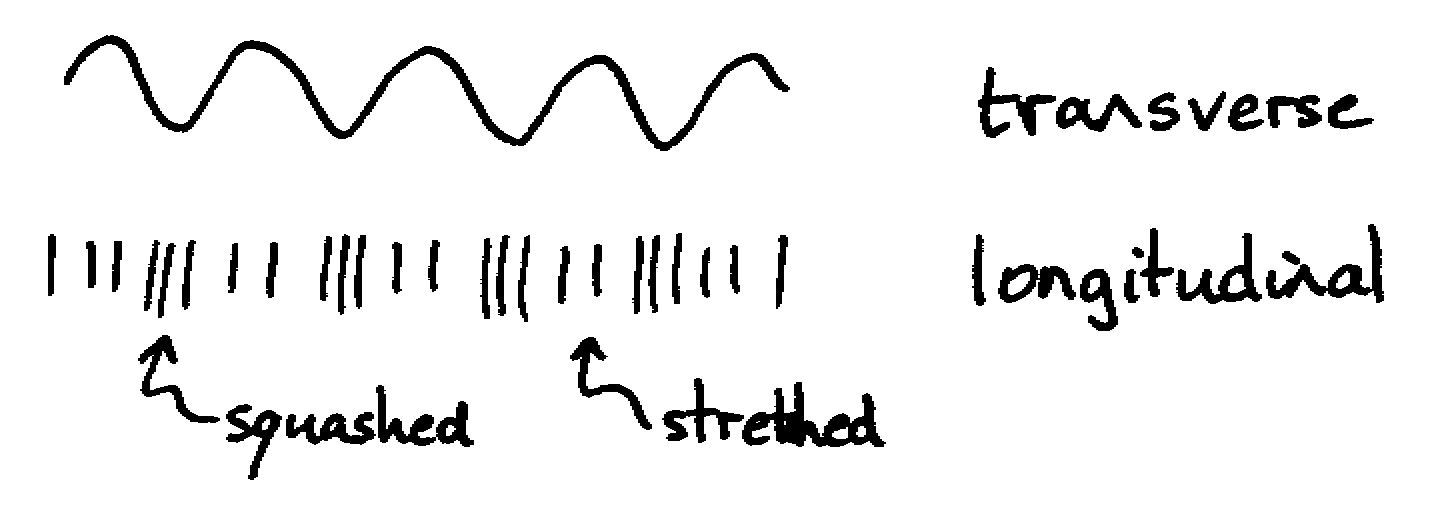
Having copied the message to its buffer, the computer now puts the ASCII code for the first letter into the output register of the VIA, forgets about it, and goes away and gets on with something else - perhaps calculating something difficult on your spreadsheet. When the printer has dealt with the “H”, it lets the computer know by signalling on the IRQ interrupt line. As soon as the computer gets the IRQ signal, it stops what it was doing and gets the next character out of its buffer - the “I” - and places this in the VIA’s output register. This process is repeated until the entire message has been passed over to the printer and printed out. In this way, the computer can handle something really slow, like printing out “HI!”, and still get on and do useful things without having to spend all its time waiting around for the printer to print.

There could still be a problem, since printers themselves are very slow devices. If you reckon that a half decent laser might print at say 8 pages a minute, each page consisting of 600 words of 8 letters (fairly average), then it’s still only printing 640 characters a second (you work it out). That’s not much faster than our snail’s pace serial line! In order to try and avoid having characters trickling over communications lines, the printer itself will also have a buffer. This means that it can accept a great block of printing all in one go, put the information in its own incoming buffer, and then print it out in its own time.

If you have the LEDs attached to your computer for the CECIL work that will follow (see the appendix for details of the LEDs), then you can actually watch how the computer transfers data to the printer if you’re lucky. Get up a decent amount of text and tell the computer to print it out. If you watch carefully, you’ll see the computer telling you that it’s printing. Probably *after* the computer tells you it has finished, the LEDs will start to flash very rapidly as the computer transfers the data from its own buffer to the printer’s buffer. The LEDs will then stop flashing as the transfer ceases, and *after* this, the printer will print out your text, as it processes the text in its own buffer. Printer buffers usually have to be large; it is not uncommon to have about 8Mb (8 megabytes, or 8 million bytes) to enable the printer handle just a few pages.

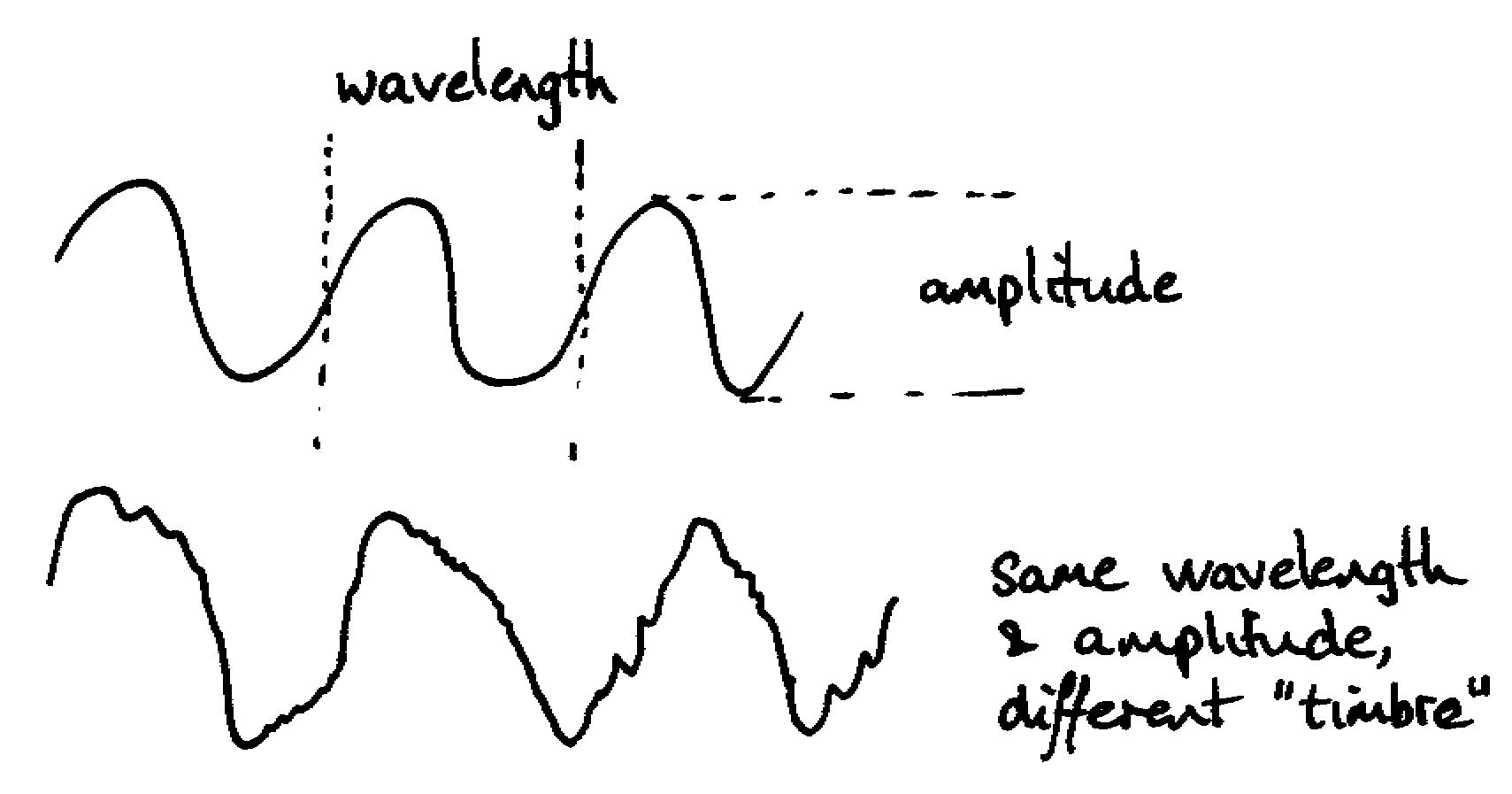
We are now able to pass data from the computer to the printer, and to do it in a way that allows the computer to use its time effectively. But there is still a problem. To understand why, we will need to have a look at some wave theory. Wave theory is usually thought of as being difficult, involving lots of maths - but that’s not true. You already know about waves if you’ve ever watched ripples on a pond, looked at brake lights going on whilst travelling down the motorway, or watched strings vibrating on a musical instrument.

A wave on a pond is basically just a displacement of the water up and down as the ripple passes by. If you drop a stone into the pond, the ripples go out in circles from the point at which the stone hit the water. What is slightly curious is that the water itself is just moving up and down - you can see this if you watch the ripples pass something floating like a cork or a boat - yet the ripples travel outwards. It’s even more fun to watch brake lights going on when the motorway is crowded. Usually it starts when someone pulls out to change lanes; the car behind then has to brake slightly to avoid running into the back. The car behind that one then also has to brake, and so on right back down the lane. The effect is for a ripple of brake lights to travel along the lane - you should be able to see it coming towards you, especially at night. The remarkable thing is that the ripple of brake lights travels *backwards*, even though the cars themselves are travelling forwards!



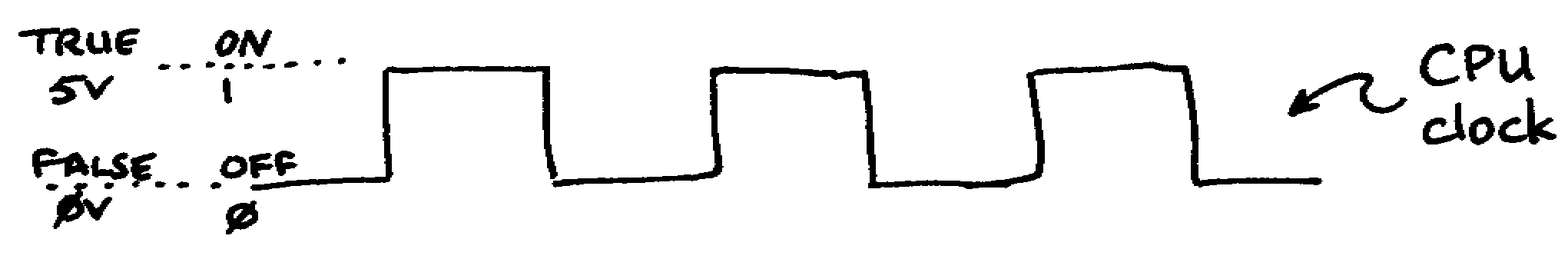
**Figure 3.4: Transverse and Longitudinal Waves**

We need to think about how to represent waves. There are two types of wave, *transverse* and *longitudinal*. Transverse waves are easy to picture; its like looking at the ripples on a pond from the side, so they go smoothly up and down. Longitudinal waves are impossible to draw, but I’ve tried in figure 3.4. Sound waves are an example of longitudinal waves. When we talk, we vibrate the air by pushing it backwards and forwards. This makes “squash” waves, where the air pushes into the air in front, and this pushes the air in front of that, and so on until it reaches our ears. Fortunately, the waves that we’re interested in computers are transverse, so they’re nice and easy to picture; and although sound waves are really longitudinal, we usually draw them as if they were transverse, because it’s easier to work with.



**Figure 3.5: Characteristics of a Wave**

Figure 3.5 shows an “anatomy” of a wave. Any wave “wobbles” at a given speed. For our transverse wave this means that the ups and downs will be close together if it is a fast “wobble”, and further apart if it is a slow one. If we were doing music, we would talk about this as *pitch*, where it might be in the pitch of middle C, or in G, for example. However, this is science, so we’ll call it *wavelength*. The wavelength is literally that, how far (long) the wave travels in one wobble up and down. A related term is *frequency* which is how many wiggles up and down you get in one second. We’re more likely to talk about frequency, but it would be rather difficult to draw the million ups and downs that you get in one second, even for a wave that’s slow in computer terms! Then there’s the shape of the wave. This is a smoothly varying wave, and supposed to be a pure *sine* wave. It sounds horrible if you play it through a loudspeaker, rather like the tuning note that you get (used to get?) on televisions outside of broadcasting time. Something like a piano would produce a much more interesting shape - and sound! - but it would still have the same basic rate of “wobble” - the same frequency. In music, the shape gives rise to what is called tone, or timbre; in science, we’d have to talk about *harmonics*, which simply means that it’s not pure and contains other frequencies in it. The last bit that’s important to us is the size of the “wobble”, or the height in our diagram. That’s called amplitude; in music, this would alter the volume. The greater the amplitude, the greater the volume.



**Figure 3.6: A Computer Waveform**

We can now look at what happens when a computer moves data around on its data bus. Figure 3.6 shows a computer wave. We’ve already said that computers consist of lots of little electronic circuits (flip-flops) that switch between two voltages; this is shown as 0 Volts and 5 Volts in figure 3.6. In order to remind us of what we covered in the last chapter - and what we will need when we get on to CECIL - the values of TRUE and FALSE, on and off, and 0 and 1 are also included alongside 5 and 0 Volts. In order to organise the to-ing and fro-ing of data, the computer has a *clock* that produces regular time signals, so it can keep all its transfers in step. These time signals constantly switch between 0 Volts and 5 Volts, and they switch very fast. This is actually what the manufacturers are talking about when they say a computer is a 60MHz or a 200MHz machine. It simply means that the computer’s clock is changing at a rate (or frequency) of 10 or 200 million times a second (MegaHertz, or MHz). That’s why transfer rates of a few thousand a second is so slow in computer terms, incidentally.

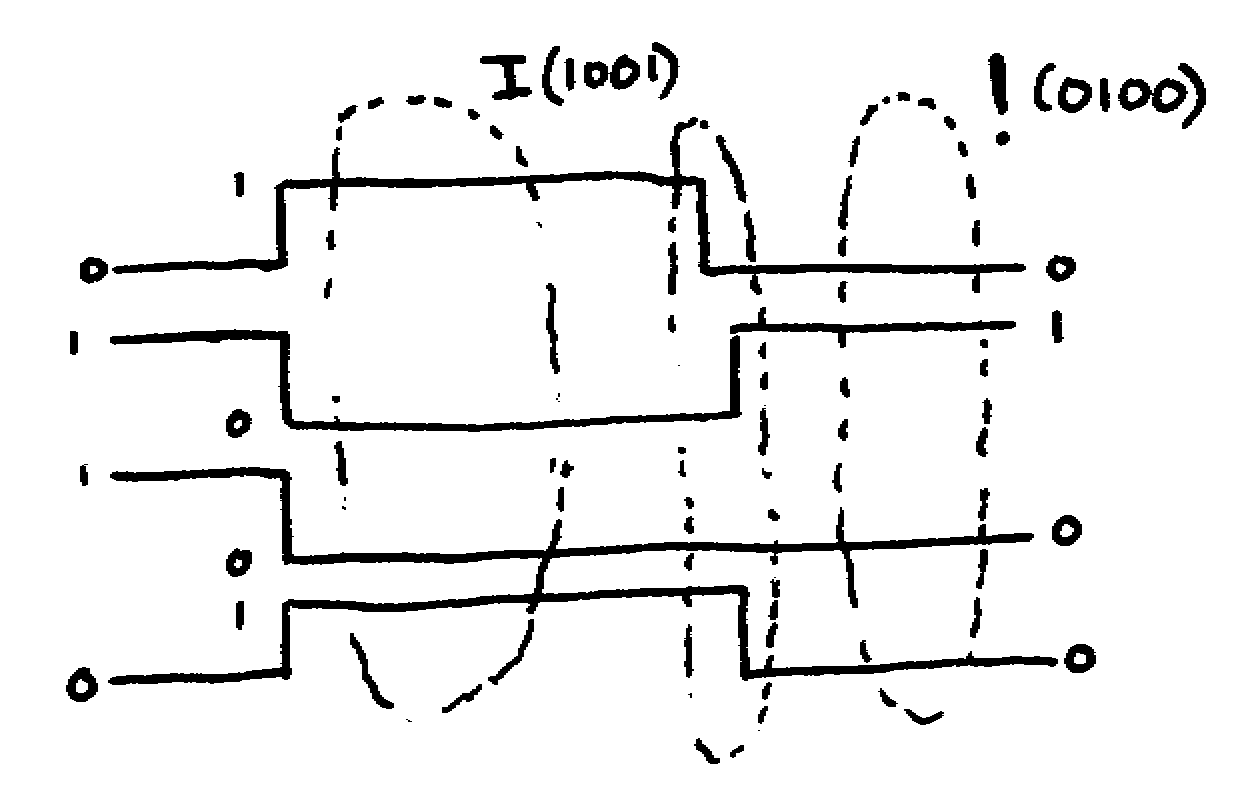
Now, the waveform in figure 3.6 is very square in shape, with rapid changes from 0 Volts to 5 Volts and back again; in fact, it’s called a *square wave*. But this is only approximately what it look like; in reality, it looks like figure 3.7.



**Figure 3.7: Actual Shape of a Computer’s Square Wave (Magnified)**

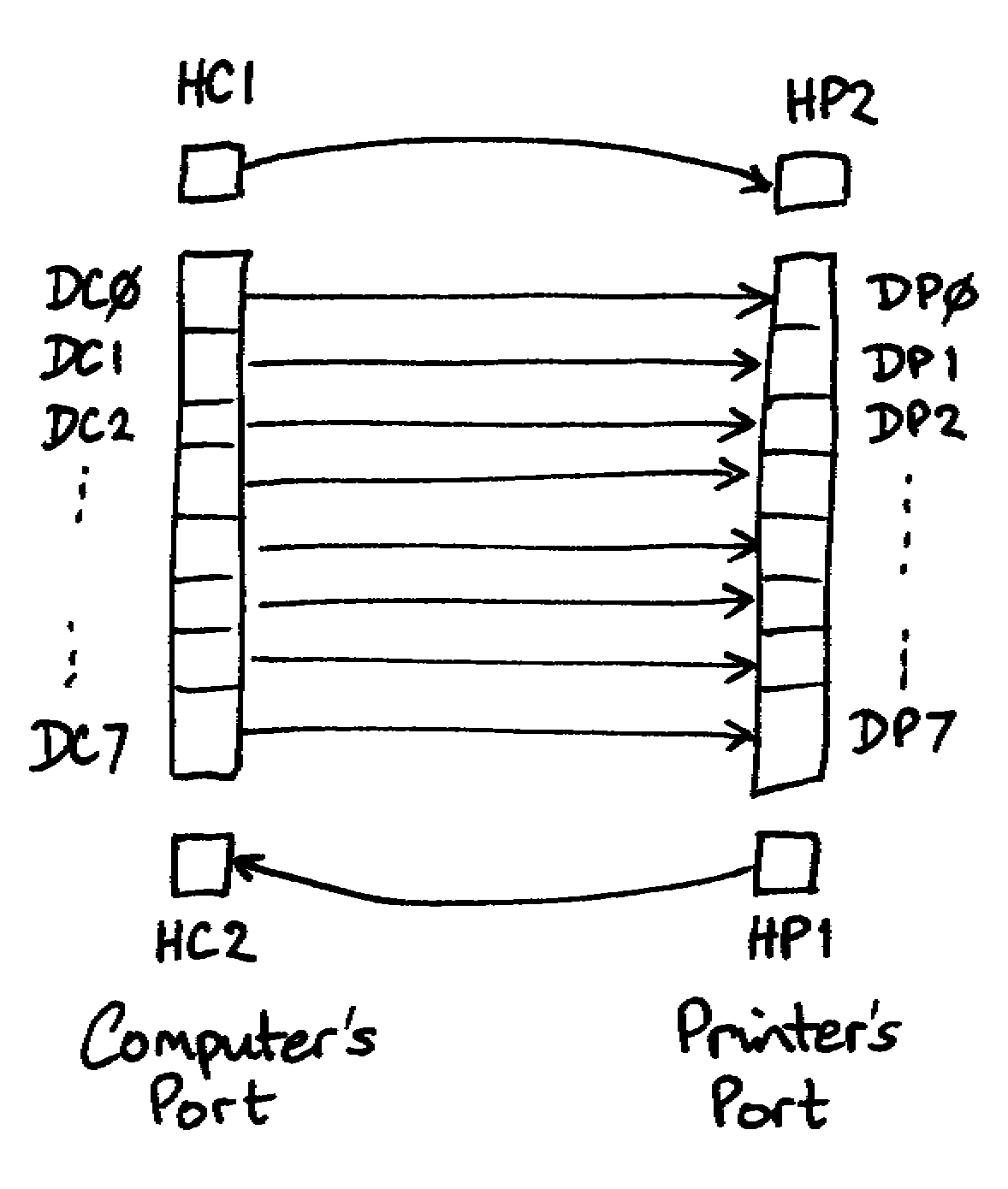
If we were to look at the shape of a computer’s wave more closely, we would see that it really doesn’t go straight up and down with nice sharp angles at all, it actually takes time to change from 0 Volts up to 5 Volts, for example, and when it gets there, it tends to “bounce” a bit. That’s not too surprising; if you think of the suspension on your car, you have the same problem. If the shock absorbers are broken, your car will bounce around like mad every time you hit a bump, like the bounce on our computer wave above. If the shock absorbers are too firm, however, it takes so long for the suspension to move that the car move with the wheel - a rough ride! On our diagram, the equivalent effect would be that the computer waveform would never quite get to the top before it was time to come down again. You can’t have it both ways; either you have lots of bounce, or it takes so long for the wave to get to the top that it never really gets there at all!

Now, this is crucially important when we come to consider how the computer transfers information. Let’s look at the last two characters of our example, just the “I” and the “!”. If we take just the bottom four wires out of the eight in total that join the computer and the printer, then they will carry waveforms like those in figure 3.8.



**Figure 3.8: Character Transfer**

We’re looking at four wires, and therefore four waveforms, each switching between 0 Volts and 5 Volts. In the middle of the diagram, you can see the waveforms set up for the letter I, carrying the code 1001 in binary. Towards the right, you can see the waveforms have switched to give the code for !, 0100 in binary. If you look closely at the point between them where the waves change, you’ll see that they don’t quite change at the same time; the top line goes to 0 before the others change. This could be because they really do change at different times (slightly), or because the computer *thinks* they do because of the slope we saw in figure 3.7, for example. Whatever the reason, the effect is that there is another code entirely that exists, even if only briefly, which is 0001. Let’s say this happens to be the code for “\*”. So, whilst the computer thinks that it is transferring the message “HI!”, the printer thinks it is getting the message “HI\*!”, and the user gets upset because it’s ruined his document he’s trying to print.



**Figure 3.9: Handshaking**

In order to get round this, a system called *handshaking* is used, using two handshake lines as well as the data lines for the information itself. You can see this in figure 3.9. The first step in the transfer consists of the computer putting the data, say for “I”, into the output register of the VIA. It then **waits a bit**. This is vital; it gives the electronics a little time for all the wobble that we noticed in figure 3.7 to die down. When everything is settled, it then changes the value of the first of the computer’s handshake lines, HC1 in figure 3.9, but H1 in figure 3.2. This is the signal to the printer that the data on the data lines (DC0 to DC7) has settled down and can now be read. The printer now reads this data **and gives it time to settle down in its own input register**, before signalling back to the computer that it has read it. It uses its own HP1 to send this signal back, which connects to the computer’s HC2 line. It is this HC2 line which fires off the IRQ signal that we mentioned right at the outset. The computer now knows that the printer has got the “I” safely, and it can put the data up for “!”, repeating the process all over again.

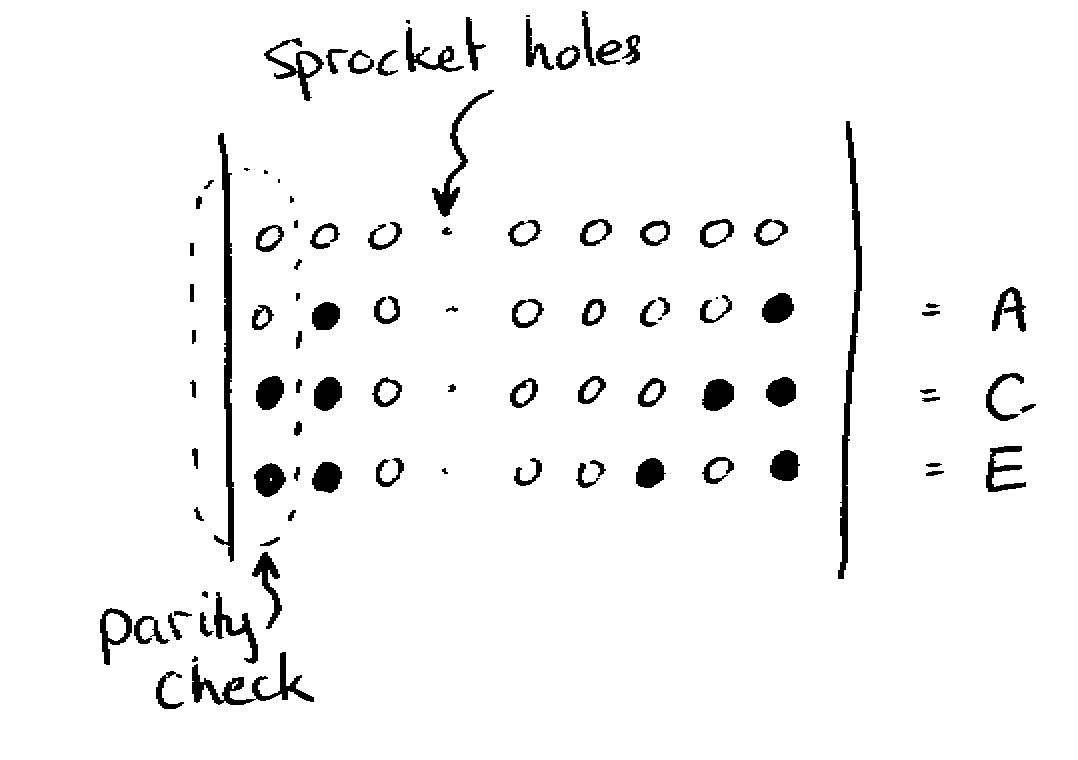
With this in mind, we will now be able to look at how CECIL handles input and output when we get there.

# Chapter 4 Dealing with Discs

We’ve now covered the min principles of how a computer works. There’s still one area that needs to be considered in detail, though; discs. We’ll start this chapter by considering the broader picture with regard to *mass storage* (i.e. discs and tape streamers, etc.), and quickly move on to consider the detail of how discs work.

## A Bit of History

In the early days (and that’s not so very long ago!), the main form of input for day to day work on many computers was paper tape. This is a good place to start, since it is easy to see how data was stored and read back in. Figure 4.1 shows a section of paper tape containing the word “ACE”. You can see there were eight holes across the width of the tape (an earlier standard had just five), and that these were split into columns of five and three by a smaller set of sprocket holes. This meant that it was easy to see which way round the tape should go in the tape reader. Only seven of the hole positions were used for the data itself, so you should be able to work out, from our work in chapter two, that the maximum number that could be held on tape like this was 127.



**Figure 4.1: Paper Tape**

Early systems used mechanical readers with little pins that pressed against the tape. If a hole was punched out of the tape in a certain position, the pin pushed through and the tape reader registered a “1”. If there was no hole, then the pin was kept in its position, and the reader registered a “0”. A cog fitted into the sprocket holes and pulled the tape through the reader at a known speed. A mechanical system such as this was very slow, so later systems used rubber rollers to grip the tape, and the holes were sensed electronically using lights and light sensors. These were much faster, but there was now an increased risk of tearing the paper tape.

With a system in place to store small integers on the paper tape, it was possible to record any information you wished by using the various codes that we considered in chapter two. If you look back at figure 4.1, you should be able to work out that the number recorded on the tape alongside the letter “A” is 65 (64 +1). If you check this against the table of ASCII values, you’ll find that 65 is the ASCII code for “A”. Now, there is a problem that we haven’t addressed so far which is quite important when we’re thinking about mass storage. What happens, for example, if the hole hasn’t been punched out properly? We need to be able to detect the problem. The method that was used to do this was to use the eight column as a *parity check*. There are two ways of doing this, using either an even or an odd number of holes. Let’s describe *even parity*; in this case, the number of holes across the tape must always be an even number. Thus the letter “A” consists of two holes, with just the 64 bit and the 1 bit punched out. It already has an even number of holes (2), so the parity isn’t used. But the next letter is “C”, consisting of three holes, the 64, the 2 and the 1. This is an odd number of holes, so to make sure that the total number of holes across the tape is even, the parity is punched out to make the total number of holes even. Similarly with the letter “E”, the parity has to be used again to make sure that the total number of holes across the tape is even. Now, when the reader reads the tape, it counts up the number of holes punched each time. If it is even, it ignores the last parity hole, and reads in the number given by the remaining seven holes. If it finds an odd number of holes, however, it complains that there is an error, so that some action can be taken to trap it or correct it.

## More Error Checking

This is all very well so far, but problems can still arise. If a double error occurs in one line, which it will, sooner or later, the tape reader will assume everything is alright, since there will still be an even number of holes. Suppose something has fallen across the tape and covered both of the last two holes of the “E” in figure 4.1, then all it will see is the 4 bit and the 1 bit. It will therefore happily read this as code 5 instead of flagging this as an error, since there is still an even number of holes as far as it is concerned.

In order to get round this, other forms of error checking are also used, such as *checksums*. In this case, every single byte or character is added up as it is read in, and the final total noted. This total is then checked against the total that was recorded when the tape was made. If it agrees, then it is assumed that everything went satisfactorily; otherwise an error is flagged. In practice, it is more convenient to store the checksum in, say, a byte, so every time the value goes over 256, the 256 is thrown away, and the remainder is kept as part of the running total. Mathematicians would call this adding it up *modulo 256* - it just means add it all up, divide by 256 and only keep the remainder.

There are other forms of error checking, but if you want to know more about his, read it up.

## Other forms of Storage

So far, we’ve considered paper tape since it’s a simple medium that’s easy to visualise. Other forms came along, such as punched cards, magnetic tape readers, magnetic discs and laser discs. It’s not too important to worry about the individual methods, but useful to note that there are underlying technologies that are developing all the time. We’ve already seen a mechanical method with the early forms of tape reader, and noted that it gave way to an optical system.

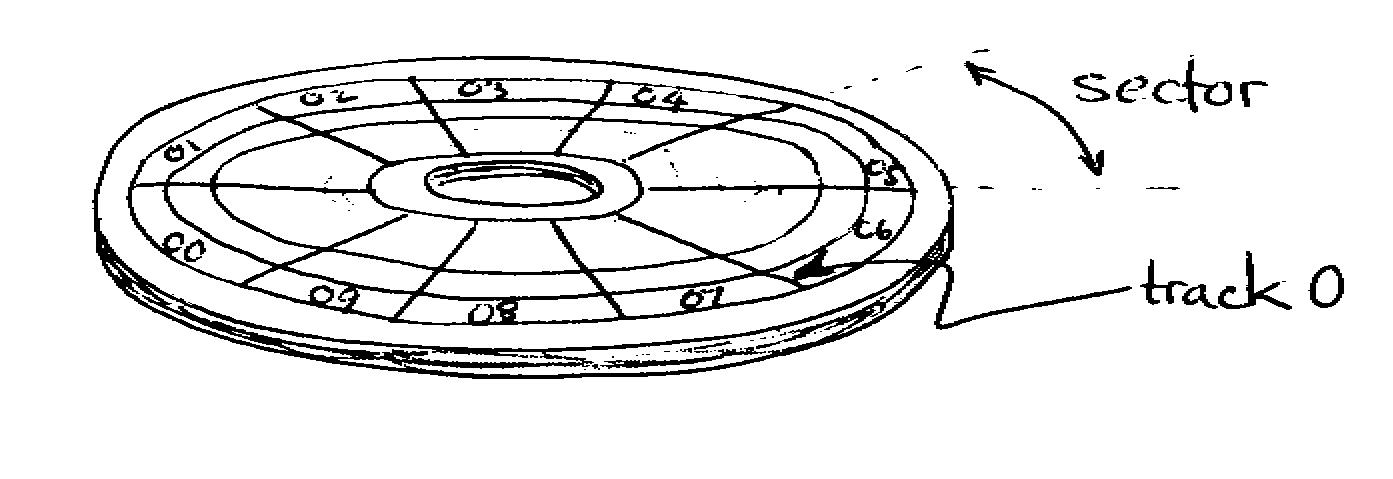
Magnetic methods are more recent, although they’ve still been around a long time, and are still very important for day-to-day work. The audio cassette illustrates the technology well. Basically, it consists of a great length of plastic tape wound round two spools. The tape is covered with a magnetically-sensitive material which is initially randomly magnetised. The tape is passed under an electromagnet which is driven by an audio amplifier, leaving a magnetic pattern that matches the sound signal. When this tape is passed back under the electromagnet, it generates little signals which match the original; they can be fed back into the amplifier and the sound regenerated. Exactly the same can be done with the computer, substituting computer data for the audio signal. Magnetic discs use exactly the same technology; the electromagnet simply travels backwards and forwards across a revolving disc instead of having a length of tape pass underneath it.

CD ROMs are now very popular. They work in the same way as magnetic discs, except they use a reflected laser beam instead of electromagnetism. Since lasers can be extremely finely focused, they are able to store much more information in the same physical space, hence the massive storage capacity of CDs. However, the basic technique used so far has been to burn out the CD when it is first written to, and then read the reflections off the surface to read it back. Clearly, this has a significant disadvantage in that it is not very easy to “re-burn” a CD, and therefore they are not very helpful when it is required to change the stored information a great many times, as conventionally happens with magnetic discs.

## Operation of a “Floppy”

We’re going to look at the construction of a “floppy” disc; “hard” discs work the same way, it’s just that there’s much more of a hard disc. Modern floppies don’t look to be floppy at all; that’s because we see the case rather than the inside. The inside of a floppy disc consists of a truly floppy disc of plastic, coated in magnetically-sensitive material, as we thought before. Hard discs consist of a number of discs of metal, but they’re still coated in magnetically-sensitive material.

When you first get a disc, it is randomly magnetised. Your first job is to format it. This consists of a process of recording onto the disc a pattern of tracks and sectors so that the computer can find its way around the disc. Figure 4.2 shows a simple (and old) system for formatting a disc.



**Figure 4.2: Tracks and Sectors on a Disc**

When you format it, the computer is marking out perhaps 80 tracks in concentric rings around the disc. It is **not** like the old vinyl audio records which had a single track winding round the record from the edge to the middle like a huge spring; these are separate rings. Each ring is then divided up into sectors, ten in our diagram, but frequently many more on current technology. Each sector then consists of a block of information, perhaps containing a thousand or two bytes of information. This means that the computer can zero in on any part of the stored data by specifying a certain track and sector, and get just that little block of data without having to wade through the entire contents of the disc. Incidentally, this is why tapes are used mostly for backups now, since that’s the one time you actually want to go through the entire stored contents in order.

In order to make sure that our disc works correctly, each track and each sector has a little gap all around it, and the first part of every sector has a little label to say which track and sector it is. This means that the disc drive can jump to any place on the disc and read the sector’s header to make sure it’s reached the right place.

## Finding Files on a Disc

We now have to have a method of knowing where our files are stored on the disc. A simple way of doing this is simply to keep a *directory* containing all the names of the files stored on the disc, together with a look-up table of where those files start and end on the disc. Suppose there are three files stored on the disc, *Prog1*, *Prog2*, and *File1*., and that the directory is going to be held in sectors 0 and 1 on track 0 (the outside track). These three *filenames* might be stored in the first three spaces in sector zero, and the corresponding spaces in sector one would contain the start and finish addresses on the disc. So *Prog1* might start at track 0 sector 2, straight after the *catalogue* or directory, and run on to track 1 sector 5. *Prog2* might start at track 1 sector 6, straight after *Prog1*, and go on to track 3 sector 2. *File1* might then start at track 3 sector 3 and go on to track 5 sector 7.

What happens now if *Prog2* is updated? The program is read in to the computer, changed and written back to the disc. However, the new program may be bigger. It was 18 sectors long, if there are 10 sectors to a track; let’s suppose that it is now 21 sectors long. There just isn’t the space to write it back where it was; it must now be put, say, after the end of *File1*. So it is now written back to the disc starting at track 5 sector 8 and going on to track 7 sector 8. Notice that the old copy hasn’t been deleted, but this doesn’t matter, since we now update the directory so that the entry for *Prog2* now gives the new values for the starting and ending sectors.

In passing, we may notice two things here. Firstly, the role of the directory is critical. If the information in the directory is lost, the entire contents of the disc are lost. That’s why it is usual to keep *two* copies of the directory information in modern systems, so that the disc contents can be recovered more easily if one copy gets corrupted. Secondly, old versions may often exist for a while after an update or a deletion - although the computer will often reuse the space fairly quickly!. That’s one reason why it is often possible to undelete files - they don’t disappear straight away after you’ve deleted them. Systems like Windows 95 go even further, of course, and keep compressed copies of deleted files just in case you’ve made a mistake!

This sort of system is all very well for small discs, but hopeless as the discs get bigger - the catalogue rapidly becomes unwieldy. A better approach is to use a tree system which is not unlike the directory system used on most disc systems. Thus there will be an initial block of data which describes the disc and the files in the main directory. However, some of these *files* may themselves actually be sub-directories which contain directory information on their own files. Hence there is a sort of tagging system, where getting to a file consists of finding one directory from another until the file itself is reached.

Modern disc systems include some clever techniques such as recording the shape of the disc, so different disc sizes can be used on the same system, and recording exactly what each sector of the disc contains. In particular, it is possible to mark any sectors which fail as bad sectors, so the disc can continue to be used, even though there may be a fault on it. Otherwise, all the same sorts of checking systems are used as were used for systems like paper tape, such as parity checks and checksums, for example.

## Improving Disc Performance

As program and data files get larger and larger, and as users get more and more demanding, it becomes more important to try and find ways of improving disc performance. One way of improving the speed at which the disc drive can retrieve its files from the disc is to ensure that files that are likely to be needed together (e.g. a wordprocessor program and its wordprocessing files) are stored together on the disc. That way, the read head doesn’t have to travel so far across the surface of the disc.

Another problem concerns wastage of space on the disc. We noted earlier that there was a problem if we changed a file and it was now bigger than before; the space it came out of may not be big enough. Rather than look for a big enough space on the disc, one solution is to allow the file to be split into two parts. The first part goes in the old space until it is filled up, and then another space is found that is big enough to take the rest of the file. This tends to make it possible to get more on the disc, as otherwise, you get left with lots of “holes” where files were, but they are now too big to fit the spaces.

This technique allows you to get more on the disc, but it also creates another problem. If many of the files end up stored in many different parts of the disc, the speed of access goes right down, with the disc drive spending most of its time looking for the next part of the file in a completely different part of the disc. This is a problem of *fragmentation*. For this reason, disc systems will try and recombine files whenever possible.

Another technique which is used to speed up access and reduce storage space requirements is to use *compression* techniques. Suppose that a file has fifteen line feeds all together in it - perhaps you have left fifteen blank lines between paragraphs so you can draw a diagram in afterwards. Rather than store all fifteen linefeeds (15 bytes), you could store the linefeed character, followed by a “x15” character - just two characters. In general, it is possible to reduce the size of files substantially by using such techniques, requiring less storage space, and making it faster to store and load, since there’s less data to transfer.

## Getting into CECIL

That’s all that we’re going to cover in terms of techniques used in hardware design; from here on, we’re going to see how it all works in the context of the CECIL language.

# Chapter 5 The SIM20 - an imaginary computer

## The imaginary machine

CECIL is designed to be the assembly language for the processor of an imaginary computer, called the SIM20 (SIM20 computer has a 10-bit address bus, and a 10-bit data bus. Therefore there are 1024 memory locations in the SIM20, numbered from location 0000 to location 1023.

Each location in the memory of the SIM20 computer can store a single word, consisting of 10 bits — therefore each location in memory can store any number from 0 to 1023 (this is handy, since it means we can store in memory a number corresponding to any address in the SIM20 computer.

The SIM20 computer has ports for input and output. Inputs are:

1. keyboard, and
2. serial port

Outputs are:

1. video,
2. serial port, and
3. parallel port.

By writing to the video port, text output can be sent to the screen. The serial and parallel ports allow the SIM20 computer to communicate with a variety of different devices, such as printers (output), modems (input and output), external CDs (output), and backup devices such as ZIP drives and floppy disk drives (input and output).

## The imaginary processor

At the heart of the SIM20 computer is an imaginary processor, called the SIM20 (SIM20 processor has a set of simple actions it can do, called its “instruction set”. Assembly language programs consist of a sequence of these instructions, and also sets of numbers to be used as “data” for the program to manipulate.

The SIM20 has a number of internal memory stores, or registers, which make calculations simpler to program. There are three registers within the SIM20 processor:

1. the ACCUMULATOR,
2. the X-REGISTER and
3. The Y-REGISTER

Since there are only three registers, many of the instructions for the SIM20 processor allow words of data to be copied between them and the 1024 locations of main memory that the SIM20 computer can address.

Every processor also has a set of “flags”, which are set according to the results of each executed instruction — for example the SIM20 processor has a “zero flag”, which is set whenever a calculation on the ACCUMULATOR results in zero.

In addition to these flags, the SIM20 processor also has an additional register, called the PROGRAM COUNTER. This is a 10-bit word that contains the address of the current instruction being executed (i.e. whereabouts a program running in memory has got to).

## A summary of what’s inside the SIM20 chip

To summarise, the following can be found inside a SIM20 processor:

1. internal stores:   
    ACCUMULATOR  
    X-REGISTER

Y-REGISTER

1. flags  
    zero flag   
    carry flag   
    negative flag
2. program counter

Each of these features of the SIM20 processor is described in detail in later chapters.

## An organised memory

The memory inside the SIM20 computer has a clear organisation. Since the SIM20 computer has a 10-bit address bus, it can address (refer to) 1024 different locations, numbered from 0000 to 1023.

Only a portion of these locations are freely available for storing programs and program data, these are locations 0000 to 0907.

The remaining locations are reserved for a number of uses, many of which are for aiding the communication of the SIM20 computer with hardware devices. Hardware communication locations are required for such things as storing the numbers representing keys pressed on the keyboard, or data being received from a modem, or data to be sent out to a printer. Some locations are reserved for future hardware expansion.

In addition to locations reserved for hardware communication, there is also a set of memory locations reserved for temporary storage of data when the internal ACCUMULATOR and X-REGISTER and Y-REGISTER are insufficient. This store is called a “stack”, and a number of the SIM20 instructions relate specifically to the storing and retrieval of data in the stack.

## Addressable memory

0000

...... User area for programs & data

0905

0906 }

0907 } Timer register

0908

...... Stack

1006

1007 Stack pointer

1008 Interrupt vector

1009 Interrupt enable

1010 Random number register

1011 Pitch register }

1012 Duration register} Sound card

1013 Keyboard port

1014 Keyboard adapter flags

1015 Video port

1016 Video adapter flags

1017 Serial out port

1018 Serial in port

1019 Serial adapter flags

1020 Parallel out port

1021 Parallel in port

1022 Parallel adapter flags

1023 Start vector

## Non-addressable memory (flags and program counter)

It is useful to be able to inspect the contents of the SIM20’s memory, and in particular to see what is currently being held in its various registers and status of its flags. The CECIL compilers provide a facility for doing this. Memory dumps are straightforward; you can specify which locations of memory you want to see, and the computer will print out a table showing the location numbers and the corresponding contents. So, if your program is in locations 0000 to 0023, say, you can print out a table of the actual memory contents.

But you will find it really helpful to see what the various registers and flags contain. This can also be done using the memory dump facility. **The way in which this is done may vary from compiler to compiler**, but the following table shows how one version presents the registers and flags as if they were additional memory.

***address***

1024 Program Counter

1025 Status register

1026 Accumulator

1027 X Register

1028 Y Register

The next table gives the bit numbers for flags in status byte.

1 Zero flag

2 Negative flag

4 Carry flag

# Chapter 6 An Introduction to CECIL

## What CECIL Is

In previous chapters, we’ve mentioned that there are various types of programming language that are used with computers. Most are *high-level* languages; that means that they are designed with the programmer in mind. Some are designed to be easy to learn, like BASIC and LOGO, some are designed to be very powerful, like C, and others might be designed to do certain jobs well, like artificial intelligence and so on.

The computer doesn’t understand these languages; it only understands 0s and 1s. A number which is stored in a byte of memory may be understood in various ways; the computer could understand it to represent an integer, or a floating point number, or an ASCII character, or any of the other things we thought about in chapter two. But it might also understand it to be a *program instruction* - in other words, when the computer reads it, it will take it to be an instruction like *put a copy of this number in the accumulator*, or *store a copy of the accumulator in a given memory location*. As far as we are concerned, in each case all that is stored is a binary number - *there is no difference*. It all depends on the context; if the computer is expecting an instruction, it will try and do it, if it’s expecting a number, it will try and use it as a number. In the case of CECIL, the number 32 can be understood in various ways. If it'’ expecting a number, it will read it as 32. If it’s expecting a character, it will read it as a space character (like pressing the space bar), but if it’s expecting a program instruction, it will do what it says and set the carry flag, since 32 is the CECIL program code to do just that. The important thing to realise is that it’s all just numbers inside the computer.

This means that when you run your Pascal program (or C or Delphi or whatever), the computer must translate the lines of your program into numbers held in its memory which represent the various basic instructions that it understands. A CPU does not understand a “FOR” loop; the rest of this book is designed to help you understand just how much work the computer has to do to construct just one “FOR” loop for you.

We could just write our programs in numbers, using 32 every time we want to set the carry flag - but it is not easy to remember all those numbers! What CECIL does, along with other *assembly languages*, is to allow you to write programs using *mnemonics*, like **cset**, which are rather easier to remember than **32**.

## Interpreters and Compilers

If we are going to write all our programs in mnemonics, then we must have a way of converting them into *machine code* - the numbers that the computer actually understands. The computer program that we use to do this is called a *compiler*. So we now have a text file, which is our original program consisting of lines of commands like *cset* and *add*, and some sort of output file, consisting of the translation into pure numbers that the computer will understand. To distinguish between these, the original program file that we understand is called the *source code*, and the file of numbers (machine code) that the computer understands is called the *object code*. Figure 6.1 shows some source code and the corresponding object code side by side. Don’t worry too much about how it all works just yet, just get an idea of what CECIL is doing when it compiles your source code into object code.

**Program (Source) Code**

.start load data1  
 add data2  
 print  
 stop

**Compiled (Object) Code**

*position file  
in file* *contents*  
0000 1  
0001 6  
0002 5  
0003 7  
0004 21  
0005 38

**Figure 6.1: Source Code and Object Code**

At this point, all that CECIL has done is to convert your language (the CECIL program, or source code) into the computer’s language (the numbers or object code). Notice that the source code has things like formatting (the neat layout) which is meaningful to us, but not to the computer. The object code is just a long string of numbers.

Now, if anything interesting is to happen, the computer must load up your compiled program into memory and actually *do* it. This process is simply running the program. So there are two stages to running most programs; first of all the source code must be compiled, and then the compiled code must be loaded into memory and run as a program.

Some languages work differently, most notably BASIC and LOGO. In these languages, the two processes of compiling and running are merged into one. Hence BASIC will take the first line of your program, compile just that line, and then do it straight away. It then takes the next line of your program, compiles that into machine code, and does that straight away. Languages that work in this sort of way, compiling and running bits of your program all in one go are called *interpreted languages*, and the programs that run them are called *Interpreters*. This might sound like a really good idea - and it certainly makes it much easier to see what is going on if the program goes wrong. However, it is less efficient and takes much longer for the program to run, so most languages are still compiled. Perhaps it is not surprising that interpreters are usually found on teaching languages where performance is not critical, but learning about them is.

## An Example CECIL Program

Figure 6.2 shows an example of a full CECIL program, including all the various bits and pieces it needs to be properly understood, plus a few other things which ought to be included. We’ll look at the structure of the program step by step.

**The Header**

The first part is the *header*. This consists of three lines, beginning *program*, *author*, and *date*. This part is there so that any program can be identified. The three lines must always be present in any program, and must be in that order. You should also be sure to use lower case (no capital letters), since some versions of CECIL may be fussy about this.

The program line is designed to give you pace to give a brief descriptive title. There must be a space, or preferably a tab character which makes the layout neater, following the word *program*; after that, you can write anything you want. But do try and make it relevant!

The author line is a reminder to you to put a name to your work. If your programs are stored on a network, for example, this can be quite important.

The date line should be used to note down when you wrote or last modified this program. Again, you can put anything you like after the word date; you can you any format for the date you like, or could even use it to record a version number if you wished. This line is there to remind you to make a note of which version this particular program is.

In total, the header is not there because it is needed for the computer to run the program, it is used only for administration. It will be printed out when the program is compiled or run, for example, so if there is a lot of output sent to a network printer, you will be able to tell which print-out belongs to you.

program A Simple Program  
author David Argles  
date 30.01.97  
;  
; This is a simple program to add  
; two numbers together which are  
; held in data1 and data2 and print  
; out the answer to the screen  
;  
;--- program starts here---  
;  
.start load data1  
 add data2  
 print  
 stop  
;  
;--- now the data follows---  
;  
.data1 insert 3  
.data2 insert 5  
;  
;---end of source code---

**Figure 6.2: An Example CECIL Program**

**Comment Lines**

Immediately after the header comes some comment lines. These are also ignored by the computer as far as the compiling of the program is concerned. The facility is provided so that you can include notes in your program listing which will remind you what your program is about, but which will not interfere with the compiling of the program.

Different CECIL compilers handle comments in slightly different ways. The safe way to deal with them is to assume that a comment line must begin with a semi-colon, and that, if it does, CECIL will ignore the whole of the rest of the line. So all the lines in figure 6.2 that start with a semi-colon will be ignored by CECIL.

I’ve put a bare minimum of comments in the program above. Following the header, I’ve put a block of comment lines which describe the program that follows in outline. I have also put a comment as the last line indicating that this is indeed the end of the program. That’s a useful check since a line or two at the end can sometimes go over to another page when printing out and get lost or forgotten. I’ve then also split the program up into manageable chunks, in this case just a program chunk and a data chunk.

Programmers can get quite excited about how comment lines should be used (actually, a lot of them can’t be bothered to put them in at all, and then they regret it later!), but the important thing is that you *should use them!!!*

**Program (and Data) Code**

The bit that actually does the work is the smallest part! That’s why programmers can get lazy - but you mustn’t; you get marks for structuring your programs properly! There are just six program lines here, starting with the line *.start*.

To understand this section, you need to think of it as consisting of three *fields*, or columns. The only column that has to have something in is the middle one containing the program instruction (the *instruction field*). So lines three and four of the program itself consist of just *print* in line three, and *stop* in line four. These are instructions that tell the computer to do something.

Some instructions require a bit more information. In line one of the program code, the instruction is *load*. That’s no good unless we also tell the computer *what* to load. We use the third column (*data field*) for that, so line one says that it is the contents of somewhere called *data1* that must be loaded.

That’s all very well, but the computer needs to know where *data1* is. The first column, the *label field*, is used for this. So lines five and six actually contain the data, and the y are marked by using any label you care to make up, in this case data1 and data2. However, CECIL could think that this was meant to be an instruction, since you don’t have to use tabs and spaces to keep the layout neat (though you really ought to). So to distinguish labels from instructions, a dot is put before any label placed in the first column. **Be careful** about this; when you put a label in the first (label) field, you must precede it with a full stop. When the label is used in the third (data) field, you must not put a full stop in - the computer knows it must be a label this time.

## Some Hints

There are a few things worth considering when you write CECIL programs:

1. Be sure to use lower case for CECIL program commands like *load* and *stop*. Some versions of CECIL are case-sensitive.
2. For the same reason, be careful to make sure that you refer to labels in the same way; *Data1* may not be treated as the same thing as *data1*.
3. You will probably find it helpful to set up separate program and data areas in your code
4. Have a clear start point and clear end point to your program, and try and make that there are no “emergency exits” popping out in the middle. Notice that the program in figure 6.2 uses a *.start* label. This is not compulsory, but it is recognised by CECIL as meaning that you want the computer to start at this point. If you don’t include a .start, CECIL will assume that you want to start at the beginning.
5. Use plenty of comments to describe your program. You will be amazed how quickly you forget what you were trying to do when you come back to your program later!
6. Use the date line in the header to record which version you are onto. It is all to easy to keep multiple copies and then forget which one is the latest!
7. Use tabs to lay your program out properly. The computer wont get confused if you don’t - but you will!

## How to Do It

We’re now onto the practicalities of how to actually type in a CECIL program, compile and run it. There are different versions of the CECIL compiler depending on what sort of machine you have. At this point, you need to turn to the appendices and find the introductory tutorial for your computer. There are also instructions for how to get hold of the CECIL compiler. Now turn to the relevant appendix and have a go!

# Chapter 7: Life cycle of a CECIL program



***Figure 7.1: The program life cycle.***

## What this chapter is about

This chapter presents a complete, annotated session with CECIL, from loading a program and changing it, to compiling, debugging and running the program.

The next chapter introduces the programming language concepts of CECIL, while *this* chapter aims to illustrate the stages of a program life cycle as they occur using the CECIL programming environment.

## How this chapter is organised

Figure 7.1 above shows a simple view of the life cycle of a program. The four stages of the program life cycle are:

1. stage 1: program design,
2. stage 2: writing the program
3. stage 3: compiling (and debugging) the program,
4. stage 4: running (and debugging) the program

The first stage, program design, we are assuming has already been completed. Chapter 9 describes in detail how to use the stepwise refinement program design technique for designing CECIL programs.

## The program design

This chapter is based around a program that is to subtract one stored number from another (i.e. 14 - 12), and then display the result in the output window. The program design stage has produced the following low level design:

.start 1 load first number

2 subtract second number

3 display result

4 end program

Stage 2: Writing the program

## Starting CECIL

To start CECIL the following steps need to be followed:

1. close down any application on your computer that is not required   
   (choose “Quit” or “Exit” from these applications’ menus),
2. open the folder where the CECIL application is stored   
   (perhaps on the network, the hard disc or on a floppy),
3. load (and auto-run) the CECIL application.   
   (double click with left mouse button on the CECIL program icon if in Windows)

CECIL should now auto-load ready for you to start work.

## The program

The following program, based on a simple conversion from the pseudocode to CECIL instructions has been created. Note that there are some deliberate mistakes (to be corrected in stages 3 and 4), so please enter the program verbatim (letter for letter, but missing out the line numbers).

line

1 program My first program

2 author Matt Smith

3 date 14 / April / 1996

4

5 ;-- program --

6 .start load num1

7 sub num2

8 print

9 stop

10 ;-- data --

11 .n1 insert 14

12 .n2 insert 12

There are several steps involved here. Don’t forget to follow the relevant introductory tutorial in the appendix for your version of CECIL. Don’t forget to ***!!!SAVE YOUR PROGRAM!!!*** as soon as you’ve typed it in, and do use tabs to lay your program out in a readable way. It will also be useful to make sure that you have gained some familiarity with the editor for your version of CECIL so that you can correct your mistakes as well as the deliberate ones in the program above.

Stage 3: Compiling the program

We now need to compile the program. Either choose **compile only** from the menu, or type in **compile** at the command prompt, depending on which system you are using.

The following should appear as output:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* line

Starting compiler first compilation starts

Compiling program:

my first program 1 picks up “program” text

Author:

matt smith 2 picks up “author” text

Date of program:

14 / April / 1996 3 picks up “date” text

Today's date:

Tue,16 Apr 1996.09:04:46 system date/time

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

label: start, loc: 0 6 records “.start” label   
 as 0

--Label not found: num1 -- 7 not yet met label “num1”

--Label not found: num2 -- 8 not yet met label “num1”

label: n1, loc: 6 11 records “n1” label as 6

label: n2, loc: 7 12 records “n2” label as 7

Some labels not found labels “num1” and “num2”

were not known when compiling

Checking prog second time second compilation starts

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Starting compiler

Compiling program:

my first program 1 finds “program” text

Author:

matt smith 2 finds “author” text

Date of program:

14 / April / 1996 3 finds “date” text

Today's date:

Tue,16 Apr 1996.09:04:46 system date/time

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

label: start, loc: 0 6 “start” matches   
 previous value of 0

--Label not found: num1 -- 7 no previous value   
 for “num1”

--Label not found: num2 -- 8 no previous value   
 for “num1”

label: n1, loc: 6 11 “n1” matches   
 previous value of 6

label: n2, loc: 7 12 “n2” matches   
 previous value of 7

Some labels still not found never did find addresses for

“num1” and “num2” labels

Doing memory dump:

loc: 0 val: 1 6 m/code for “load”

loc: 1 val: -1 6 unknown label “num1”

loc: 2 val: 4 7 m/code for “sub”

loc: 3 val: -1 7 unknown label “num2”

loc: 4 val: 21 8 m/code for “print”

loc: 5 val: 38 9 m/code for “stop”

loc: 6 val: 14 11 value at address   
 labelled “n1”

loc: 7 val: 12 12 value at address   
 labelled “n2”

Starting loc is 0 since a “start” label is  
 known, this is the starting  
 location for the program

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Program failed to compile message stating failure of

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* compilation

The most important message in the output after compilation is the statement that the program has failed to compile. Once this is known, by looking up at the messages for the second compilation, we can hopefully locate and correct the error.

## Identifying the type of error

The first step is to identify what type of error. The two sets of information we have from the second compilation are the messages about each line of compilation, and also the final listing of compiled machine codes and the addresses in which they have been stored.

***Use the messages in the “Output” window to help you identify the error !***

***missing / mis-spelt labels*** Any minus ones (“-1”) in the memory dump of locations after compilation tend to indicate that there is a problems with one or more labels.   
Also, there will be “-- label not found --” messages for the second compilation.

***missing parameters*** Missing parameters are another common error, e.g. forgetting to give a label after an “add” instruction. This may also lead to odd missing label messages, since the next instruction (e.g. “print”) will then be interpreted as the label for the previous instruction missing a label parameter, causing some error messages.

***label used twice*** The same label may not be used to label two different memory locations. Often a problem when using “asc\_X” labels for ASCII data, when a character (X) occurs twice in the word to be stored. Either delete the repeated line, or if different data were labelled the same (e.g. an upper and lower case “D” both labelled “.asc\_d”) choose a different name for the second label.

**WARNING**: some versions of CECIL ignore case, so “.asc\_D” and “.asc\_d” may be treated as the same label.

***A good convention is to always write in lower case***

***mis-spelt documentation text*** Any mis-spelling of the keywords “program”, “author” or “date”, or any text appearing before these words at the top of the program, will cause errors during compilation. Looking at the messages at the beginning of either the first or second compilation should identify this kind of error.

***comments between a label and instruction*** CECIL will not be able to correctly compile a program that has any comments either between a label and an instruction, or between an instruction and its data.

***Always write comments on a line by themselves***

In our program above we get errors for both compilations about missing labels “num1” and “num2”.

## Locating and debugging compilation errors

When we look closer at our use of labels in the program although the “load” and “sub” instructions of lines 6 and 7 in the program refer to labels “num1” and “num2”, the actual data section of the program (lines 11 and 12) use the labels “.n1” and “.n2”.

This bug can be solved by making sure we use the same labels when referring to them in the program instructions and when declaring the data to be stored. Replacing the old lines 11 and 12 with the following should solve the problem.

line

10 ;-- data --

11 .num1 insert 14

12 .num2 insert 12

***NOTE: It is useful when debugging to clear the old contents of the “Output” window before recompiling, by choosing the “Delete all” option from the “Output” window menu.***

When the program is SAVED, then recompiled, the second compilation should be as follows in the output.

label: start, loc: 0 6 “start” matches   
 previous value of 0

label: num1, loc: 6 7 “num1” matches   
 previous value of 6

label: num2, loc: 7 8 “num2” matches   
 previous value of 7

Doing memory dump:

loc: 0 val: 1 6 m/code for “load”

loc: 1 val: 6 6 absolute address 6   
 from label “num1”

loc: 2 val: 4 7 m/code for “sub”

loc: 3 val: 7 7 absolute address 7   
 from label “num2”

loc: 4 val: 21 8 m/code for “print”

loc: 5 val: 38 9 m/code for “stop”

loc: 6 val: 14 11 value at address   
 labelled “num1”

loc: 7 val: 12 12 value at address   
 labelled “num2”

Starting loc is 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Program compiled success this time!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Stage 4: Running the program

Now we need to run the program. Either choose **Run** from the menu, or type **run** at the command prompt.

The output show display the following:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Executing program:

my first program

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Program run completed

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This result is a little surprising, since 14 - 12 is 2 ! However, if we look closer at our program, we can see that a rule of thumb has been broken. The two rules of thumb for addition and subtraction are:

1. when using “add”, always immediately precede the “add” instruction line by the “cclear” instruction,

e.g. cclear  
 add num1

1. when using “sub”, always immediately precede the “sub” instruction line by the “cset” instruction,

e.g. cset  
 sub num2

The reason for this is to do with how the carry bit (which can be set to 0 and 1 by “cclear” and “cset” respectively) is used for addition and subtraction arithmetic in CECIL.

In our program, we have forgotten to precede our “sub num2” instruction line with the “cset” instruction. The program should be changed as follows:

line

1 program My first program

2 author Matt Smith

3 date 14 / April / 1996

4

5 ;-- program --

6 .start load num1

7 cset << insert this line

8 sub num2

9 print

10 stop

11 ;-- data --

12 .num1 insert 14

13 .num2 insert 12

The program should now be SAVED then recompiled, and rerun. The program should now output the correct answer of 2.

## Saving the evidence of a successful run

If you are using the Archimedes version of CECIL, the text in the “Output” window should contain the results of the two compilations, and the output from the running program. By choosing “Save” from the “Output” menu this record of successful compilation and run can be saved as a text file, for printing or inclusion in a word processed document as part of the program documentation. If you are using the IBM DOS version, you will have to enable printing and/or saving of the output by using the **printer** and **spool** commands.

This is very important, since you will need to produce evidence of your compilation and runs when you hand in your work.

# Chapter 8: CECIL language Concepts

## This chapter

This chapter aims to introduce the fundamental CECIL language concepts, necessary for the writing of CECIL programs. It does not duplicate all of the technical detail in the following, reference chapter, therefore these two chapters will need to be used in conjunction with each other.

This chapter first describes each of the different internal and external stores which CECIL programs can access and manipulate. The chapter then describes how simple numeric and character data can be stored using CECIL data declaration instructions. The general form for all CECIL programs is presented. Instructions for input and output are then presented, followed by a look at how data can be moved around memory locations. Then follows description of simple arithmetic. Next the status flags of the SIM20 processor are described. The chapter then goes on to loop at how complex programs can be built up using instructions for changing the linear execution of sequences of instructions, and how loops can be written. Finally more complex representations are considered.

The chapter sections are as follows:

8.1 Data stores and labelled addresses

8.2 Binary, integer and character data types

8.3 The structure of CECIL programs

8.4 Input and output

8.5 Moving data around memory

8.6 Simple arithmetic

8.7 The flags of the SIM20

8.8 Changing the flow of control

8.9 Looping using jumps and the X-register

8.10 Representing complex data

8.1 Data stores and labelled addresses

## The ACCUMULATOR

The majority of processing done by the SIM20 processor works on a single 10-bit internal store, called the ACCUMULATOR. The ACCUMULATOR is like a scratchpad area for the processor — a place to temporarily place a value to be worked on, or to be copied to another store.

The CECIL instructions that work on the ACCUMULATOR can be classified into three categories:

1. those that place a new value into the ACCUMULATOR,
2. those that leave the ACCUMULATOR unchanged, but copy the current value into a store elsewhere,
3. those that perform an arithmetic calculations on the value of the ACCUMULATOR.

After any arithmetic operations on the ACCUMULATOR there are a number of “flags” that are set.

A simple program to demonstrate the central role of the ACCUMULATOR is as follows:

program simple number display

author Matt Smith

date 15 / April / 1996

;-- program --

.start load num1 ;load value at *num1* (63) into *ACC*

add num2 ;add value at *num2* (2) to *ACC*

print ;display value of *ACC* (65)

printch ;display character whose ASCII

;code is in *ACC* (“A”)

stop ;end program

;-- data --

.num1 insert 63 ;store 63 at address labelled *num1*

.num2 insert 2 ;store 2 at address labelled *num2*

As is described in the comments for the program above, all the work going on works with the ACCUMULATOR, such as loading values in from addressable memory, doing arithmetic, and displaying numbers and characters to the output window.

## Addressable memory

The SIM20 computer has 1024 locations of addressable memory — locations 0000 to 1023. By “addressable” it is meant that the SIM20 processor can retrieve data from the locations, and place data into the locations.

The locations from 0000 to 0907 are purely for user programs and data. Other locations (up to 1023) have special uses, but can still be addressed by the SIM20 computer.

Instructions retrieving a copy of data from an address (but not changing the stored value) include:

load, xload, loadmx, add, sub, comp, xcomp

Instructions storing data into memory are:

store, xstore

Locations in memory where program instructions and data are stored can be given names, both for easy reference in the program, and to avoid the need for absolute addressing. Without labels many program instructions would have to refer to exact address locations in memory, and each time a program was changed, or more data added, all the references to changed locations would have to change.

For example, the program below refers to the absolute address where the data item “42” is stored (in location 0004). The line numbers have been added for easy reference, they are not required when typing programs into CECIL.

Line

1 program absolute address

2 author Matt Smith

3 date 15 / April / 1996

4

5 ;-- program --

6 .start load 4

7 print

8 stop

9

10 ;-- data --

11 insert 42

The compiled version of the code is as follows:

address value

0000 27 (machine code for “load”)

0001 4 (absolute address, location 0004)

0002 28 (machine code for “print”)

0003 36 (machine code for “stop”)

0004 42 (data, integer 42)

The program begins at line 6 (compiled into addresses 0000 and 0001), with a statement instructing CECIL to place the number stored at memory location 0004 into the ACCUMULATOR. If we look at the compiled version of the program above, we can see that address 0004 contains the number 42. So 42 is placed in the ACCUMULATOR.

Line 7 (compiled into addresses 0002) instructs CECIL to display in the output window the current value in the ACCUMULATOR. So “42” is displayed in the output window. The “**stop**” instruction at line 8 (compiled into address 0003), tells CECIL to terminate the program’s execution. Line 11 declares the data item “42”, and is compiled into address 0004.

If the program were changed to print the number twice, the absolute address of the data item “42” will change (i.e. from 0004 to 0005), therefore the reference to the data’s address in the “**load**” statement needs to be changed.

The revised program, and its compiled form are listed below.

line

1 program absolute addresses

2 author Matt Smith

3 date 15 / April / 1996

4

5 ;-- program --

6 .start load 5

7 print

8 print

9 stop

10

11 ;-- data --

12 insert 42

The program above compiles to:

address value

0000 27 (machine code for “load”)

0001 5 (machine address, location 0005)

0002 28 (machine code for “print”)

0003 28 (machine code for “print”)

0004 36 (machine code for “stop”)

0005 42 (data, integer 42)

To solve the problem of having to keep changing the absolute address references to data (or program instructions), all assemblers[[1]](#footnote-1) allow the use of textual “labels”. Labels are character strings (e.g. “.start”, “.loop”, “.num1”, “.asc\_y”), which are used in two ways for programming:

1. a label is placed next to a particular program instruction or data declaration instruction, and that label is associated with the address of that instruction once the program has been compiled (e.g., the “.start” label is always placed by the first program statement, which is usually compiled into location 0000, as in “**.start load 5**” in the program above).

***NOTE: the full stop is used to declare a label***

1. any program instructions which need to refer to an address (e.g. “**load**”) can have in their statement either an absolute address (e.g. “**load 4**”), or they can have a label as a parameter (e.g. “**load num1**”), and at the time of compilation, the address of the instruction that is labelled will automatically replace the label (e.g. “**load num1**” would become “27 12”, where “27” is the machine code for “**load**”, and assuming “12” was the address 0012 associated with the label “**num1**”).

***NOTE: the full stop is omitted when referring to a labelled address***

Labels are placed before the progam or data declaraction instruction that are to be associated with, and are a string of characters, with a full-stop prefix. Note that there are no spaces allowed in a label, or after the full stop, but the use of the underscore character “\_” is permitted. Examples include:

**.num1 .start .asc\_a .subroutine1 .name2**

A program to illustrate labels is given below:

line

1 program labelled addresses

2 author Matt Smith

3 date 15 / April / 1996

5 ;-- program --

6 .start load num1

7 print

8 load asc\_a

9 printch ;display char whose ASCII code

10 ;is in ACCUMLATOR

11 stop

13 ;-- data --

14 .num1 insert 42

15 .asc\_a insert 65

The program above compiles to:

address value

0000 27 (machine code for “load”)

0001 7 (address associated with “.num1”, 0007)

0002 28 (machine code for “print”)

0003 27 (machine code for “load”)

0004 8 (address associated with “asc\_a”, 0008)

0005 32 (machine code for “printch”)

0006 36 (machine code for “stop”)

0007 42 (data, integer 42)

0008 65 (data, ASCII code for “A”)

label “.num1” associated with address 0007

label “.asc\_a” associated with address 0008

## The X-register

The X-register is another 10-bit word store, just like the ACCUMULATOR. It is very useful for storing data needed for the running of a program, while the data that is being worked upon is stored in the ACCUMULATOR. For example, when writing a program that has a loop in it, the number of times the loop has been executed is usually stored in the X-register.

For a number of the instructions that work on the ACCUMULATOR, there are X-register equivalents:

**load <label>** copy value from labelled address into ACCUMULATOR

**xload <label>** copy value from labelled address into X-register

**store <label>** replace old value at labelled address with a copy of value in ACCUMULATOR

**xstore <label**> replace old value at labelled address with a copy of value in X-register

**push** place a copy of value in ACCUMULATOR onto top of stack

**xpush** place a copy of value in X-register onto top of stack

**pull** replace ACCUMULATOR with copy of value from top of stack, and remove this top value from the stack

**xpush** replace X-register with copy of value from top of stack,   
and remove this top value from the stack

**comp <label>** compare the value at the labelled address with the ACCUMULATOR, and set the flags accordingly

**xcomp <label>** compare the value at the labelled address with the X-register, and set the flags accordingly

In addition there are two instructions that only affect the X-register:

**xinc** add 1 to the value in the X-register (and set flags accordingly)

**xdec** subtract 1 from the value in the X-register (and set flags accordingly)

Note that the above two instructions (“xinc” and “xdec”) will result in a resetting of the carry, zero and negative flags, so when used in loops, conditional jumps based on the flags must be carefully written to make sure they respond to flags set after the planned arithmetic operation.

## The stack

The stack is a temporary store in addressable memory (locations 0908 to 1006). Stacks are a LIFO data structure, meaning “Last In First Out”. The way they work is that only the most recent piece of data added to the stack is accessible. It is likened to a “stack” of books, where only the top book can be taken off the pile at any one time.

There are two operations that can be done with stacks:

1. push the “push”ing of a number onto the top of the stack,
2. pull the “pull”ing off, and storing of the top number from the stack.

So we could have an empty stack and then do the following series of operations on it:

push 10, push 14, pull, push 6, pull, pull

The result would be the following changes to the stack:

description stack value from pull

before operations empty

after “push 10” (bottom) 10 (top)

after “push 14” (bottom) 10 14 (top)

after “pull” (bottom) 10 (top) 14

after “push 6” (bottom) 10 6 (top)

after “pull” (bottom) 10 (top) 6

after “pull” empty 10

All stack operations work with either the ACCUMULATOR or the X-register. The three push instructions are:

push push a copy of the number in the ACCUMULATOR onto the top of the stack,

xpush push a copy of the number in the X-register onto the top of the stack.

ypush push a copy of the number in the Y-register onto the top of the stack.

The two pull instructions are:

pull pull a copy of the number on the top of the stack into the ACCUMULATOR (and remove that number from the stack),

xpull pull a copy of the number on the top of the stack into the X-register (and remove that number from the stack),

ypull pull a copy of the number on the top of the stack into the Y-register (and remove that number from the stack),

As one might expect, two types of error can occur:

stack underflow error this error occurs when a “pull” or “xpull” or “ypull” instruction is executed when the stack is empty (i.e. no value to pull),

stack overflow error this error occurs when a “push” or “xpush” or “ypush” instruction is executed, and the stack is full (i.e. all reserved addresses for the stack data are full).

8.2 Binary, integer and character data types

CECIL is designed to work with two type of data: integers (whole numbers), and ASCII characters. In fact all work in assembly language is with integers, however, CECIL has a “built in” ASCII table which allows easy input and output of ASCII characters.

### ASCII characters and codes

There are only two program instructions, and one data declaration instruction for working with ASCII data, they are as follows:

**printch** this instruction displays in the output window the character whose ASCII code is currently in the ACCUMULATOR (so if “65” were in the ACCUMULATOR then an upper case “A” would be displayed in the output window,

**getkey** this instruction will wait until a key has been pressed by the user, and then places in the ACCUMULATOR the ASCII code corresponding to the pressed key,

The following program demonstrates the use of these two instructions:

line

1 program demo of ASCII instructions

2 author Matt Smith

3 date 15 / April / 1996

4

5 ;-- program --

6 .start load prompt ;load ASCII code for

7 ;prompt into ACCUMLATOR

8 printch ;display char for prompt

9 getkey ;wait until key pressed,

10 ;and put ASCII code in ACC

11 printch ;display char just pressed

12 stop

13

14 ;-- data --

15 .prompt insert “>“ ;character to use as a prompt

16 ;in the output window

The program above compiles to:

address value

0000 27 (machine code for “load”)

0001 6 (address associated with “prompt”, 0006)

0002 28 (machine code for “printch”)

0003 45 (machine code for “getkey”)

0004 28 (machine code for “printch”)

0005 36 (machine code for “stop”)

0006 62 (data, ASCII code for “>”)

label “prompt” associated with address 0006

### Binary integers in CECIL

Although, from the point of view of the SIM20 processor, all data in the computer is stored in terms of 10-bit binary words, only a small numbers of CECIL instructions work on data in its binary form specifically. These include the straightforward “printb” instruction (which displays the current contents of the ACCUMULATOR as a list of each of the ten bits into the output window), “lshift” and “rshift” bit manipulation instructions, and the logical instructions such as “and” and “not”.

### Decimal integers in CECIL

Even though all numbers are stored as 10-bit binary words in CECIL, when writing CECIL programs one can think of the numbers in decimal (i.e. 4 whether considered in binary or decimal is still the number 4). All data declarations using the “word” instruction are decimal numbers, and both the “print” and “printd” instructions output decimal numbers to the “Output” window. So unless working with any binary specific instructions, one does not need to often consider the fact that all CECIL operations work on numbers in their binary form.

8.3 The structure of CECIL programs

CECIL programs have to begin with three lines:

**program <program description>  
author <author’s name>  
date <date program written>**

Although it is then possible to have a sequence of instructions and data mixed, good programming practice says that the remainder of the program should fall into two distinct sections, an instruction section and a data section. Which comes first doesn’t really matter. In this book a convention of instruction section followed by data section has been adopted.

Both instruction and data sections can be thought of as having three columns:

<optional label> <instruction or data declaration> <label or data>

Some program instructions require the third column, such as:

**add num1  
 sub num2  
 load space  
 store userkey**

Other instructions do not have an associated label, such as:

**print  
 stop  
 wait  
 push**

The CECIL data declaration instruction is:

**insert <data> (e.g. insert 42)**

So a CECIL program as a whole can be considered as being in a 3 by 3 grid as follows:

|  |  |  |  |
| --- | --- | --- | --- |
|  | col 1  label declaration | col 2  instruction | col 3  label ref. or data |
|  |  |  |  |
| 1. documentation text |  | **program** | **test program** |
|  |  | **author** | **Matt Smith** |
|  |  | **data** | **2 / May / 1996** |
|  | **;-- program --** |  |  |
| 2. instruction section | **.start** | **load** | **num1** |
|  |  | **print** |  |
|  |  | **stop** |  |
|  | **;-- data --** |  |  |
| 3. data section | **.num1** | **insert** | **42** |

8.4 Input and output

Typed input to a CECIL program is only possible through a single instruction: “getkey”. This instruction when met in a program will make the program wait until a key is pressed, and then store the ASCII code of the pressed key in the ACCUMULATOR.

A typical program using getkey is:

program example of keyboard input

author Matt Smith

date 2 / May / 1996

.start getkey ;wait until a key is pressed

;(and place ASCII code in ACC)

printch ;display the character in Output win.

stop

There are two important places to which CECIL programs can send output:

1. the “Output” window,
2. the parallel port interface.

### Sending output to the “Output” window

The following instructions are ways CECIL can send output to the “Output” window:

**print** displays the contents of the ACCUMULATOR as a decimal (denery) integer in the “Output” window

**printb** displays the contents of the ACCUMULATOR as a 10-bit binary word in the “Output” window

**printch** displays the ASCII character corresponding to the integer in the ACCUMULATOR in the “Output” window

**printd** displays the integer corresponding to a 20-bit word, whose low- word address is stored in the ACCUMULATOR, and whose high-word address is one more that the value in the ACCUMULATOR.

### Sending output to the parallel port interface

CECIL programs can send 8-bit binary words to the parallel port by storing numbers at address 1020 (the parallel output register). Output is limited to 8-bits due to the physical restriction of I/O being a maximum of 8-bits on the I/O board.

e.g. load num1 ;load a number into ACCUMULATOR  
 store 1020 ;store number in parallel I/O register

8.5 Moving data around memory

There are number of reasons one may wish to move data between different locations in memory. Since most arithmetic operations use the current value in the ACCUMULATOR, there is a need to move data between labelled addresses and the ACCUMULATOR.

Consider the following program design:

.start 1 load 7 into ACCUMULATOR

2 add 100 to the number (7 + 100)

3 display the result (107)

4 put cursor onto a newline

5 add 100 to the previous result (107 +100)

6 display the new total (207)

7 stop

In order to implement step 4 in the above design, the ASCII code for a linefeed (i.e. 10) needs to be placed into the ACCUMULATOR, and then a “printch” instruction executed. However, in doing so, the result of 17 will have been overwritten with the ASCII value of 10.

The solution is to temporarily store the result of the first calculation somewhere, while displaying the newline character, then copying the result back into the ACCUMULATOR, for the rest of the calculation to be completed and printed.

There are 3 common ways to implement this design:

1. use the stack (i.e. after displaying the result of the first calculation (step 3), “push” the current ACCUMULATOR value onto the stack, then display the newline, then “pull” the value back from the stack to complete the calculation),   
   e.g. load num1  
    cclear  
    add hundred  
    print  
    push ;push total onto stack   
    load newline  
    printch  
    pull ;pull total back from stack   
    <continue>
2. an extra labelled memory location can be declared in the program, which is used as a temporary store for the total

e.g. load num1  
 cclear  
 add hundred  
 print  
 store temp ;store total in “temp”   
 load newline  
 printch  
 load temp ;retrieve total from “temp” ---<continue>---  
 ;-- data --   
 .num1 insert 7  
 .hundred insert 100  
 .newline insert 10  
 .temp insert 0 ;initial value is just a dummy

1. (this one is a rather messy solution, since location “.num1” is used for storing two different things)   
   since the original number is not used after initially being loaded in, this labelled location can be used as the place to store the initial result while displaying the linefeed,

e.g.load num1  
 cclear  
 add hundred  
 print  
 store num1 ;i.e. store total in “num1”  
 load newline  
 printch  
 load num1 ;now retrieve total from num1  
 ----<continue>----  
 ;-- data --   
 .num1 insert 7 ;used for both initial value 7  
 ;and temporary store for total

In a similar way (although less often needed) values can be moved between the X-register and memory locations or the stack using the corresponding X-register instructions:

**xload, xstore, xpull, xpush**

8.6 Simple arithmetic

There are only 4 instructions for performing arithmetic in CECIL:

**1 xinc** add 1 to the value in the X-register,

**2 xdec** subtract 1 from the value in the X-register,

**3 add <label>** add the number at the labelled address to the current ACCUMULATOR value (placing the result in the ACCUMULATOR),

**4 sub <label>** subtract the number at the labelled address from the current ACCUMULATOR value (placing the result in the ACCUMULATOR).

Due to the method of binary implementation of the “add” and “sub” instructions, which uses the carry flag, a “cclear” instruction should always precede the “add” instruction, and a “cset” should always precede the “sub” instruction.

A simple program to illustrate addition and subtraction is as follows:

program example of addition and subtraction

author Matt Smith

date 5 / May / 1996

;-- program --

.start load num1 ;load 7 into ACC

cclear

add num2 ;add 4, so 11 now in ACC

print

cset

sub num3 ;subtract 2, so 9 now in ACC

print

stop

;-- data --

.num1 insert 7

.num2 insert 4

.num3 insert 2

8.7 The flags of the SIM20

There are 3 flags in the SIM20, which are set according to the results of any arithmetic calculations performed on either the ACCUMULATOR or the X-register, i.e. the flags are rest after any of these instructions:

xinc, xdec, add, sub, comp, xcomp

Also any binary arithmetic operations will cause the flags to be reset:

and, or, eor

In addition, the two binary shift instructions will cause the carry flag to be set:

lshift, rshift

Finally, the carry flag can be explicitly set to 0 or 1 by the instructions “**cclear**” and “**cset**” respectively.

### The flags

Each of the three flags is a single bit, i.e. can store the number 0 or 1. The value of the flags represent the (boolean) truth values of TRUE (1) and FALSE (0).

### The Zero flag

The zero flag is set (i.e. 1 = “TRUE”) when the result of an arithmetic operation is zero. The zero flag is also set when two values being compared using either the “**comp**” or “**xcomp**” instructions are the same. Otherwise the zero flag is clear (i.e. 0 = “FALSE”).

### The Negative flag

The negative flag is set (i.e. 1 = “TRUE”) when the result of an arithmetic operation is zero. The negative flag is also set when the value at the labelled address is larger the value in the ACCUMULATOR or X-register when using the “**comp**” or “**xcomp**” instructions respectively. Otherwise the negative flag is clear (i.e. 0 = “FALSE”).

### The Carry flag

The carry flag is set (i.e. 1 = “TRUE”) when the result of an arithmetic operation is too large to be stored in the 10-bits of the ACCUMULATOR (or X-register, if the arithmetic was being performed on that). For example, if a calculation adding 1000 to 40 were performed, the carry flag would be set (since 1023 is the largest number that could be stored in the 10-bits of the ACCUMULATOR).

The carry flag is also set when the bit being lost due to an “**lshift**” or “**rshift**” is a 1 (i.e. when the left-most bit was a 1, before and “**lshift**”, or when the right-most bit was a 1 before an “**rshift**”).

The carry flag is also always set after subtractions when the result is zero or negative (this is because of how it is used during addition and subtraction).

The carry flag is used for addition and subtraction, and can be explicitly set (to 1) or cleared (to 0) by the instructions “**cset**” and “**cclear**” respectively.

8.8 Changing the flow of control  
(or how to write procedures and “IF”s in CECIL)

It is possible to write CECIL programs that do not simply execute each statement in the program section of the program in a linear sequence, this can be achieved by the use of “jump” instructions.

There are two types of jump instruction:

1. unconditional jumps,
2. conditional jumps.

### Unconditional jumps

An unconditional jump will ALWAYS cause the flow of control to change — i.e. the jump will always be executed and the program counter changed to the labelled location.

There are two instructions for unconditional jumps:

**jump <label>** this instruction causes a simple change of control whereby the program counter is set to the labelled address, and program execution continues from there,

**jmptosr <label>** a “jmptosr” jump also causes a change of the program counter, and an unconditional jump to the labelled address, BUT it also records the value of the program counter before making the jump. “jmptosr” stands for “jump to subroutine”, and should be used in conjunction with the “return” instruction. Since after a “jmptosr”, when the next “return” instruction is executed the original program counter is retrieved, and program execution continues from where it left off.

Example of “jump” (display “a”, then jumps to a routine to display “b”, then jumps back to part of the main program to display a “c” and then stop):

program example of jump

author Matt Smith

date 5 / May / 1996

;-- program --

.start load asc\_a

printch

jump print\_b

.main load asc\_c

printch

stop

;-- routine to display a “b” --

.print\_b load asc\_b

printch

jump main

;-- data --

.asc\_a insert 97

.asc\_b insert 98

.asc\_c insert 99

Example of “jmptosr” (much neater than the above, this program displays an “a”, then jumps to a subroutine to display a newline, returns from that subroutine, displays a “b”, jumps back to the subroutine to display newline again, then returns again to finally display a “c” and then stop):

program example of jmptosr

author Matt Smith

date 5 / May / 1996

;-- program --

.start load asc\_a

printch

jmptosr newline

load asc\_b

printch

jmptosr newline

load asc\_c

printch

stop

;-- routine to display a newline --

.newline load asc\_newline

printch

return

;-- data --

.asc\_a insert 97

.asc\_b insert 98

.asc\_c insert 99

As can be seen, the use of “jmptosr” and “return” is more desirable, since the program still maintains some overall top-level structure, and is much easier to follow when debugging or modifying. Also, the “return” can return to any place the subroutine was called from, it allows the **reuse** of the same subroutine.

An additional benefit is that fewer labels are needed, again keeping the program simpler and easier to follow.

### Conditional jumps

A conditional jump will only cause the flow of control to change if some CONDITION is true. All three of the conditional jump instructions available in CECIL work according the values of the different flags that are set after arithmetic operations on either the ACCUMULATOR or X-register.

NOTE: Each conditional jump behaves like a “jump” if its condition is TRUE — i.e. “return” cannot be used with a conditional jump.

There are three instructions for conditional jumps:

**jizero <label>** this instruction causes a simple change of control whereby IF THE ZERO FLAG IS SET (“1”) then the program counter is set to the labelled address, and program execution continues from there,

**jineg <label>** this instruction causes a simple change of control whereby IF THE NEGATIVE FLAG IS SET (“1”) then the program counter is set to the labelled address, and program execution continues from there,

**jicarry <label>** this instruction causes a simple change of control whereby IF THE CARRY FLAG IS SET (“1”) then the program counter is set to the labelled address, and program execution continues from there,

**jipos <label>** this instruction causes a simple change of control whereby IF BOTH THE NEGATIVE AND ZERO FLAGS ARE CLEAR (i.e. “0”) then the program counter is set to the labelled address, and program execution continues from there.   
***NOTE: Remember, there is no “positive flag”, but if neither the negative nor zero flags are set, then the result of the last arithmetic operation must have been positive***

If the condition for conditional jump is not true, then the jump instruction is ignored, and the program counter is simply incremented by 1 as usual (and the next instruction is executed).

Example of “jineg” and “jipos” (this illustrates how “IF” type constructs can be implemented in CECIL):-

program example of jineg

author Matt Smith

date 5 / May / 1996

;-- *program* --

.start load num1

cset

sub num2

;-- *IF result negative THEN* --

;-- *jump to display “-”* --

jineg print\_neg

;-- *IF result positive THEN* --

;-- *jump to display “+”* --  
 jipos print\_pos

;-- *ELSE display “0”* --

load asc\_0

printch

.end stop

;-- *routine to display a “-” sign* --

.print\_neg load asc\_negsign

printch

jump end

;-- *routine to display a “+” sign* --

.print\_pos load asc\_possign

printch

jump end

;-- *data* --

.num1 insert 20

.num2 insert 77

.asc\_0 insert “0”

.asc\_negsign insert “-”

.asc\_posssign insert “+”

The above program also demonstrates an important programming convention:

***every program ought to have a single start point and a single end point***

Thus, although both the “print\_neg” and “print\_pos” routines could have ended with their own “stop” instruction, the program is much easier to understand and follow as above, when each routine jumps back to a final “end” label. If the program were needed to be changed at a later date to do some additional processing, the additional code could be added at the point in the main program labelled “end”, whereas had each routine had its own “stop” then either lots of code would have to be duplicated throughout the program, or even more jump instructions would have to be added, making the program much less easy to understand, debug or extend.

## A useful technique

One program associated with using jump instructions is that often the actions performed in a routine that is jumped to use the ACCUMULATOR. This is especially a problem when often calling a subroutine using “jmptosr” and “return” (for example to display newline characters).

A solution to this problem is careful use of the stack. The pseudocode for this technique is as follows:

.subroutine\_label 1 push current ACCUMULATOR value onto stack

2 <do main subroutine processing>

3 pull back old ACCUMULATOR value from top of stack

4 return to place subroutine was called from

When writing subroutines this way, the subroutine processing can use the ACCUMULATOR as much as one might wish, but to the place the subroutine is called from it is as though the ACCUMULATOR has not be touched. A simple program to illustrate this is given below.

program example of a subroutine leaving ACC alone

author Matt Smith

date 5 / May / 1996

;-- *program* --

.start load num1

;-- *display first num* --

print

;-- *print newline* --

jmptosr newline

cclear

;-- add “num2” --

add num2

;-- display new total --

print

;-- *print newline* --

jmptosr newline

cset

;-- *sub “num3” from total* --

sub num3

;-- *display final total* --

print

stop

;-- *routine to display a newline* --

;-- *first save old value of ACC* --

.newline push

load asc\_newline

printch

;-- *retrieve old value of ACC* –

pull

return

;-- *data* --

.num1 insert 8

.num2 insert 15

.num3 insert 3

.asc\_newline insert 10

**NOTE: Although the ACCUMULATOR is effectively unchanged by the subroutine, the zero, negative and carry flags may have been changed, so care should be taken when using conditional jumps soon after calls to subroutines — generally conditional jump statements should be placed immediately after the arithmetic operations they are to respond to.**

8.9 Looping using jumps and the X-register

A loop can be implemented reasonably straightforwardly using a combination of the X-register and conditional jumps.

A simple example might be to display a character 3 times. The number of times to loop is (another programming convention) placed in the X-register. The ASCII code of the character to display is loaded into the ACCUMULATOR. After each time the character is displayed, an “xdec” is executed, and if the zero flag is set a conditional jump out of the loop can occur, otherwise a jump statement can be used to execute the loop instructions again.

A program to display the letter “F” three times is as follows:

line

1 program example of a loop

2 author Matt Smith

3 date 5 / May / 1996

4

5 ;-- program --

6 .start xload counter

7 load asc\_f

8 .loop printch

9 xdec

10 jizero end

11 jump loop

12 .end stop

13

14 ;-- data --

15 .counter word 3

16 .asc\_f insert “F”

The program would execute as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| after line | ACC | X-register | Comment |
| 6 | ?? | 3 | 3 has been loaded into X-register |
| 7 | 70 | 3 | ASCII for “F” (70) has been loaded into ACC |
| 8 | 70 | 3 | “F” displayed in “Output” window |
| 9 | 70 | 2 | X-register has been decremented by 1 |
| 10 | 70 | 2 | the zero flag was not set (0), so conditional jump not executed |
| 11 | 70 | 2 | the unconditional jump back to line 8 is executed |
| 8 | 70 | 2 | “F” displayed in “Output” window |
| 9 | 70 | 1 | X-register has been decremented by 1 |
| 10 | 70 | 1 | the zero flag was not set (0), so conditional jump not executed |
| 11 | 70 | 1 | the unconditional jump back to line 8 is executed |
| 8 | 70 | 1 | “F” displayed in “Output” window |
| 9 | 70 | 0 | X-register has been decremented by 1 |
| 10 | 70 | 0 | the zero flag was set (1), so conditional jump to line 12 was executed |
| 12 | 70 | 0 | program stops |

One of the Appendices goes into detail about how X-register loops can be used to display strings of any length.

8.10 Representing complex data

When numbers are too large to fit into 10-bits (i.e. numbers bigger than 1023), the only solution is to start “sticking” words together. By treating two words as if they are the two halves of a “super” word, we can then work with numbers that can be represented with 20-bits.

CECIL only has one instruction to illustrate how to work with multi-word numbers: “**printd**”. To understand how this instruction works one must consider what CECIL needs to know when working with multi-word numbers.

Convention states that we call the left-most 10-bits (bits 10..19) of a double word number the “high” word, and the right-most 10-bits (bits 0..9)the “low” word (since the “high” word bits represent the higher numbers).

Two values stored physically next to each other in the computer are called “contiguous”, the “**printd**” instruction expects the address of the low-word to be in the ACCUMULATOR, and the high-word to be the next higher contiguous address.

To represent the value 1062, our double word number would looks as follows:

High Word

524288 262144 131072 65536 32768 16384 8192 4096 2048 1024

0 0 0 0 0 0 0 0 0 1 = 1024

(as decimal this high-word is equal to 1)

Low Word

512 256 128 64 32 16 8 4 2 1

0 0 0 0 1 0 0 1 1 0 = 38

(as decimal this low-word is equal to 38)

So TOTAL = 1062, since 1x (1024) + 38 = 1062

line

1 program Double word example

2 author Matt Smith

3 date 5 / May / 1996

4

5 ;-- *data* --

6 .low insert 38

7 .high insert 1

8 .loc\_double insert 0

9

10 ;-- *program* --

11 ;-- *first load location of low* --

12 ;-- *word into ACCUMULATOR* --

13 .start load loc\_double

14 ;-- *display value, whose* --

15 ;-- *low word is in* --

16 ;-- *ACCUMULATOR, and whose* --

17 ;-- *high word is in address* --

18 ;-- *following low word* --

19 printd

20 stop

The number 1062 should be displayed in the “Output” window.

Note that the program needs to load the address of the the low word into the ACCUMULATOR — for this reason the address if the low word needs to be known, so the data section of the program is placed first, so the address of the low word is guaranteed to be zero. This is perhaps the only time that one needs to “hard code” an absolute address in a program (i.e. the “loc\_double” label labels an address where the absolute address of 0000 is stored — the address of the low word).

When implementing arithmetic routines to work with double word numbers one needs to work with the low-words of each number first, and then add in the carry bit to the high words,

e.g.

NUM1 + NUM2, do the following

low word of RESULT = low word of NUM1 + low word of NUM2

high word of RESULT = carry flag (bit) of low word addition + high word of NUM1 + high word of NUM2

Chapter 9:  
Designing and Writing a Program

## What this chapter is about

This chapter presents the steps in the design of a simple CECIL program. It then goes on to illustrate some more advanced CECIL features by presenting a number of refinements to the program. Some of the refinements are a little unnecessary for the simple addition program presented in this appendix, but are included to present features of the language in context.

This chapter aims to illustrate the steps of software design and implementation as they occur using the CECIL programming environment.

## How this chapter is organised

This chapter is broken down into stages and steps. Program design can be considered to fall into four steps:

1. step 1: **written** program specification,
2. step 2: initial program design,
3. step 3: detailed program design,
4. step 4: coding (implementation) of design into CECIL instructions.

There is also a fifth step in this case:

1. step 5: refinement(s) of the program design.

STEP 1: Written program specification

## The need for a specification

Before program design can begin, a specification of what the program has to do is required. Remember, a program specification states *what* a program is to do, a program design states *how* a program will achieve the program specification.

It is **very** important that before any program design or implementation goes ahead, the specification is represented in a **written** form. This helps the program design process in two ways especially:

1. ***writing a program specification focuses on precisely what the program is to do***   
   (the discipline of having to write a specification helps to identify what the inputs and outputs really are, and the relationships between them; it also helps in the identification of any unusual circumstances which the program ought to be able to cope with; and the requirement of stating the form of the user interface helps everyone develop a feel for what the program should do; if possible, examples of data, before and after processing should be presented, and examples of the actual form of the user interface for example data), and
2. ***a dated, written specification is unchanging***   
   (most of the problems met when developing software in a non-rigorous fashion come from changes made to the specification during the actual design and implementation process — such problems can be eliminated if a written specification is created before design and coding, and this specification is agreed upon between the final user, and the software designers and developers).

Before doing anything else, we need a program specification. For large programs there may be all sorts of specific details and restrictions on what the program is to do, but it should always be possible to summarise what a program does in a few sentences.

## The specification

Our program specification is as follows:

***“Write a program to input two numbers, and display their total.”***

Although this might seem a very simple program to write, in a low level language like CECIL even simple actions require a number of steps be performed.

We will add a couple of assumptions to make our program easier to design:

1. assumption 1: each number will be a in the range 0..9   
   (i.e. only a single keypress is needed for each number)
2. assumption 2: the user will press no key apart from 0..9   
   (i.e. we don't need to add any error checking routines for bad input)

We will also specify a very simple user interface

1. each number will be prompted for by the words “1st” and “2nd”
2. the result will be displayed in the form “N1 + N2 = TOTAL”   
   (where N1 and N2 represent the two integers, and TOTAL represents their total)

An example of what the user would see could be:

1st  
 4  
 2nd  
 3  
 4 + 3 = 7

STEP 2: Initial program design

## Pseudocode for a stepwise design

Since we have a straightforward specification, we can now go immediately to program design, using stepwise refinement. Were the specification a little more complex, there would be a need to produce more details about the precise requirements of the program to be written.

In this chapter we will use a pseudocode notation for our design.

1. addition program

1.1 input ASCII codes for the two numbers

1.2 convert the ASCII codes to corresponding integers

1.3 do calculation

1.4 display result

1.5 stop program (**stop**)

1.1 input ASCII codes for the two numbers

1.1.1 input and store first ASCII code

1.1.2 input and store second ASCII code

1.2 convert the ASCII codes to corresponding integers

1.2.1 convert and store integer for first digit’s ASCII code

1.2.2 (as for 2.1, for second ASCII code)

1.3 do calculation

1.3.1 load first number’s integer into ACCUMULATOR

1.3.2 add in second number

1.3.3 store result

1.4 display result

1.4.1 display first number

1.4.2. display “ + “

1.4.3 display second number

1.4.4 display “ = “

1.4.5 display answer value

## Refining the input and output

There are two things about CECIL that are important for our program design:

1. input is all in the form of ASCII keycodes,
2. integers can be output simply, but text must be output as ASCII codes.

1.1.1 input and store first ASCII code

1.1.1.1 display “1st” on screen as a prompt for first number

1.1.1.2 put cursor on a new line

1.1.1.3 input first number’s ASCII code

1.1.1.4 store ASCII code

1.1.1.5 put cursor onto a new line (ready for “2nd” prompt)

1.1.2 input and store second ASCII code

(as for 1.1.1, but with “2nd” and storing second number’s ASCII code)

Once a high level design has been created, it is a good idea to begin to plan out the data that will be needed for the program. We will need memory stores for:

1. the two input numbers ASCII codes,
2. the two input numbers integer values, and
3. the integer result of the addition.

## ASCII codes need to be stored as data

We will also need the ASCII data for the characters to be output. The table below gives the code for each character, including the LINEFEED (LF) and SPACE characters, to allow us to present the output on different lines, as shown in the example of the input/output from the specification. Note, a full set of ASCII codes can be found in Appendix C.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | s | t | 2 | n | d | + | = | LF | space |
| 49 | 115 | 116 | 50 | 110 | 100 | 43 | 61 | 10 | 32 |

Our data section of the program could look something like the following:

.char1 word for ASCII char for first number

.char2 word for ASCII char for second number

.num1 word for integer for first number

.num2 word for integer for second number

.total word for integer of total of the two numbers

.asc\_1 word for ASCII code of “1”

.asc\_s word for ASCII code of “n”

.asc\_t word for ASCII code for “t”

.asc\_2 word for ASCII code for “2”

.asc\_n word for ASCII code for “n”

.asc\_d word for ASCII code for “d”

.asc\_plus word for ASCII code for “+”

.asc\_equal word for ASCII code for “=“

.asc\_newline word for ASCII code for <NEWLINE>

.asc\_space word for ASCII code for <SPACE>

STEP 3: Detailed program design

We can now refine further each of the steps in our program design, referring to the data labels identified above.

## Displaying characters in CECIL

To display characters in the “Output” window using CECIL, it is necessary to load into the ACCUMULATOR an ASCII code, and then use the **printch** instruction. For example to display the letter “A”, which has the ASCII code 65, we would have to have 65 stored at a known address in memory (for example, “**asc\_a**“), and use the following two instructions:

**load asc\_a**

**printch**

To put the cursor on a new line, the ASCII character LINEFEED (10) is displayed like any other character. To display a space, the ASCII character for a space (32) is displayed like any other character.

## Refining the prompts and digit input

Knowing how CECIL displays characters on screen, we can go about designing the text output parts of our program.

1.1.1.1 display “1st” on screen as a prompt for first number

1.1.1.1.1 load ASCII code for “1” into ACCUMULATOR

1.1.1.1.2 print out the “1” character

1.1.1.1.3 load ASCII code for “s” into ACCUMULATOR

1.1.1.1.4 print out the “s” character

1.1.1.1.5 load ASCII code for “t” into ACCUMULATOR

1.1.1.1.6 print out the “t” character

1.1.1.2 put cursor on a new line

1.1.1.2.1 load ASCII code for NEWLINE into ACCUMULATOR

1.1.1.2.2 print out NEWLINE character

1.1.1.3 input first number’s ASCII code

1.1.1.3.1 wait for a keypress, and put ASCII code into ACCUMULATOR

1.1.1.4 store ASCII code

1.1.1.4.1 store ASCII code in ACCUMULATOR in location “**.char1**“

## Displaying integers in CECIL

CECIL has a number of different instructions for displaying the contents of the ACCUMULATOR in the output window. We have met **printch**, which interprets the value in the ACCUMULATOR as an ASCII code, and outputs the corresponding character. The instruction **print** simply displays the number in the ACCUMULATOR as an integer in the output window.

For example, the program fragment below displays the number 27 in the output window.

load num1

print

;-- data --

.num1 insert 27

## Displaying the result of the calculation

To display the result of the calculation in the form “N1 + N2 = ANS”, we will need to use a combination of sending characters and integers to the output window.

1.4.1 display first number

1.4.1.1 load in first integer

1.4.1.2 display integer as decimal

1.4.2 display “ + “

1.4.2.1 display a <SPACE>

1.4.2.2 display “+” sign

1.4.2.3 display a <SPACE>

1.4.3 display second number

1.4.3.1 load second integer

1.4.3.2 print second integer

1.4.4 display “ = “

1.4.4.1 display a <SPACE>

1.4.4.2 display “=“ sign

1.4.4.3 display a <SPACE>

load .asc\_space

printch

1.4.5 display total of the integers

1.4.5.1 load total of the integers

1.4.5.2 print total

## Converting from ASCII codes to integers

There is a slight problem with our program, since we are inputting the two numbers as ASCII codes, but we wish to add their integer values together. Therefore we need to convert from the ASCII codes for the characters “0” .. “9”, to their corresponding integers.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| character | “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9” |
| ASCII  code | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 |

The table below shows each character, and its ASCII code:

A simple way to calculate the integer of a digit, given its ASCII code, is to subtract from the ASCII code the number 48 (i.e. the ASCII code for the digit “0”).

For example:

character “7” has the ASCII code 55, 55 - 48 = 7

One needs to make sure, of course, that the ASCII code is actually for a digit (from 48..57), i.e. that the user has actually pressed a digit key, rather than some other key on the keyboard.

## Refine steps for converting ASCII codes to integers

We need to store the ASCII code for zero, and then subtract this from the ASCII codes for the two digits entered by the user.

1.2.1 convert and store integer for first digit’s ASCII code

1.2.1.1 load ASCII code for first digit (“**char1**”) into ACCUMULATOR

1.2.1.2 subtract from the ACCUMULATOR the ASCII code for “0”

1.2.1.3 store the integer in the ACCUMULATOR to “**num1**”

1.2.2 convert and store integer for second digit’s ASCII code

as for 1.2.1, but using “**char2**” and “**num2**”

STEP 4: Coding of design

## The full version of the program

Note, although for illustration the stepwise refinement has been shown to a very fine granularity (i.e. down to steps where each step corresponds to a single CECIL instruction), it would make the program too verbose if every instruction had its own comment. Therefore, only the major steps numbers from the design have been included as comments.

program addup two numbers

author Matt Smith

date 6 / April / 1996

;-----------------------------------------

;-- 1 Addition program --

;------------------------------------------------------

;-- this program inputs two single digit --

;-- integers from the user, and displays their total --

;------------------------------------------------------

;---------------------------------------------

;-- 1.1 input first and second ASCII codes --

;---------------------------------------------

;-- 1.1.1 input first ASCII code --

;-----------------------------------

;-----------------------------------

;-- 1.1.1.1 display “1st” prompt --

;-----------------------------------

.start load asc\_1

printch

load asc\_s

printch

load asc\_t

printch

;----------------------

;-- 1.1.1.2 newline --

;----------------------

load asc\_newline

printch

;-----------------------------------------------

;-- 1.1.1.3 input ASCII code for first digit --

;-----------------------------------------------

getkey

;-----------------------------------------------

;-- 1.1.1.4 store ASCII code for first digit --

;-----------------------------------------------

store char1

;----------------------

;-- 1.1.1.5 newline --

;----------------------

load asc\_newline

printch

;-----------------------------------

;-- 1.1.2 input second ASCII code --

;-----------------------------------

;-----------------------------------

;-- 1.1.2.1 display “2nd” prompt --

;-----------------------------------

load asc\_2

printch

load asc\_n

printch

load asc\_d

printch

;----------------------

;-- 1.1.2.2 newline --

;----------------------

load asc\_newline

printch

;------------------------------------------------

;-- 1.1.2.3 input ASCII code for second digit --

;------------------------------------------------

getkey

;------------------------------------------------

;-- 1.1.2.4 store ASCII code for second digit --

;------------------------------------------------

store char2

;----------------------

;-- 1.1.2.5 newline --

;----------------------

load asc\_newline

printch

;--------------------------------------------

;-- 1.2 convert ASCII codes into integers --

;--------------------------------------------

;------------------------------------------------------

;-- 1.2.1 convert and store first digit’s ASCII code --

;------------------------------------------------------

;--------------------------------------------------

;-- 1.2.1.1 load ASCII code for “.char1” into ACC--

;--------------------------------------------------

load char1

;------------------------------------------

;-- 1.2.1.2 subtract ASCII code for “0” --

;------------------------------------------

cset

sub asc\_0

;-----------------------------------------------

;-- 1.2.1.3 store result at address labelled –-

;-- “.num1” --

;-----------------------------------------------

store num1

;-------------------------------------------------

;-- 1.2.2 convert second ASCII code to integer --

;-------------------------------------------------

;--------------------------------------------------

;-- 1.2.2.1 load ASCII code for “.char2” into ACC--

;----------------------------------------------------

load char2

;------------------------------------------

;-- 1.2.2.2 subtract ASCII code for “0” --

;------------------------------------------

cset

sub asc\_0

;-----------------------------------------------

;-- 1.2.2.3 store result at address labelled –-

;-- “.num2” --

;-----------------------------------------------

store num2

;---------------------------------------

;-- 1.3 add the two numbers together --

;---------------------------------------

;----------------------------------------

;-- 1.3.1 load first integer into ACC --

;----------------------------------------

load num1

;----------------------------------

;-- 1.3.2 add in second integer --

;----------------------------------

cclear

add num2

;---------------------------------------------

;-- 1.3.3 store result at address labelled --

;-- “.total” --

;---------------------------------------------

store total

;----------------------------------------------------

;--- 1.4 display result in form “N1 + N2 = TOTAL“ --

;----------------------------------------------------

;---------------------------------

;-- 1.4.1 display first number --

;---------------------------------

;-- load in first integer

load num1

;-- display integer as decimal

print

;--------------------------

;-- 1.4.2 display “ + “ --

;--------------------------

;-- display a <SPACE>

load asc\_space

printch

;-- display “+” sign

load asc\_plus

printch

;-- display a <SPACE>

load asc\_space

printch

;---------------------------------

;-- 1.4.3 display second number --

;---------------------------------

;-- load second integer

load num2

;-- print second number

print

;--------------------------

;-- 1.4.4 display “ = “ --

;--------------------------

;-- display a <SPACE>

load asc\_space

printch

;-- display “=“ sign

load asc\_equal

printch

;-- display a <SPACE>

load asc\_space

printch

;------------------------------------------

;-- 1.4.5 display total of the integers --

;------------------------------------------

;-- load total of the integers

load total

;-- display total

print

;-------------------------

;-- 1.5 stop program ! --

;-------------------------

stop

;---------------------------------

;-- data --

;---------------------------------

;-- ASCII code of first digit

;-- (the zero is a dummy, to allow labelling of an address)

.char1 insert 0

;-- ASCII code of second digit

.char2 insert 0

;-- integer for first number

.num1 insert 0

;-- integer for second number

.num2 insert 0

;-- integer for total of the two numbers

.total insert 0

;-- ASCII codes for chars to be displayed on screen

.asc\_1 insert 49

.asc\_n insert 115

.asc\_t insert 116

.asc\_2 insert 50

.asc\_n insert 110

.asc\_d insert 100

.asc\_plus insert 43

.asc\_equal insert 61

.asc\_newline insert 10

.asc\_space insert 32

.asc\_0 insert 48

STEP 5: Refinement of program design

## Refinement 1: Breaking up the program using subroutines

It is often a good idea to break up a program into subroutines. This has a number of advantages, including allowing easy “copy-and-paste” reuse of subroutines for new programs, and also it allows the development of parts of the program in isolation from the rest of the program.

There are two ways to make the program counter jump from one location to another:

1. unconditional jump,
2. conditional jump.

At the moment we only need to use unconditional jumps, i.e. instructions the **always** cause the program counter to be changed (i.e. to make the flow of control of the program jump from instructions at one place in memory to another).

There are two instructions for unconditional jumps: **jump** and **jmptosr**. Wherever possible, one should use **jmptosr**, since when used carefully act very much like procedure or function calls in higher level languages, such as LOGO, Pascal and C.

To convert a program into subroutines means adding:

1. **jmptosr** calls,
2. address labels next to the first instruction in each subroutine called, and
3. **return** instructions at the end of each subroutine   
   (to return control back to the address of the last **jmptosr** instruction).

The use of subroutines allows an approach to the program implementation to match more closely the program design. The beginning of the program could be re-written as follows:

program add up two numbers

author Matt Smith

date 6 / April / 1996

;-----------------------------------------

;-- 1 Addition program --

;-----------------------------------------

;-- this program inputs two single digit

;-- integers from the user, and displays their total

;---------------------------------------------------

.start jmptosr input\_digits

jmptosr convert\_to\_integers

jmptosr do\_calculation

jmptosr output\_results

stop

The beginning and ending of each subroutine could be re-written as follows:

;---------------------------------------------

;-- 1.1 input first and second ASCII codes --

;---------------------------------------------

;-- 1.1.1 input first ASCII code --

;-----------------------------------

.input\_digits load asc\_1

printch

...

;----------------------

;-- 1.1.2.5 newline --

;----------------------

load newline

printch

return

;--------------------------------------------

;-- 1.2 convert ASCII codes into integers --

;--------------------------------------------

;------------------------------------------------------

;-- 1.2.1 convert and store first digit’s ASCII code --

;------------------------------------------------------

;--------------------------------------------------

;-- 1.2.1.1 load ASCII code for “.char1” into ACC -

;--------------------------------------------------

.convert\_to\_integers

load char1

...

;--------------------------------------------------

;--1.2.2.3 store result at address labelled *num2* --

;--------------------------------------------------

store num2

return

;---------------------------------------

;-- 1.3 add the two numbers together --

;---------------------------------------

;----------------------------------------

;-- 1.3.1 load first integer into ACC --

;----------------------------------------

.do\_calculation

load num1

...

;--------------------------------------------------

;-- 1.3.3 store result at address labelled *total* --

;--------------------------------------------------

store total

return

;--------------------------------------------------

;--- 1.4 display result in form “N1 + N2 = TOTAL“ --

;--------------------------------------------------

;---------------------------------

;-- 1.4.1 display first number --

;---------------------------------

;-- load in first integer

.output\_results

load num1

...

;------------------------------------------

;-- 1.4.5 display total of the integers --

;------------------------------------------

;-- load total of the integers

load total

;-- display total

print

return

This refinement could be taken further, by having some of the subroutines above broken down into subroutines themselves, for example, step 1.2 (convert ASCII to integers) could be broken down as follows:

;--------------------------------------------

;-- 1.2 convert ASCII codes into integers --

;--------------------------------------------

.convert\_to\_integers

jmptosr asc1\_to\_integer

jmptosr asc2\_to\_integer

return

;------------------------------------------------------

;-- 1.2.1 convert and store first digit’s ASCII code --

;------------------------------------------------------

;------------------------------------------------

;-- 1.2.1.1 load ASCII code for *char1* into ACC --

;------------------------------------------------

.asc1\_to\_integer

load char1

;------------------------------------------

;-- 1.2.1.2 subtract ASCII code for “0” --

;------------------------------------------

cset

sub asc\_0

;--------------------------------------------------

;-- 1.2.1.3 store result at address labelled *num1*--

;--------------------------------------------------

store num1

return

(same for “.asc2\_to\_integer”)

## Refinement 2: Re-using a subroutine

We can make the subroutine for converting the ASCII code of a digit to its corresponding integer (step 1.2) by making two assumptions:

1. assumption 1: the ASCII code will be in the ACCUMULATOR before the subroutine is called,
2. assumption 2: the corresponding integer for the ASCII code will be left in the ACCUMULATOR after the subroutine has finished.

It is a convention in computing to call these sort of assumptions the “***interface***” to the subroutine. I.e. they specify how things are passed as ***input*** and ***output*** to the subroutine, but do not restrict the “***implementation***” of how the subroutine is to do the action.

It is important to carefully specify the interface for assembly language subroutines, because low level languages to not provide the facility of parameters (also called arguments) for the passing of values to and from subroutines (higher level languages such as LOGO, Pascal and C all allow parameters to be passed between subroutines).

As can be seen in the code below, the comments clearly specify the interface for the subroutine in terms of where data is stored BEFORE and AFTER the subroutine has been called.

;-------------------------------

;-- subroutine .asc\_to\_integer --

;-------------------------------

;-- converts ASCII code to corresponding integer

;--

;-- interface

;-- BEFORE: ASCII code in ACCUMULATOR

;-- AFTER: corresponding integer in ACCUMULATOR

;-------------------------------------------------

.asc\_to\_integer cset

sub asc\_0

return

;-- data --

.asc\_0 insert 48

Since we have written a general purpose subroutine, we can remove the duplicated code, and re-write the two subroutines **asc1\_to\_integer** and **asc2\_to\_integer** to place data correctly before calling the subroutine **asc\_to\_integer**.

;------------------------------------------------

;-- 1.2.1 converts ASCII code of first char to --

;-- corresponding integer --

;------------------------------------------------

;-- load ASCII code into ACCUMULATOR

.asc1\_to\_integer load char1

;-- call subroutine

jmptosr asc\_to\_integer

;-- store result at “.num1”

store num1

return

same for subroutine “.asc2\_to\_integer” (using “.char2” and “.num2”)

## Refinement 3a: Declaring simple data

All data is stored in memory as 10-bit words using the “**insert**” data declaration instruction, which is followed by the decimal integer representing the 10-bit number to be stored.

This is fine for storing integers, but a little opaque when the numbers actually represent the ASCII codes of characters to be displayed.

code using **insert**

;-- data --

;-- ASCII code for upper case A

.asc\_a insert 65

## How we can use “insert” in our addition program

A small change, but making the program listing a little clearer for humans to read, is to use “**insert**” data declaration instructions in the data section of our program.

For example:

;-- ASCII codes for chars to be displayed on screen

.asc\_1 insert “1”

.asc\_s insert “s”

.asc\_t insert “t”

.asc\_2 insert “2”

.asc\_n insert “n”

.asc\_d insert “d”

.asc\_plus insert “+”

.asc\_equal insert “=“

.asc\_newline insert 10 ;-- no char, must state ASCII code

.asc\_space insert “ “

.asc\_0 insert “0“

## Refinement 3b: Printing data in a loop

The “**insert**” data declaration instruction only declares one word at a time.

code using **insert**

;-- data --

;-- lower case “hello”

.text insert 104

insert 101

insert 108

insert 108

insert 111

## Using the X-register to print strings in a loop

A more radical refinement is the use of a loop to display a string of characters. This is a little complex, so we will develop the loop with a small, simple program, then extend that program into a re-usable subroutine, which we will then fit into our addition program.

Consider the simple program below to display the word “yes”.

.start load asc\_y

printch

load asc\_e

printch

load asc\_s

printch

stop

;-- data --

.asc\_y insert “y”

.asc\_e insert “e”

.asc\_s insert “s”

We can see that there are two actions repeated for each letter in the character string:

load asc\_<CHAR>

printch

We can also see that each time the actions are repeated, the ASCII code stored at the **previous address + 1** is retrieved.

We will have to know two things: (1) the address of the letter to read; (2) the ASCII code of the letter. Convention says to store data about the addresses in the X-register, and the ASCII codes in the ACCUMULATOR (since **printch** refers to the value in the ACCUMULATOR).

Before designing our program, we need to understand which CECIL instructions can be used for a string printing loop.

## A load modified by the value in the X-register

To load the first letter of the string “Yes”, we can use the statement:

load asc\_y

To load the second letter, we could use the statement:

load asc\_e

However, for this we have had to state explicitly the address of each letter individually. We wish to be able to say something like “***load the letter following ‘Y’*** ”. This is found at the address of “**asc\_y**” + 1.

CECIL provides another “**load**” instruction, called “**loadmx**”. This instruction loads a word into the ACCUMULATOR from an address related to the given label, but modified by the value in the X-register. What it actually does is to take the address of the given label, and add to that address the current value of the X-register, and then load the word at this modified address into the ACCUMULATOR.

We can place and change values in the X-register using the “**xload**” , “**xinc**” and “**xdec**” instructions. These instructions do the following:

**xload** load a value from a labelled address into the X-register,

**xinc** increments the X-register (adds 1 to the old value), and

**xdec** decrements the X-register (subtracts 1 to the old value).

For example:

xload zero ;put zero into X-register

loadmx yes ;load word at address (.yes + 0) into ACC

;(i.e. load ASCII for “Y” into ACC)

printch ;display “Y”

xinc ;add 1 to X-register (now holds 1)

loadmx yes ;load word at address (.yes + 1) into ACC

;(i.e. load ASCII for “e” into ACC)

printch

xinc ;add 1 to X-register (now holds 2)

loadmx yes ;load word at address (.yes + 2) into ACC

;(i.e. load ASCII for “s” into ACC)

printch

stop

;-- data --

.zero insert 0

.yes insert 89 ;ASCII for “Y”

insert 101 ;ASCII for “e”

insert 115 ;ASCII for “s”

This data section could be re-written:

;-- data --

.zero insert 0

.yes insert “Yes”

So far it looks as though this is a lot more work than is worth it. However, we have achieved something, because we now have a set of identical statements being repeated, i.e.:

loadmx yes ;load into ACC word

;at address = (yes + X-reg)

printch

xinc ;add 1 to X-register

## A sentinel value to stop looping

Since we now have identical statements being repeated, we can write them once, label the address of the first repeated instruction, and use a “**jump**” instruction to create a loop.

The code would look as follows:

xload zero ;put zero into X-register

.loop loadmx yes ;load next character

printch ;printch out current character

xinc ;increment X-register

jump loop ;set PC back to .loop

The only problem is that we have now written an infinite loop !

There are two ways to usually used to stop a loop:

1. loop a certain number of times (by keeping a count, and stopping when count matches desired number of repetitions)
2. loop until a termination condition is met.

Although we know there are only 3 letters in the word “Yes”, if a loop were written that would work for a string of any length, then such a loop would be much more reusable. To work with strings of any length, a common programming practice is the use of a sentinel value, which is places after the last valid piece of data, and the loading of the sentinel value is coded as the termination condition of the loop.

Since there is no printable character with an ASCII code of zero, zero is commonly used as a sentinel value when working with ASCII data. Our data section of the program now looks as follows:

;-- data --

.zero insert 0

.yes insert “Yes”

insert 0 ;sentinel value (no need for a label)

## Designing a loop using the X-register and a sentinel value

Using pseudocode we can design a general purpose algorithm to print out strings of any length as follows. Note that this CECIL form of pseudocode introduces the use of labels for step numbers.

1.1 load zero into the X-register (**xload zero**)

1.2 load first character into ACC (**loadmx <label>**)

.loop 1.3\* LOOP

1.3 display character and load next character

.exit 1.4 end program (**stop**)

1.3 display character and load next character

1.3.1 display current character (**printch**)

1.3.2 add 1 to current address (**xinc**)

1.3.3 load character at current (modified) address into ACC (**loadmx <label>**)

.loop 1.3\* LOOP

.loop 1.3\*.1 LOOP again unless sentinel character in ACC

.loop 1.3\*.1 LOOP again unless sentinel character in ACC

1.3\*.1.1 compare current character with a sentinel value

1.3\*.1.2 if current character is sentinel value, then exit loop

1.3\*.1.3 if not sentinel value, jump back to step 1.3 (**jump loop**)

The design above has been written bearing in mind that there are no loop instructions available in CECIL, so a loop has to be built out of comparison instructions, and conditional jump instructions. Final refinement of steps 1.3\*.1.1 to 1.3\*.1.2 only require the use of some additional CECIL instructions. The “comp” instruction will set the zero flag, if the value at the labelled address is the same as that in the ACCUMULATOR. The “jizero” instruction causes the program counter to jump to the labelled address, if the zero flag is set (i.e. 1). These steps with their code look as follows:

1.3\*.1.1 compare current character with a sentinel value

**comp zero**

1.3\*.1.2 if current character is sentinel value, then exit loop

**jizero exit**

## The complete, compiled program

The CECIL code, plus the result of compilation looks as follows:

address label instruction data address value

0000 .start xload 0000 15

0001 zero 0001 0015

0002 load 0002 1

0003 yes 0003 0016

0004 .loop printch 0004 22

0005 xinc 0005 28

0006 loadmx 0006 17

0007 yes 0007 0016

0008 comp 0008 9

0009 zero 0009 0015

0010 jizero 0010 12

0011 exit 0011 0014

0012 jump 0012 8

0013 loop 0013 0004

0014 .exit stop 0014 38

0015 .zero 0 zero 0015 0

0016 .yes insert “Yes” 0016 121

0017 0017 101

0018 0018 115

0019 insert 0 0019 0

## A dry run of the program

The following is a complete dry run of the program to display our “Yes” string. Each description, and value given for the ACCUMULATOR and X-register is the situation after the statement for the given program counter has been executed (e.g. after the “**0000 load .zero**“ statement, the value of the ACCUMULATOR is unknown, and the X-register is zero).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| program counter | statement | description | ACC | X-register |
| 0000 (.start) | **xload zero** | load 0 into X-register | unknown | 0 |
| 0002 | **load yes** | load value from 0016 into ACC | 121 | 0 |
| 0004 | **printch** | output character corresponding to 121 (i.e. output “Y”) | 121 | 0 |
| 0005 | **xinc** | add 1 to X-register | 121 | 1 |
| 0006 | **loadmx zero** | load value from 0016 + 1 = 0017 into ACC | 101 | 1 |
| 1008 | **comp zero** | compare ACC with contents of 0015 (101 and 0 are different, so zero flag is zero) | 101 | 1 |
| 1010 | **jizero exit** | no action, since zero flag not set | 101 | 1 |
| 1012 | **jump loop** | set program counter to 0004 | 101 | 1 |
| 0004 (.loop) | **printch** | output character corresponding to 10 (i.e. output “e”) | 101 | 1 |
| 0005 | **xinc** | add 1 to X-register | 101 | 2 |
| 0006 | **loadmx zero** | load value from 0016 + 2 = 0018 into ACC | 115 | 2 |
| 1008 | **comp zero** | compare ACC with contents of 0015 (115 and 0 are different, so zero flag is zero) | 115 | 2 |
| 1010 | **jizero exit** | no action, since zero flag not set | 115 | 2 |
| 1012 | **jump loop** | set program counter to 0004 | 115 | 2 |
| 0004 (.loop) | **printch** | output character corresponding to 10 (i.e. output “s”) | 115 | 2 |
| 0005 | **xinc** | add 1 to X-register | 115 | 3 |
| 0006 | **loadmx zero** | load value from 0016 + 3 = 0019 into ACC | 0 | 3 |
| 1008 | **comp zero** | compare ACC with contents of 0015 (0 and 0 are the same, so zero flag is set to 1) | 0 | 3 |
| 1010 | **jizero exit** | since zero flag is 1, the program counter is set to 0014 | 0 | 3 |
| 1014 (.exit) | **stop** | program terminates | 0 | 3 |

## Printing character strings in the integer addition program

We can use the above to create two subroutines for the printing of the “1st” and “2nd” strings as follows:

.input\_digits jmptosr display\_first

jmptosr input\_num1

jmptosr display\_second

jmptosr input\_num2

;-- 1.1.2.5 newline --

load newline

printch

return

;-- subroutine to display “1st” --

.display\_first xload zero

load first

.loop\_first printch

xinc

loadmx first

comp zero

jizero exit\_first

jump loop\_first

.exit\_first load .space

printch

return

(same for “.display\_second”)

;-- data --

.first insert “1st”

insert 0

.second insert “second”

.space insert “ ”

We now have loops to display the strings of “1st” and “2nd” on screen.

It would be nice to write a re-usable string printing subroutine though ...

# Appendices

## 

## Appendix Page

## Introductory Tutorial: PC Windows 81

## Debugging Hints 90

## Machine codes 91

## The LEDs 92

## Language Reference 93

## ASCII character set 107

## Register and Flag Virtual Addresses 108

## Input and Output on the SIM20 109

# Appendix A Introductory Tutorial for CECIL PC Windows Version

## Introduction

What follows is an introductory tutorial for the PC Windows Version of the CECIL compiler. CECIL is not designed to stand alone; it is part of a complete teaching package. You will probably be reading this as part of the extensive documentation that has been produced. In case you are not, this introduction gives a brief explanation of the CECIL concept. Following this, there will be some notes on how to download and install CECIL on your own computer, then a walk-through of the CECIL package showing how to undertake basic operations, such as writing, editing and saving CECIL programs. Finally, there will be a few notes on getting further help.

CECIL is a simple assembly language for an imaginary microcontroller called a SIM20. It is designed to illustrate all the principles of machine code programming, whilst avoiding some of the more arcane and unnecessary intricacies that so often confuse the newcomer. In dealing with CECIL programs, you will find that you are learning about the design of the SIM20 itself, and this is also part of our intention in producing it.

In the world of computing, machines are often simulated before any electronic engineer gets anywhere near to assembling the first chip on the motherboard. A program that allows you to compile machine code for a computer that is not the same as the one you are running it on is called a *cross compiler*. You are about to use a cross compiler for CECIL; your PC will pretend to be a SIM20 for a while so that you can try out your CECIL programs.

## A Note on Versions

You need to be careful about using CECIL. Just as with any other language, there is a standard for the CECIL language, which is defined elsewhere. But different versions of the CECIL compiler exist, and some differences exist between them.

The most important thing to remember for the moment is that the PC Windows version **is** case sensitive, so *Data1* will be understood to be different from *data1* if used as a label, for example. In addition, commands **must** be in lower case, otherwise they will not be understood. **DON’T FORGET!!**

You will also be expected to use your common sense in the following notes. They are designed to help your run CECIL on **your** computer, and I don’t know how your computer is set up! These notes will help you with CECIL, not sorting out how your computer does things.

## Installing CECIL

There are five steps to installing Cecil. These are as follows.

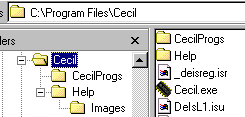
1. Get hold of the PC Windows version of the Cecil installation file. It is distributed as a single zip file (this makes it easier to email, for example), and will have a name like "Cecilxxxx.exe" where the xxxx stands for the version number. At the time of writing these notes, the current version is 0.951beta, so the distribution file is called "Cecil0951.exe". If you need to get hold of this file, go to http://www.soton.ac.uk/~da/ and look for it from there.
2. You now need to "unzip" the file. In order to do this, you will need a special program; "Winzip" is a good one, and you can download it free from the Internet if you don't have a copy. It is well worth while having this program installed on your computer in any case. Winzip can be downloaded from "http://www.winzip.com". Having downloaded it, follow the instructions to get it installed on your computer.
3. Once Winzip (or any other similar program) is installed on your computer, Cecilxxxx.zip should be recognised as a "zipped" file. In that case, double-click it, and the unzipping program should be launched with the Cecil zip file. You now have a choice, either to unzip Cecil onto your hard disc or onto a floppy disc. Let's assume you want to unzip Cecil onto your hard disc. It's probably easiest to create a new, clean folder for the unzipped files, then you can delete them all again to save a bit of space after you've installed Cecil.
4. Now navigate to the unzipped Cecil folder, and run "Setup.exe". The "exe" extension may not be visible if you have file extensions hidden, but the correct file will have the usual grey & blue computer icon that comes with setup files.
5. Follow the instructions that come up during the install procedure.  
     
   You may choose where to install Cecil, for example onto your hard disc, or you could choose to install to a floppy so that you have a working floppy disc to take around with you. Cecil is small enough to fit onto a floppy and still leave space for a fair number of programs that you will want to write. However, note that if you install to a floppy disc, an icon will be placed on your start menu that points to your floppy disc. It would therefore be better to install to floppy disc *first* if you want to do this, and then to install to hard disc afterwards, since this will leave all the icons pointing to the right places.

You should now have Cecil installed so that you can run it on your hard disc by picking up the icon from Start Menu -> Programs -> Cecil or from your floppy disc by double-clicking the Cecil.exe icon.

## Sorting out Problems

The Installation procedure is supposed to be designed so that you don't have to worry about where files and directories go, or how to put things onto the Start Menu, or how to use icons, for example. Reality is usually different, so this section is designed to help you get things going if things have gone wrong.

The Cecil installation procedure should create the following file structure in your chosen directory. If you don't change it, the default installation directory is "C:\Program Files\Cecil".



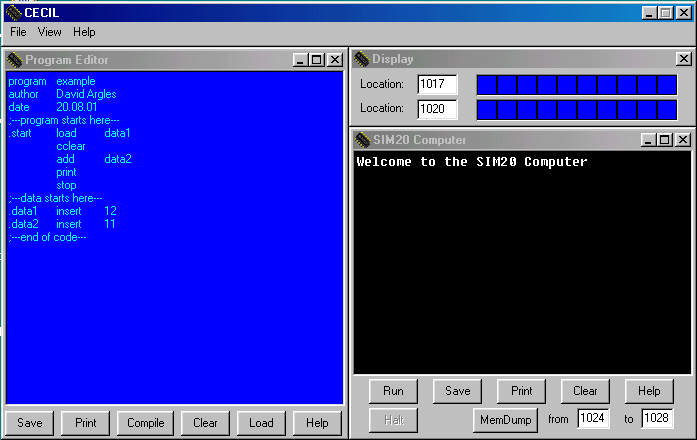
You will see that the expected layout is that your Cecil directory should include the Cecil program (Cecil.exe), some isr and isu files (I don't know what they do; if you know or find out, please tell me. Perhaps they're important...) and two directories. The first directory is called "CecilProgs". This is designed to take any programs you write in Cecil, and contains a few examples that I've written. I take no responsibility for these examples; I wrote them for me to test things out while writing the program and whether they work or are written properly is for you to decide!

The second directory is called "Help"; this contains all the files needed to run the help windows in Cecil. Inside the Help directory is another directory called Images. If you know a bit about customising your PC, this might be useful for you to know about, since it contains some icons that you might want to use to personalise your installation, for example. If you don't know how to do this, I don't intend to tell you here.

The crucial issue is that, if anything goes wrong during installation and Cecil can't find its help files, it will ask you to find them when you launch the program. The first file it looks for is CecilPNotes.rtf; try and track it down, point Cecil at it, and all will run happily. If you can't find it, don't worry, Cecil will still run, although it won't have any help files available.

## Running CECIL

Once Cecil is installed, running it is easy. Either go to Start Menu -> Programs -> Cecil or, if you're running it from your floppy disc, double-click the Cecil.exe icon. When Cecil launches, you will get the "Splash Screen" for three seconds, and then it will change to the Cecil environment.



There are four main windows that should appear at start-up which should look as follows.

You should find that the big window on the left, which is titled "Program Editor" and has a blue background, has a program in it entitled "example". This is where you will write your programs in Cecil for the SIM20 computer.

The big window on the right, which is titled "SIM20 Computer" and which has a black background, is where you will see any output from the SIM20 computer. At start-up, all it says is "Welcome to the SIM20 Computer." You will need to study this window carefully when you start to compile and run your programs.

Don't be deceived, there is a window which is actually more important than these two. The long narrow one at the top, which is just labelled "CECIL," is actually the one that controls the whole simulation. If you close this one down, the whole program will close down. If you close down the Program Editor or the Output Window, you can make them reappear by going to the CECIL window and using the drop-down menu to select View -> Editor or View -> Output as appropriate. You can also use options from this drop-down menu to tidy up your windows, to centre them on the screen, or to bring up the help files, for example.

The fourth window that appears is the "Display" window, and gives a simulated LED display of two chosen memory locations. We'll worry about those later.

*It might be worth taking a break now to launch Cecil on your computer, and to fiddle around with the windows and menus just to get used to what everything does.*

## Running a Program

Since Cecil has launched with a program already written in the Program Editor, we can go ahead and run it straight off. We'll worry about how programs are written later.

There are two stages to running a program on a computer. At the moment, the SIM20 has an empty memory, so we need to "compile" our program to convert the text that we read in the Program Editor into "machine code" that the SIM20 can understand.

Lets' pretend that you don't believe me. We can see what is in the SIM20's memory by using the "MemDump" (Memory Dump) button and the bottom right of the SIM20 window. First change the numbers that are in the little windows alongside the button so that we will go from location 0 to location 10:



Now press the MemDump button. This should give you a print-out as follows:

Doing memory dump:

loc: 0 val: 0

loc: 1 val: 0

loc: 2 val: 0

loc: 3 val: 0

loc: 4 val: 0

loc: 5 val: 0

loc: 6 val: 0

loc: 7 val: 0

loc: 8 val: 0

loc: 9 val: 0

loc: 10 val: 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

What this has done is printed out a list of the contents of the first 11 memory locations - note that, being a computer, it numbers its memory locations from 0, not 1! You will also see that all the locations ("loc") contain 0 ("val").

Now go to the Program Editor window. At the bottom of this window is a button called "Compile."



Click this button, and you will see a large amount of text appear in the SIM20 window. For now, we don't need to worry too much about all this text; the important thing is that the last few lines should simply say "Program compiled OK." What has happened is that the program text has been taken from the Program Editor window and converted into machine code in the SIM20's memory ("compiled").

Now if you go back to the MemDump button and do another memory dump of the SIM20's memory, you will find that all sorts of funny numbers should have appeared in the first 9 locations of the SIM20's memory. It should look something like this:

Doing memory dump:

loc: 0 val: 1

loc: 1 val: 7

loc: 2 val: 33

loc: 3 val: 3

loc: 4 val: 8

loc: 5 val: 38

loc: 6 val: 0

loc: 7 val: 12

loc: 8 val: 11

loc: 9 val: 0

loc: 10 val: 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We don't need to be concerned about what these numbers mean just yet - we'll come back to that. The important thing is that the SIM20 now has a program stored in its memory which it can run.

Let's run it now. There is a button at the bottom of the SIM20 window called "Run."



Click this, and you should see the following text appear in the SIM20 window:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Executing program:

example

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

23

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Program run completed

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This is simply telling us that it knows it has a program stored in its memory called "example", that it began by running it, that it produced some output, "23", and that the program run completed satisfactorily.

Whilst the program listing (i.e. the program written in the Program Editor window on the left) might still be fairly meaningless at this stage, you might be able to pick out the two lines near the end labelled "data1 and data2 which have the numbers 12 and 11 on. Also, there's a line further up which has the command "add" on. What this program has done is to add the two numbers together from the data1 and data2 line and to print out the answer. If you don't believe me, try changing the numbers, recompile the program by clicking the "Compile" button again, so that the SIM20 has its stored program changed as well, and then click the "Run" button. While you're doing this, you might like to find out what happens if either number gets too big, or if the answer gets too big. How much is too big, anyway?

## Writing a Program

Let's now get down to writing our own program. The first thing to do is to clear the old program out of the program editor. Click the "Clear" button at the bottom of the Program Editor.



You will almost certainly see a window pop up that says your old program is not saved. Just click "Yes" to confirm that you want to go ahead and clear it, and you will be left with a blank editor.

You are now able to type in your program. Use the tab key as a separator and use it just once each time. Here’s the program:

program Load & print a number  
author David Argles  
date 31.01.97  
;  
;-- program starts here --  
;  
.start load number  
 print  
 stop  
;  
;-- now the data --  
;  
.num insert 4  
;  
;-- end of source code --

Two things to do now before doing anything else. First of all, **!!!SAVE YOUR PROGRAM!!!** To do this, click the "Save" button at the bottom of the Editor Window.

This will put up a conventional Windows save dialogue; decide where you want to save your work, give your program a sensible name, and then click "Save."



The second thing to do is to check what you've written. You'd be amazed how many problems you can save yourself just by checking what you've done carefully.

When you're happy with everything, click on the "Compile" button as you did with the previous program. Text will scroll through the SIM20 Window, but at the bottom, you will see the dreaded words, "Program failed to compile." Don't panic - it contains a deliberate error...

The first thing to do when your program doesn't compile is to look at the output from the SIM20 and to see what it's telling you. It's usually very helpful, it just needs a bit of work to understand what it's trying to tell you. The output looks like this.

|  |  |
| --- | --- |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Starting compiler...  Program name: Load & print a number  Author: David Argles  Date: 31.01.97  Today's Date: 18.12.2001 at 13.13hr  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Label: start Address: 0  !! Label not found: number !!  Label: num Address: 4  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Some labels not found  Checking program second time  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Program name: Load & print a number  Author: David Argles  Date: 31.01.97  Today's Date: 18.12.2001 at 13.13hr  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  !! Label not found: number !!  !! Some labels still not found !!  Doing memory dump:  loc: 0 val: 1  loc: 1 val: 65535  loc: 2 val: 38  loc: 3 val: 0  loc: 4 val: 4  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Starting location is 0  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  !! Program failed to compile !!  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* | The first part tells us that the compiler is starting up and that it has found the three necessary lines which begin "program", "author" and "date".  It then runs through checking that the program is written correctly and that it can find the various "labels" that are referred to by the program. In this case, it can't find a label called "number, so it looks through a second time.  Same as before, just going through a second time.  Still can't find anywhere called "number", so it gives up.  First a memory dump, so you can see what you have got. This is a very simple program which uses just 5 locations.  The program starts at location 0...  ...and it didn't compile! |

If you now look back at the original program, you will find that the sixth line says:

.start load number

However, the data line (line 12) that goes with it says:

.num insert 4

We've told it to load "number", but given it "num"! All we need to do is to edit our program so that we use the same word in both places. Let's edit line 12 so that it says:

.number insert 4

Having made our change, it's worth getting into the habit of saving our program again. Click the "Save" button and save your program again, so you've saved the update.

## Running

You are now ready to have another go at seeing if your program will work. You need to re-compile it; the machine code that was produced by last time’s unsuccessful compilation won't work. Click the "Compile" button. If you've done it right, you should see the words, "Program compiled OK." If not, just work it back through and find your mistake.

Your program should now compile successfully, producing the following output:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Starting compiler...

Program name: Load & print a number

Author: David Argles

Date: 31.01.97

Today's Date: 18.12.2001 at 13.40hr

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Label: start Address: 0

!! Label not found: number !!

Label: number Address: 4

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Some labels not found

Checking program second time

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Program name: Load & print a number

Author: David Argles

Date: 31.01.97

Today's Date: 18.12.2001 at 13.40hr

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Doing memory dump:

loc: 0 val: 1

loc: 1 val: 4

loc: 2 val: 38

loc: 3 val: 0

loc: 4 val: 4

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Starting location is 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Program compiled OK

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

If it doesn’t, track down your errors, correct them, and get your program going. Something to note; the compiler still couldn't find the "number" label first time through. That's because it goes through in order, and doesn't see where the "number" label is defined until it gets to the end of the program. It then goes through a second time, and this time it knows where "number" will be stored. This is common practice in compilers; they work through your program in two passes in order to work out where everything goes in memory.

You can now click the "Run" button. This should give the following output.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Executing program:

Load & print a number

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

4

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Program run completed

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This is simply telling you that it is running your program, "Load & print a number", that the output is 4, and that the program run has concluded correctly.

## What else do I need to know?

***Memory Dumps***

We've already seen that you get a memory dump of your program from the SIM20's memory when you compile your program, and that you can also get a memory dump by using the "MemDump" button at the bottom of the SIM20 window.

You can also use memdump to examine the SIM20's various flags and registers. These are presented as if they were extra memory locations above the real top, and numbered from 1024 to 1028. The following is what I got when I clicked "MemDump" for locations **1024 1028**:

Doing memory dump:

Displaying register contents...

PgCtr: 4

status: 0

flags-> C: 0 N: 0 Z: 0

areg: 4

xreg: 0

yreg: 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The meaning and use of these various registers is something for another session; for now, just notice that there are 5 registers, PgCtr, status, and the a, x and y registers which each hold a value. The status register contains flags which are printed out separately (I find this easier to follow - it saves having to understand binary arithmetic...). These are labelled C, N & Z for the Carry, Negative and Zero flags.

***Printing***

You’re going to need to have print-outs of your various bits of output. There are two ways to do this. Firstly, tidy up the output window, and then just click the "Print" button. If you prefer, you can drag across the text you want, use <ctrl-C> to copy it, and then paste it into your favourite editor and print it from there, or you could save your output as a file somewhere, and then load it into your favourite editor and print that way. In the same way, you can print your program file from the program editor.

***Help***

You may well find that you need help of various sorts whilst running your CECIL programs. The best way of getting into this is to go to the main CECIL window and select Help -> Introduction and then navigating to "The Interface" and selecting the Help tab. This will explain the various help windows that exist and how to use them.

## Quitting

Sooner or later, you’ll want to finish using CECIL, difficult though that might be to imagine. Simply close the main CECIL window either by clicking on the cross at the top right of the window, or select File -> Exit from the drop-down menu. If you have an unsaved program, you will be prompted to save it - but don’t rely on this. Always **!!!SAVE YOUR PROGRAM!!!**

## Getting CECIL From the Web

From here on you’re on your own - apart from the rest of the full manual, that is! But you may want to know about how to get hold of a copy of CECIL. It’s kept on our Blackboard site. Begin at:

**http://www.blackboard.soton.ac.uk/**

Now pick up the links to **WR101**. Follow on through to the Resources section and you'll find the CECIL distribution file. Click on it, and follow any instructions.

# Appendix B: Debugging Hints

Suggestion 1: Check the two most common mistakes — are all “add” instructions preceded by a “cclear”, and are all “sub” instructions preceded by “cset” ?

Suggestion 2: The third most common mistake is a comment inserted between an address label, and an instruction (or data declaration instruction). CECIL will not compile programs with anything between a label and the instruction being labelled (i.e. make sure comments are either on a line to themselves, or only added to the **right** of a completed instruction or data declaration line).

Suggestion 3: Use comments well.   
Have a comment after the three line program header, which summarises what the whole program does in a few sentences. Make sure that ALL data either appears BEFORE the program, or AFTER the program, and a comment clearly labels the boundary between the two.

Suggestion 4: When a program is misbehaving, add extra “stop” instructions, preceded by a “print”. This means that the program will display what is in the ACCUMULATOR and then stop.

Then do a memory dump (“Edit.Memory Dump” choice), to see the values of the three flags (and confirm the address of the program counter.

Suggestion 5: Memory dumps can also be used to display what is in the keyboard input buffer, and the contents and pointer to the top of the stack.

Suggestion 6: If a subroutine works in a program by itself, but not as part of your main program, check that it is not trying to use the same data label as some other part of the program.

If it doesn’t work outside the program, then fix it before adding it to your main program !

Suggestion 7: Develop you program in modules. If complicated ones, develop and test in a separate program, then paste in. Avoid problems with duplicate labels by keeping to some convention, and making a separate master list of program labels on a piece of paper as you design and program.

# Appendix C: Machine Codes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | |  |  | |
|  | |  |  | |
| **Machine Code Order** | |  | **Alphabetical Order** | |
| 0 | stop |  | 3 | add |
| 1 | load |  | 5 | and |
| 2 | store |  | 33 | cclear |
| 3 | add |  | 9 | comp |
| 4 | sub |  | 32 | cset |
| 5 | and |  | 7 | eor |
| 6 | or |  | 34 | getkey |
| 7 | eor |  | 49 | intdisable |
| 8 | jump |  | 48 | intenable |
| 9 | comp |  | 15 | jicarry |
| 10 | jineg |  | 10 | jineg |
| 11 | jipos |  | 11 | jipos |
| 12 | jizero |  | 12 | jizero |
| 13 | jmptosr |  | 13 | jmptosr |
| 15 | jicarry |  | 8 | jump |
| 16 | xload |  | 1 | load |
| 17 | loadmx |  | 17 | loadmx |
| 18 | xcomp |  | 30 | lshift |
| 19 | yload |  | 50 | nop |
| 20 | ystore |  | 6 | or |
| 21 | pause |  | 21 | pause |
| 22 | printd |  | 38 | print |
| 23 | return |  | 37 | printb |
| 24 | push |  | 39 | printch |
| 25 | pull |  | 22 | printd |
| 26 | xpush |  | 25 | pull |
| 27 | xpull |  | 24 | push |
| 28 | xinc |  | 36 | retfint |
| 29 | xdec |  | 23 | return |
| 30 | lshift |  | 31 | rshift |
| 31 | rshift |  | 0 | stop |
| 32 | cset |  | 2 | store |
| 33 | cclear |  | 4 | sub |
| 34 | getkey |  | 47 | swapas |
| 35 | wait |  | 44 | swapax |
| 36 | retfint |  | 45 | swapay |
| 37 | printb |  | 46 | swapxy |
| 38 | print |  | 35 | wait |
| 39 | printch |  | 18 | xcomp |
| 40 | ypush |  | 29 | xdec |
| 41 | ypull |  | 28 | xinc |
| 42 | yinc |  | 16 | xload |
| 43 | ydec |  | 27 | xpull |
| 44 | swapax |  | 26 | xpush |
| 45 | swapay |  | 43 | ydec |
| 46 | swapxy |  | 42 | yinc |
| 47 | swapas |  | 19 | yload |
| 48 | intenable |  | 41 | ypull |
| 49 | intdisable |  | 40 | ypush |
| 50 | nop |  | 20 | ystore |

# Appendix D: The LEDs

## Flashing lights

It is possible for your computer to have a set of 10 LEDs (light emitting diodes) attached to the parallel port at the back.

It is possible to make these lights switch on or off, according to binary words sent out along the computer’s parallel port.

Note however, the I/O port on your computer works with 8-bit bytes, and so although there are 10 LEDs attached, only the rightmost 8 of these will ever light up, i.e. the left 2 LEDs will NEVER LIGHT, and so the 2 most significant bits will never affect the LEDs.

## The parallel out address

To send binary words to the parallel port on your computer in the CECIL language, you need to refer to the memory map of the SIM20 computer. From this we can see that memory location 1020 is the parallel out register.

This means that any binary word stored in this location is sent out along to any device (such as the LEDs) connected to the parallel port on the computer.

The simplest program to use the LEDs loads a word into the ACCUMULATOR and then stores it at location 1020, i.e.:

**program a simple LED program**

**author Matt Smith**

**date April 1996**

**;------- data --------**

**.num1 insert 1023**

**;------- program --------**

**.start load num1**

**store 1020**

**stop**

## A “Knight Rider” program

An interesting “Knight Rider” effect can be produced by a CECIL program, that uses left and right shift instructions, to switch on and off the LEDs from one side then the other.

HINT: Use LSHIFT and RSHIFT to move a bit from left to right

HINT: Slow the program down using “wait”

HINT: Re-use a subroutine when slowing the program down

HINT: Re-use a subroutine when doing lshift/rshift operation

HINT: Slow the program down using a loop

HINT: Slow the program down using character output to the “Output” window

# Appendix E: Language Reference

## The CECIL language

While this chapter contains a reference to the main instructions, special character and label that make up the CECIL language, you should refer to the CECIL program HELP for latest information.

The entries in the chapter are ordered alphabetically, although before the instructions the special characters are described (such as “;” and “<tab>“).

The entries described in this chapter are summarised in the table below.

|  |  |  |
| --- | --- | --- |
| <RETURN> | jizero | .start |
| <SPACE> | jmptosr | stop |
| <TAB> | jump | store |
| ; | load | string |
| . | loadmx | sub |
| add | lshift | wait |
| and | or | xload |
| cclear | print | xcomp |
| comp | printb | xpush |
| cset | printch | xpull |
| eor | printd | xinc |
| getkey | pull | xdec |
| jicarry | push | xstore |
| jineg | return | insert |
| jipos | rshift |  |

## About the entries

Each entry in this chapter is laid out in the same way. There are six pieces of information given for each language element (less for special characters and labels).

These headings appear in the left hand column of the page, and the details for each entry to the right. The six pieces of information are summarised below. Note if common errors are associated with an instruction/ character/label, a WARNING entry may be present to advise how to avoid such problems.

***Instruction*** The name of the entry being described (and any parameters)

***(Label/Character)***

***Flags*** A description of how the value of any flags effect execution (***effected by***), and of any changes to flags after execution of the instruction (***affects***)

***Description*** A description of what the instruction does (or what a special character/label is used for)

***Example(s)*** One or more examples of the use of the instruction/character/label

***Related instructions*** A list of any related instructions/characters/labels

Language reference

***Character*** <anything> <RETURN> <anything>   
(carriage return key press, generating a carriage return and linefeed)

***Description*** CECIL ignores all <RETURNS> in a program, so they may be used freely to improve program readability.

***Example(s)*** .**start load .num1**  
 is compiled the same as   
**.start  
  
 load  
 .num1**

***Related instructions*** none

***Character*** <anything> <SPACE> <anything>

***Description*** CECIL ignores all extra spaces in a program. However, spaces are used as SEPARATORS between labels, instruction names and data or other labels. Wherever one space can be used, any number can be used (as long as at least one).

***Example(s)*** .**start load .num1**  
 is compiled the same as   
**.start load num1**

***\*\* WARNING \*\**** Do **NOT** leave spaces between the full stop and label name

***Related instructions*** none

***Character*** <anything> <RETURN> <anything> (“Tab” key press)

***Description*** There are no tab characters in the CECIL editor, so each time the tab key is pressed, a set of spaces is added (depending on the currently default level of indentation). Since one or many spaces is considered the same thing in CECIL, this means some reasonable formatting of programs is possible.

***Example(s)*** .**start load .num1**  
 is compiled the same as   
 **.start load .num1**

***Related instructions*** none

***Character*** ; (semicolon)

***Syntax*** <anything> ; <text of a comment>

***Description*** This command tells CECIL to ignore any text to the right,   
i.e. it allows textual comments to be inserted into a program.   
  
COROLLARY: Therefore comments should come at the end of the line or have a whole line dedicated to them.

***Example(s)*** **add num1 ;add number at .num1 to current value of ACCUMULATOR   
;---- data begins here -------**

*\*\** ***WARNING \*\**** CECIL will not compile a program that has any comments between an address label and an instruction, so the following lines is **not a permitted use of comments**:   
 **.start ;--- program starts here ---   
 add num2**

***Related instructions*** none

***Character*** .<label name> (full stop, immediately followed by name for an address label)

***Description*** The full stop indicates that the address of the next instruction (including data declaration instructions) is to be labelled with the given name. This label can be referred to as the parameter for parts of programs needing to refer to or change memory locations.

***Example(s)*** .**start load .num1**  
**.num1 insert 24  
.name insert “matt”**

***\*\* WARNING \*\**** Do **NOT** leave spaces between the full stop and label name

***Related instructions*** **insert**

***Instruction*** add <label>

***Description*** This adds the contents of the number at the data label 'Data', the current value of the accumulator (it also adds to this the current value of the carry flag).

***Flags*** effected by: will add the contents of the carry flag (1 or 0) to the result   
affects: carry flag set if total of addition is greater than 1023   
 zero flag set if result of addition is zero

***Example(s)*** **cclear  
add .num1**

***\*\* WARNING \*\**** Addition will be incorrect if the carry flag has not been cleared before addition, so ALWAYS precede “add” with the “cclear” instruction.

***Related instructions*** **cclear, jizero**

***Bit Manipulation***

***Instruction*** and <label>

***Description*** A logical AND is performed between the current value of the ACCUMULATOR and the number at the given address label. The result of the AND is placed in the ACCUMULATOR, and the zero flag set accordingly.   
  
I.e. for each of the 10 bits in the ACCUMULATOR and the labelled address an AND is performed, and where both bits are 1, the new bit for the ACCUMULATOR is 1. If either or both bits are 0, then the new bit is 0. See Appendix G.

***Flags*** effected by: none  
affects: zero flag set if result of AND is zero for all 10 bits

***Example(s)*** **and .num1**

**load .num1 (7) 0000000111 = 7**

**and .num2 (9) 0000001001 = 9**

**print 0000000001 = 1**

(only the '1's column had a '1' for both numbers, so only this column has a ' 1' in the answer)

***Related instructions*** **eor, or, jizero, lshift, rshift**

***Instruction*** cclear

***Description*** This simply operation sets the carry flag to “0” (whatever its previous value).   
  
Since the carry flag is referred to by the “**add**” and “**sub**” operations, it is important to set or clear the flag appropriately before executing those operations.   
  
Always zero the carry flag before an “**add**” statement.

***Flags*** effected by: none  
affects: carry flag set to 0

***Example(s)*** **cclear  
add .num1**

***Related instructions*** **add, cset, sub, jicarry**

***Instruction*** comp <label>

***Description*** The value at the labelled address is compared with the current value of the ACCUMULATOR.  
  
There is no change to the ACCUMULATOR.

***Flags*** effected by: none  
affects: zero flag set if numbers are the same   
 negative flag set if number at labelled address is larger than ACCUMULATOR

***Example(s)*** **load .num1  
comp .num1**

***Related instructions*** **xcomp, jizero, jineg**

***Instruction*** cset

***Description*** This simply operation sets the carry flag to “1” (whatever its previous value).   
  
Since the carry flag is referred to by the “**add**” and “**sub**” operations, it is important to set or clear the flag appropriately before executing those operations.   
  
Always set the carry flag before a “**sub**” statement.

***Flags*** effected by: none  
affects: carry flag set to 1

***Example(s)*** **cset  
sub .num1**

***Related instructions*** **add, cclear, sub, jicarry**

***Bit Manipulation***

***Instruction*** eor <label>

***Description*** This instruction performs a logical "EXCLUSIVE OR" operation between the bits of the current value of the ACCUMULATOR, and the integer stored at the labelled address.   
  
The result of the “EOR” is stored in the ACCUMULATOR (overwriting the previous value).   
  
A logical "EOR" compares each bit of two binary numbers in turn, the result of each comparison is a new bit, a '1' if one of the bits were '1', a '0' otherwise (i.e. 0 if both 0 or both 1). See Appendix G.

***Flags*** effected by: none  
affects: zero flag set if result of “OR” is zero

***Example(s)*** **load .num1 ;; assume .num1 = 9 = 0000001001   
eor .num2 ;; assume .num2 = 12 = 0000001100   
print ;; 0000000101**

***Related instructions*** **and, or, lshift, rshift**

***Instruction*** getkey

***Description*** This simple command will make the computer wait until a key is pressed, then place the ASCII code of that key in the ACCUMULATOR. The ASCII code is also placed in the "keyboard in register" - at memory location 1013.   
  
This command causes an overwriting of any previous value stored in the "keyboard in register".

***Flags*** effected by: none  
affects: none

***Example(s)*** **getkey  
printch**

***Related instructions*** **printch**

***Instruction*** jicarry <label>

***Description*** If the carry flag is set, THEN jump to a different part of the program (change the program counter to the labelled address), otherwise do nothing.

***Flags*** effected by: only executed if carry flag is set   
affects: none (but changes program counter)

***Example(s)*** **jicarry .subroutine1**

***Related instructions*** **jineg, jipos, jizero, jmptosr, jump, return**

***Instruction*** jineg <label>

***Description*** If the negative flag is set, THEN jump to a different part of the program (change the program counter to the labelled address), otherwise do nothing.

***Flags*** effected by: only executed if negative flag is set   
affects: none (but changes program counter)

***Example(s)*** **jineg .subroutine1**

***Related instructions*** **jicarry, jipos, jizero, jmptosr, jump, return**

***Instruction*** jipos <label>

***Description*** If neither the carry or negative flags are set, THEN jump to a different part of the program (change the program counter to the labelled address), otherwise do nothing.

***Flags*** effected by: only executed if **neither** zero flag or negative flag is set   
affects: none (but changes program counter)

***Example(s)*** **jipos .subroutine1**

***Related instructions*** **jicarry, jineg, jizero, jmptosr, jump, return**

***Instruction*** jizero <label>

***Description*** If the zero flag is set, THEN jump to a different part of the program (change the program counter to the labelled address), otherwise do nothing.

***Flags*** effected by: only executed if zero flag is set   
affects: none (but changes program counter)

***Example(s)*** **jizero .subroutine1**

***Related instructions*** **jicarry, jineg, jipos, jmptosr, jump, return**

***Instruction*** jmptosr <label>

***Description*** Pass control to the instructions beginning at the labelled address, however when the next “**return**” instruction is executed, program control will return to the instruction immediately following the “**jmptosr**” instruction.

***Flags*** effected by: none  
affects: none (but changes program counter)

***Example(s)***  **load .num1   
 print   
 jmptosr .sub1   
 print   
 stop   
  
.sub1 load .num2   
 return   
  
.num1 insert 10  
.num2 insert 28**

***\*\* WARNING \*\**** If there are no more “**jmptosr**” commands to return from, a "stack underflow" error will occur, and the program will be terminated.

***Related instructions*** **jicarry, jineg, jipos, jizero, jump, return**

***Instruction*** jump <label>

***Description*** Pass control to the instructions beginning at the labelled address.   
(i.e. change the program counter to the labelled address).

***Flags*** effected by: none  
affects: none (but changes program counter)

***Example(s)*** **jump .subroutine1**

***\*\* WARNING \*\**** Good programming practice states that the use of unconditional jumps should be kept to a minimum (they are like the dreaded “GOTO” in BASIC, Pascal and C). If unconditional jumps have to be used, wherever possible the “**jmptosr**” instruction should be used, since the use of the “**return**” instruction means that the return from the subroutine is very clear.

***Related instructions*** **jicarry, jineg, jipos, jizero, jmptosr, return**

***Instruction*** load <label>

***Description*** This instruction replaces the contents of the ACCUMULATOR with the integer stored at the labelled address.

***Flags*** effected by: none  
affects: none

***Example(s)*** **load .num1 ;;the value 1 is loaded into ACCUMULATOR   
print ;;14 is displayed in output window   
<...>  
.num1 insert 14**

***Related instructions*** **loadmx, store**

***Instruction*** loadmx <label>

***Description*** This instruction is similar to "load", but includes the current value of the X-register when working out from which memory address to copy the data into the accumulator.   
i.e.  
 (1) loadmx first adds the value in the X-register to the memory address of the data label   
 "data"   
 (2) the result is a address reference (offset from "label" by the number in the X-register)  
 (3) it is the value in this offset address that is loaded into the accumulator

***Flags*** effected by: none  
affects: none

***Example(s)*** **xload .num1 ;;the value 1 is loaded in X-register   
loadmx .num2 ;;the value 49 from address “.num3”   
 ;;(i.e. address “.num2” + 1) is loaded into   
 ;;ACCUMULATOR  
print ;;49 is displayed in output window   
<...>  
.num1 insert 1  
.num2 insert 27  
.num3 insert 49**

***Related instructions*** **load, xload, xinc, xdec**

***Bit Manipulation***

***Instruction*** lshift

***Description*** This instruction works on the value in the ACCUMULATOR as a 10-bit binary number. The instruction moves all the bits along one place to the left — the leftmost (most significant) bit being placed into the carry flag, and the old value of the carry flag being added as the new rightmost (least significant) bit. The resultant 10-bit word replaces the previous value of the ACCUMULATOR.   
   
Note, this instruction can also be though of as multiplication by two (although only if most significant bit and carry flag were 0 before the “lshift”).

***Flags*** effected by: the old value of the carry flag becomes the right-most bit   
affects: carry flag set if the most significant bit was set before the “lshift” occurred

***Example(s)*** **load num1   
lshift  
print  
  
load .num1 ;;assume .num1 = 0000001011 = 11   
lshift  
print ;; 0000010110 = 22   
lshift  
print ;; 0000101100 = 44**

***Related instructions*** **and, eor, rshift, or, not**

***Logical Operation***

***(not an instruction)*** not

***Description*** A logical “NOT” applied to a 10-bit word would mean all the 1’s become 0’s, and vice versa.  
There is no “NOT” instruction in CECIL, but one can be achieved by “EOR”ing the number in the ACCUMULATOR with an address containing 1023 (binary = 1111111111).   
  
Alternatively one can subtract the number to be “NOT”ed from 1023.   
  
See Appendix G.

***Example(s)*** **load .num1 ;; assume .num1 = 9 = 0000001001   
eor .num2 ;; assume .num2 = 1023 = 1111111111   
print ;; 1014 1111110110**

***Related instructions*** **and, eor, or, lshift, rshift**

***Bit Manipulation***

***Instruction*** or <label>

***Description*** This instruction performs a logical "OR" operation between the bits of the current value of the ACCUMULATOR, and the integer stored at the labelled address.   
  
The result of the “OR” is stored in the ACCUMULATOR (overwriting the previous value).   
  
A logical "OR" compares each bit of two binary numbers in turn, the result of each comparison is a new bit, a '1' if one or both bits were '1', a '0' otherwise. See Appendix G.

***Flags*** effected by: none  
affects: zero flag set if result of “OR” is zero

***Example(s)*** **load .num1 ;; assume .num1 = 9 = 0000001001   
or .num2 ;; assume .num2 = 12 = 0000001100   
print ;; 0000001101**

***Related instructions*** **and, eor, lshift, rshift**

***Instruction*** print

***Description*** This instruction causes the contents of the ACCUMULATOR to be printed in the output window — in the form of a decimal integer.

***Flags*** effected by: none  
affects: none

***Example(s)*** **load .num1  
print**

***Related instructions*** **printb, printch, printd**

***Instruction*** printb

***Description*** This instruction causes the contents of the ACCUMULATOR to be printed in the output window — in the form of a BINARY number. The ACCUMULATOR is unchanged.   
  
Useful for analysing the "1" or "0" state of each of the 10-bits that make up the ACCUMULATOR.

***Flags*** effected by: none  
affects: none

***Example(s)*** **load .num1  
printb**

***Related instructions*** **print, printch, printd**

***Instruction*** printch

***Description*** This instruction prints to the output window the character with the ASCII code of the integer in the ACCUMULATOR. The ACCUMULATOR is unchanged.

***Flags*** effected by: none  
affects: none

***Example(s)*** **getkey  
printch**

***Related instructions*** **print, printb, printd**

***Instruction*** printd

***Description*** Places a copy of the value of the given address onto the top of the STACK (leaving the value at the address unchanged).   
  
NOTE: Since the value is added to the top of the stack, it will be the first value to be PULLed (unless other PUSHs are made before the next PULL)

***Flags*** effected by: none  
affects: none

***Example(s)*** **load .lowlocation   
printd**

***\*\* WARNING \*\**** This makes programming a little tricky, because you need to know the location of where the low and high words are, to be able to load that location into the accumulator. This is usually done by “hard coding” the addresses into the program, and storing all data before the instructions. See Appendix FFFFFFF for a detailed description of working with double word numbers in CECIL.

***Related instructions*** **load, print, printb, printch**

***Instruction*** push <label>

***Description*** Places a copy of the value of the given address onto the top of the STACK (leaving the value at the address unchanged).   
  
NOTE: Since the value is added to the top of the stack, it will be the first value to be PULLed (unless other PUSHs are made before the next PULL)

***Flags*** effected by: none  
affects: none

***Example(s)*** **push**

***\*\* WARNING \*\**** If the STACK is full (i.e. all reserved addresses are filled with stack values), no new value can be PUSHed onto it, and a "stack overflow" error will occur, causing the program to be terminated.

***Related instructions*** **pull**

***Instruction*** pull

***Description*** Places into the ACCUMULATOR the value at the top of the STACK (and remove that value from the top of the stack).

***Flags*** effected by: none  
affects: none

***Example(s)*** **pull**

***\*\* WARNING \*\**** If the STACK is empty, no value can be PULLed, and a "stack underflow" error will occur, causing the program to be terminated.

***Related instructions*** **push**

***Instruction*** return

***Description*** Return program control (i.e. the program counter) to the instruction following the last JMPTOSR executed.

***Flags*** effected by: none  
affects: none (but changes program counter)

***Example(s)***  **load .num1   
 print   
 jmptosr .sub1   
 print   
 stop   
  
.sub1 load .num2   
 return   
  
.num1 insert 10  
.num2 insert 28**

***\*\* WARNING \*\**** If there are no more JMPTOSR commands to return from, a "stack underflow" error will occur, and the program will be terminated.

***Related instructions*** **jmptosr**

***Bit Manipulation***

***Instruction*** rshift

***Description*** This instruction works on the value in the ACCUMULATOR as a 10-bit binary number. The instruction moves all the bits along one place to the right — the rightmost (least significant) bit being placed into the carry-flag, and the old value of the carry flag being added as the new leftmost (most significant) bit. The resultant 10-bit word replaces the previous value of the ACCUMULATOR.   
   
Note, this instruction can also be though of as integer division by two (i.e. result is the number of times 2 goes into the value of the ACCUMULATOR), except when the carry flag was 1.

***Flags*** effected by: old value of carry flag becomes new left-most bit   
affects: zero flag set if result is zero   
 carry flag set to previous right-most bit

***Example(s)*** **load num1   
rshift  
print  
  
load .num1 ;;assume .num1 = 0000001011 = 11   
rshift  
print ;; 0000000101 = 5   
rshift  
print ;; 0000000010 = 2**

***Related instructions*** **and, eor, lshift, or**

***Special label*** .start

***Description*** This is not n instruction, but a special label, which tells CECIL where the program listing begins.  
If the ".start" label is not used, then the CECIL default is that the program starts at address 0000.  
  
The value of “.start” (or 0000 if there is no “.start” label) is stored at location 1023 after a program is successfully compiled, and becomes the first value of the program counter when a program is run.

***Flags*** effected by: none  
affects: none

***Example(s)*** **.start load .num**

***\*\* WARNING \*\**** It is poor programming practice to write a program that does not have a “.start” label.   
Good programming practice states that the data and instructions of a program should be in two separate halves, with either all the data first, then the program instructions and any subroutines, or vice versa.

***Related instructions*** **stop**

***Instruction*** store <label>

***Description*** This command will place a copy of the current value in the accumulator at the labelled address (replacing any existing value there).

***Flags*** effected by: none  
affects: none

***Example(s)*** **readkey  
store .asc\_key**

***Related instructions*** **load, xstore**

***Instruction*** stop

***Description*** This simple command tells CECIL to stop executing the program.   
Note that a program may have more than one stop instruction, although good programming practice recommends each program (and subroutine) has only a single entry point, and a single exit point.

***Flags*** effected by: none  
affects: none

***Example(s)*** **stop**

***\*\* WARNING \*\**** Every program should have at least one “stop” statement ! A program without a stop statement must either be in an infinite loop, or it will run into data, and start interpreting the numbers of data in memory as if they were instructions.

***Related instructions*** **.start**

***Instruction*** sub <label>

***Description*** The number at the labelled address is subtracted from the current value of the accumulator, leaving the result in the accumulator.

***Flags*** effected by: the current value of the carry flag, this is subtracted as well as the number stored  
 at the labelled address   
affects: zero flag set if result is zero   
 negative flag set if result is negative

***Example(s)*** **cset  
sub .num1**

***\*\* WARNING \*\**** The carry flag must be set to 1 (using a “cset”) immediately before each “sub” instruction, for a correct two’s compliment calculation.

***Related instructions*** **cset**

***Instruction*** wait

***Description*** The computer will sit and do nothing, for a given number of thousandths of a second, as specified by the current value in the ACCUMULATOR.

***Flags*** effected by: none  
affects: none

***Example(s)*** **load .pause ;load 500 into ACCUMULATOR   
wait ;wait 0.5 seconds   
<...>  
.pause insert 500**

***Related instructions*** **load**

***Data declaration***

***instruction*** insert <integer>  
(usually: <label> insert <integer>)

***Description*** Instructs the compiler to store the given integer as a 10-bit word in memory — an address label is optional (but usually present).

***Flags*** effected by: none  
affects: none

***Example(s)***  **insert 56  
.num2 insert 27**

***Related instructions***

***Instruction*** xcomp <label>

***Description*** The value at the labelled address is compared with the current value of the X-register (rather than with the ACCUMULATOR for the “comp” instruction).   
  
The X-register is unchanged.

***Flags*** effected by: none  
affects: zero flag set if numbers are the same   
 negative flag set if number at labelled address is larger than value in X-register

***Example(s)*** **xcomp .num1**

***Related instructions*** **comp, jizero, jineg, loadmx, xdec, xload, xinc, xpull, xpush, xstore**

***Instruction*** xdec

***Description*** This command subtracts 1 from the current value of the X-register. If this results in a zero or negative result, the appropriate flag is set.

***Flags*** effected by: none  
affects: sets zero flag if new X-register value is zero   
 sets negative flag if new X-register value is negative

***Example(s)*** **xdec**

***Related instructions*** **jineg, jizero, loadmx, xcomp, xload, xinc, xpush, xpull, xstore**

***Instruction*** xinc

***Description*** This command adds 1 to the current value of the X-register.

***Flags*** effected by: none  
affects: sets carry flag if result would have been greater than 1023

***Example(s)*** **xinc**

***Related instructions*** **jicarry, loadmx, xcomp, xdec, xload, xpush, xpull, xstore**

***Instruction*** xload <label>

***Description*** A copy of the number stored in the labelled address is copied into the X-register.

***Flags*** effected by: none  
affects: none

***Example(s)*** **xload .num1**

***Related instructions*** **load, loadmx, xcomp, xdec, xinc, xpull, xpush, xstore**

***Instruction*** xpull

***Description*** The value on the top of the stack is placed into the X-register (and removed from the top of the stack).

***Flags*** effected by: none  
affects: none

***Example(s)*** **xpull**

***\*\* WARNING \*\**** If the stack was empty, a “stack underflow” error will occur.

***Related instructions*** **loadmx, xcomp, xdec, xload, xinc, xpush, xstore**

***Instruction*** xpush

***Description*** A copy of the current value of the X register is pushed (copied) onto the top of the stack (leaving the X register unchanged).

***Flags*** effected by: none  
affects: none

***Example(s)*** **xpush**

***Related instructions*** **loadmx, xcomp, xdec, xload, xinc, xpull, xstore**

***Instruction*** xstore <label>

***Description*** The current value of the X-register is copied into the labelled address (the X-register is unchanged).

***Flags*** effected by: none  
affects: none

***Example(s)*** **xstore .num1**

***Related instructions*** **store, loadmx, xcomp, xdec, xload, xinc, xpull, xpush**

# Appendix F: ASCII character set

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ***code*** | ***char*** | ***code*** | ***char*** | ***code*** | ***char*** | ***code*** | ***char*** |
|  |  |  |  |  |  |  |  |
| 0 | NULL | 32 | <space> | 64 | @ | 96 | £ |
| 1 |  | 33 | ! | 65 | A | 97 | a |
| 2 |  | 34 | “ | 66 | B | 98 | b |
| 3 |  | 35 | # | 67 | C | 99 | c |
| 4 |  | 36 | $ | 68 | D | 100 | d |
| 5 |  | 37 | % | 69 | E | 101 | e |
| 6 |  | 38 | & | 70 | F | 102 | f |
| 7 | <bell> | 39 | ‘ | 71 | G | 103 | g |
| 8 | <backspace> | 40 | ( | 72 | H | 104 | h |
| 9 | <tab> | 41 | ) | 73 | I | 105 | i |
| 10 | <linefeed> | 42 | \* | 74 | J | 106 | j |
| 11 | <home> | 43 | + | 75 | K | 107 | k |
| 12 | <formfeed> | 44 | , | 76 | L | 108 | l |
| 13 | <carrreturn> | 45 | - | 77 | M | 109 | m |
| 14 |  | 46 | . | 78 | N | 110 | n |
| 15 |  | 47 | / | 79 | O | 111 | o |
| 16 |  | 48 | 0 | 80 | P | 112 | p |
| 17 |  | 49 | 1 | 81 | Q | 113 | q |
| 18 |  | 50 | 2 | 82 | R | 114 | r |
| 19 |  | 51 | 3 | 83 | S | 115 | s |
| 20 | <paragraph> | 52 | 4 | 84 | T | 116 | t |
| 21 | <section> | 53 | 5 | 85 | U | 117 | u |
| 22 |  | 54 | 6 | 86 | V | 118 | v |
| 23 |  | 55 | 7 | 87 | W | 119 | w |
| 24 |  | 56 | 8 | 88 | X | 120 | x |
| 25 |  | 57 | 9 | 89 | Y | 121 | y |
| 26 | <eofile> | 58 | : | 90 | Z | 122 | z |
| 27 | <escape> | 59 | ; | 91 | [ | 123 | { |
| 28 |  | 60 | < | 92 | \ | 124 | | |
| 29 |  | 61 | = | 93 | ] | 125 | } |
| 30 |  | 62 | > | 94 | ^ | 126 | ~ |
| 31 |  | 63 | ? | 95 | \_ | 127 | <del> |
|  |  |  |  |  |  |  |  |

# Appendix G: Register and Flag Virtual Addresses

1024 Program Counter

1025 Status register

1026 Accumulator

1027 X Register

1028 Y Register

The next table gives the bit numbers for flags in status byte.

1 Zero flag

2 Negative flag

4 Carry flag

# Appendix H: Input and Output on the SIM20

The aim of our course is to introduce you to the basic design principles of computer systems in general. However, CECIL has a slightly different aim; we want to get you up and running in a low level language as quickly and simply as possible. The way CECIL appears to you will depend on how the SIM20 computer that it runs on is designed; we have therefore designed the SIM20 to make your entry into CECIL programming as simple as possible. But this means that the SIM20 works a little differently from the types of computer covered in the overview given at the outset. The follow notes are designed to help you to understand these differences.

## Basic Design Principles

There are five basic principles involved in the design of the SIM20:

1. The design should enable the newcomer to CECIL to write simple introductory programs.
2. To achieve the above, as much work should be done by the hardware as possible, avoiding the need for complex coding to achieve I/O operations for example.
3. Transfers are therefore initiated by simply reading or writing to the relevant register, rather than the programmer having to handle complicated handshaking procedures.
4. From the outset, space was provided for status registers to be implemented, as in most conventional computers. However, the above design principles mean that it has not been necessary to implement them, so the SIM20 doesn’t use them.
5. Similarly, the original design for the SIM20 made provision for interrupt programming to be implemented as an extension for those who required it .

Remember that CECIL is designed to give a solid introduction to low-level programming that everyone can cope with, not as a comprehensive exemplar of all possible low-level programming concepts.

It is important to realise that CECIL and the SIM20 are designs rather than physical implementations, and that they are simulated on other computers. This means that different implementations exist, and that the value of these implementations depends on the extent to which they have followed the design specification. What follows addresses the design specification, so it should apply to any implementation.

The way these principles work out in practice will now be addressed.

***The Keyboard Interface***

The keyboard interface is accessed through the single location 1013, the keyboard input register. Notionally, it also uses the keyboard status register, but this is not used as explained above. Since there is only one register for accessing bytes representing the keys that have been pressed, this means that there is quite a bit of work for the keyboard interface to do.

There are five things you need to know about the keyboard interface:

1. The interface maintains its own input buffer. This means that keypresses are recorded by the hardware, not the software as in most other computers. The hardware then maintains this buffer, presenting the next keypress in the buffer to the SIM20 via the keyboard input register.
2. The length of this input buffer is undefined in the SIM20 specification, although in the current version it is 1 character.
3. The action of reading the keyboard input register (1013) transfers the next character out of the input buffer maintained by the hardware; it is the responsibility of the programmer to ensure that the character is not lost - i.e. you must store it somewhere safely once you’ve read it!
4. You cannot “push” characters back onto the keyboard buffer to keep them safe - writes to the keyboard input register are ignored.
5. If there are no more keypresses left in the input buffer, the interface presents the value 0 through the keyboard input register.

You already know the remaining details; for example, all keypresses are coded and presented in ASCII code. As in all other computers, keypresses can only be accessed a byte at a time; it is only in high-level languages that it is possible to input a line at a time, or to have it presented as structured data such as integers.

This has implications for how CECIL works at the programming level. There are two ways of accessing keyboard data:

1. The SIM20 processor has a *getkey* instruction defined in its instruction set which gets the next keypress from the queue as presented at the keyboard input register (1013). If the buffer is empty - which it usually is since the SIM20 works very fast - then the *getkey* instruction **waits for the next keypress to occur** before allowing program execution to continue. This is what you would normally expect. However, what you wont have realised is that it is possible to type ahead if you write a program that takes a long time to get to the input routine for any reason.
2. It is also possible to read the keyboard input register directly, however. Be careful; you need to understand the effect that this will have. Firstly, as explained above, the action of reading the input port (1013) will “pop” the next keypress off the buffer maintained by the hardware. Once you’ve read it you must do something with it, or the character will be lost. However, this approach does give you the opportunity to test the keyboard without holding up program execution.

In practical terms, this means that you have two possibilities open to you. If you want some user data from the keyboard, and that must be obtained before continuing, then use *getkey* in the normal way. But if you want to use the keyboard as an (optional) means of altering a program as it runs (games programs are prime examples of these, but there are other uses), then you can simply *load 1013*. This will read any key that’s been pressed, or return 0 if no key has been pressed.

Another use of using the *load 1013* approach is in enabling you to clear out the keyboard buffer if the typing ahead problem should arise for any reason. You simply need to write a loop that will keep *load*ing *1013* until it returns 0. This then means that the keyboard buffer is empty, and you can go ahead and read the keyboard as normal, say through *getkey*.

***The Video Interface***

The video interface is accessed through the single location 1015, although there is also the notional video status register which has not been implemented. Main principles are as follows:

1. The video output is simply a continuous stream that is sent to the output device. This is (unfortunately) undefined, but is probably a 40 or 80 column monitor screen.
2. The data stream consists of ASCII characters. The main confusion lies in whether a particular implementation requires a CR code to give a new line or a CR-LF combination to achieve it. You will unfortunately have to determine this by trial and error for your particular implementation.
3. Colour and graphics are not supported other than possible block graphics in the extended ASCII character set.
4. The extended character set (characters 128 to 255) is also undefined and will need to be determined by trial and error on a particular implementation, if indeed it exists at all.
5. Writes to the video output register initiate a transfer to the video interface.
6. Reads from the video interface give an undefined result, not necessarily the last character sent; this is a result of the powerful SIM20 instructions provided as explained below.

You don’t need to deal with the video interface at all, although the above information is useful. The SIM20 processor provides a powerful group of instructions which do far more than just writing characters to the video interface. *Print* prints numerical values from the accumulator to the screen in ASCII format as required by the interface doing all the necessary conversions for you, whilst *printch* does a simple print in ASCII format. *Printb* prints in binary, whilst *printd* prints in double-byte format. You have no need to write to 1015 (the video output register), since *printch* does exactly the same.

***The Serial Interface***

The serial interface consists of two registers, the serial output register (1017) and the serial input register (1018), together with the serial status register which is not implemented. The principles of operation are as follows.

1. Writes to the serial output register initiate a transfer to the SIM20’s serial output port, with all handshaking handled by the hardware.
2. Reads from the serial input register initiate a transfer from the SIM20’s serial input port.
3. Writes to the serial input register are ignored.

What actually happens in a specific simulation of the SIM20 depends on the configuration of the computer you are using. Although CECIL itself can take control of the various ports available, this can be overridden by, for example, Windows software which might redirect input and output to various ports, perhaps to a network printer, for example.

***The Parallel Interface***

The parallel interface consists of a single register, the parallel output register (1020), together with the parallel input register (1021) and the parallel status register (1022) which are not implemented. The principles of operation are as follows.

1. Writes to the parallel output register initiate a transfer to the SIM20’s parallel output port, with all handshaking handled by the hardware.
2. Reads from, and writes to, the parallel input register are (currently) ignored.

Once again, what happens with a specific simulation of the SIM20 depends on the configuration of the computer you are using. Although CECIL itself can take control of the various ports available, this can be overridden by, for example, Windows software which might redirect input and output to various ports, perhaps to a network printer, for example.

***The Sound Card*** *(\*not currently implemented)*

The sound card is a later addition to the SIM20 specification. It has been added in the section of memory that was originally “reserved for expansion”. The interface consists of two registers, the pitch register (1011) and the duration register (1012). The simple way to use it is as follows:

1. Decide on a pitch for the note you want to play. Middle C is defined to require the value 52; the pitch increases or decreases by a semitone if this value is increased or decreased by 4.
2. Store your chosen value in the pitch register (1011).
3. Decide on a duration for your sound. A value of 20 gives a sound 1 second long.
4. Store your chosen value in the duration register (1012).

Your note will now play through the SIM20’s sound system. Although this has a good chance of working on a PC simulation of the SIM20 - even without a sound card! - it does still depend on how the computer you are using is set up.

The following CECIL program plays middle C for half a second.

program test sound1

author davida

date 15.05.97

.start load pitch

store 1011

load durn

store 1012

stop

.pitch insert 52

.durn insert 10

There are some principles of operation for the sound card which are worth knowing:

1. The sound card maintains a queue of sounds. It doesn’t matter how fast your CECIL program writes to the sound card registers, the notes will queue up and sound for the required duration.
2. If you need to program in a break in the sounds, it is therefore necessary to include a specified length of silence in the sound queue. The way to do this is explained in the following section.
3. If your program writes to the sound card interface and then does something else, you will find that the sound card goes on playing whilst your program does other things (I think… at least, that’s the idea…).

There are some additional facilities for more advanced work also provided by the sound card interface. These are as follows.

1. The pitch register (1011) allows values for the pitch to be given that go from 0 to 255. This only uses the bottom eight bits of the register. Bits 8 and 9 (the top two bits) are used to define an envelope; this means that four envelopes are possible with values from 0 to 3. 0 is the default, and used if you stick to values of pitch from 0 to 255. Other values invoke different sound envelopes; these change the way the note sounds. The envelopes are undefined, but may give siren effects or a tremolo, for example.
2. The duration register (1012) similarly only allows values from 0 to 255 for duration. **Note:** the value 255 has a special meaning, producing a note that sounds continuously until specific action is taken to stop it. You probably want to avoid this unless you have a special use for it and take steps to stop it afterwards. See the flush control below.
3. Bit 8 of the duration register (with a value of 256) is the “flush” control. If this bit is set (one way to do this is to use a value for the duration from 256 to 510) then all previous notes in the sound queue are flushed and the new note begins immediately.
4. Bit 9 of the duration register (with a value of 512) is the “silence” bit. If this bit is set, the value in the pitch register is effectively ignored, and the sound card plays silence for the specified duration instead of a note. This is how specific gaps can be included in a sound sequence.

If you read from either of the sound card registers, they return a value that is the last value written to them.

The following program plays the first line of “Three Blind Mice”.

program sound3

author davida

date 15.05.97

.start xload notes

.loop loadmx notes

store 1011

xdec

loadmx notes

store 1012

xdec

jipos loop

stop

.notes insert 6

insert 20

insert 52

insert 10

insert 60

insert 10

insert 68

OK, so there are no comments in it and there ought to be (do as I say, not as I do…). The only thing to be especially careful of is the fact that the x register loop effectively runs backwards (easier to manage, with fewer lines of code), so the data consists of pitch/duration pairs running *backwards* towards the **notes** label in the data. The value 6 in the location actually labelled **notes** indicates that there are six items of data consisting of three pitch/duration pairs.

A final example is virtually identical, except it invokes an envelope by using pitch values greater than 255:

program sound3

author davida

date 15.05.97

.start xload notes

.loop loadmx notes

store 1011

xdec

loadmx notes

store 1012

xdec

jipos loop

stop

.notes insert 6

insert 20

insert 308

insert 10

insert 316

insert 10

insert 324

If you are typing the example programs in, don’t retype this! Simply edit the previous version to give the new values for the pitch. You should find that the resulting effect is quite different as a result of the change of envelope.

From here on, it’s down to you to experiment and enjoy. There is plenty you can do with the sound card - it’s a suitable subject for a project all on its own. If you want to make good use of it, you really ought to write some utility routines to make it easier to play the various notes and to include silences or to use envelopes without having to do complicated sums all the time!

1. An assembler is a program for compiling assembly language statements into machine code (i.e. into numbers stored at addresses). Before assemblers existed low level programs were still written in assembly language, but hand compiled, which was tedious and error prone. [↑](#footnote-ref-1)